Бесплатная электронная книга

учусь cmake

Free unaffiliated eBook created from
Stack Overflow contributors.

#cmake

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cmake

It is an unofficial and free cmake ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cmake.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с cmake

замечания

CMake - это инструмент для определения и управления сборками кода, прежде всего для C ++.

CMake - это кросс-платформенный инструмент; идея состоит в том, чтобы иметь единое определение того, как строится проект, - который переводится в конкретные определения построения для любой поддерживаемой платформы.

Это достигается путем сопряжения с различными платформами, специфичными для платформы; CMake - это промежуточный шаг, который генерирует ввод данных для разных конкретных платформ. В Linux CMake генерирует Makefiles; в Windows он может создавать проекты Visual Studio и т. д.

Поведение сборки определяется в файлах `CMakeLists.txt` - по одному в каждом каталоге исходного кода. Файл `CMakeLists` каждого каталога определяет, что должна делать система сборки в этом конкретном каталоге. Он также определяет, какие подкаталоги должны обрабатывать CMake.

Типичные действия включают:

- Создайте библиотеку или исполняемый файл из некоторых исходных файлов в этом каталоге.
- Добавьте путь к пути include-path, который используется во время сборки.
- Определите переменные, которые будет использовать buildsystem в этом каталоге, и в его подкаталогах.
- Создайте файл на основе конкретной конфигурации сборки.
- Найдите библиотеку, которая находится где-то в исходном дереве.

Окончательные файлы `CMakeLists` могут быть очень четкими и понятными, поскольку каждый из них настолько ограничен по объему. Каждый из них обрабатывает столько же сборки, сколько присутствует в текущем каталоге.

Для официальных ресурсов на CMake см. Документацию и учебник CMake.

Версии

Версия	Дата выхода
3,9	2017-07-18
3,8	2017-04-10

Версия	Дата выхода
3,7	2016-11-11
3,6	2016-07-07
3,5	2016-03-08
3,4	2015-11-12
3,3	2015-07-23
3,2	2015-03-10
3,1	2014-12-17
3.0	2014-06-10
2.8.12.1	2013-11-08
2.8.12	2013-10-11
2.8.11	2013-05-16
2.8.10.2	2012-11-27
2.8.10.1	2012-11-07
2.8.10	2012-10-31
2.8.9	2012-08-09
2.8.8	2012-04-18
2.8.7	2011-12-30
2.8.6	2011-12-30
2.8.5	2011-07-08
2.8.4	2011-02-16
2.8.3	2010-11-03
2.8.2	2010-06-28
2.8.1	2010-03-17
2,8	2009-11-13
2,6	2008-05-05

Examples

Установка CMake

Перейдите на страницу загрузки CMake и получите двоичный файл для вашей операционной системы, например Windows, Linux или Mac OS X. В Windows дважды щелкните двоичный файл для установки. В Linux запускается двоичный файл с терминала.

В Linux вы также можете установить пакеты из диспетчера пакетов дистрибутива. На Ubuntu 16.04 вы можете установить командную строку и графическое приложение с помощью:

```
sudo apt-get install cmake
sudo apt-get install cmake-gui
```

В FreeBSD вы можете установить командную строку и графическое приложение на основе Qt с помощью:

```
pkg install cmake
pkg install cmake-gui
```

В Mac OSX, если вы используете один из менеджеров пакетов, доступных для установки вашего программного обеспечения, наиболее заметным из которых является MacPorts (MacPorts) и Homebrew (Homebrew), вы также можете установить CMake через один из них. Например, в случае MacPorts, введите следующие

```
sudo port install cmake
```

установит CMake, в то время как в случае использования ящика Homebrew, который вы наберете

```
brew install cmake
```

После того, как вы установили CMake, вы можете легко проверить, выполнив следующие

```
cmake --version
```

Вы должны увидеть что-то похожее на следующее

```
cmake version 3.5.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Переключение между типами сборки, например отладка и выпуск

CMake знает несколько типов сборки, которые обычно влияют на параметры компилятора и компоновщика по умолчанию (такие как создаваемая отладочная информация) или альтернативные пути кода.

По умолчанию CMake может обрабатывать следующие типы сборки:

- **Отладка** : обычно классическая сборка отладки, включая отладочную информацию, отсутствие оптимизации и т. Д.
- **Выпуск** : типичная версия сборки без отладочной информации и полной оптимизации.
- **RelWithDebInfo::** То же, что и *Release* , но с информацией об отладке.
- **MinSizeRel** : специальная *версия выпуска,* оптимизированная для размера.

Как обрабатываются конфигурации, зависит от используемого генератора.

Некоторые генераторы (например, Visual Studio) поддерживают несколько конфигураций. CMake будет генерировать все конфигурации сразу, и вы можете выбрать из IDE или использовать `--config CONFIG` (с `cmake --build`), какую конфигурацию вы хотите построить. Для этих генераторов CMake будет стараться изо всех сил генерировать структуру каталогов сборки, чтобы файлы из разных конфигураций не наступали друг на друга.

Генераторы, которые поддерживают только одну конфигурацию (например, Unix Make-файлы), работают по-разному. Здесь `CMAKE_BUILD_TYPE` активная конфигурация определяется значением переменной CMake `CMAKE_BUILD_TYPE` .

Например, чтобы выбрать другой тип сборки, можно выполнить следующие команды командной строки:

```
cmake -DCMAKE_BUILD_TYPE=Debug path/to/source
cmake -DCMAKE_BUILD_TYPE=Release path/to/source
```

Сценарий CMake должен избегать установки самого `CMAKE_BUILD_TYPE` , так как обычно это отвечает за ответственность пользователей.

Для генераторов с одним коннектором для переключения конфигурации требуется перезапуск CMake. Последующая сборка, скорее всего, перезапишет объектные файлы, созданные более ранней конфигурацией.

Простой проект «Hello World»

Учитывая исходный файл C ++ `main.cpp` определяющий функцию `main()` , сопровождающий файл `CMakeLists.txt` (со следующим содержимым) будет инструктировать *CMake* для генерации соответствующих инструкций сборки для текущей системы и компилятора C ++ по умолчанию.

main.cpp *(пример C ++ Hello World)*

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

add_executable(app main.cpp)
```

Смотрите, как живут на Coliru

1. `cmake_minimum_required(VERSION 2.4)` устанавливает минимальную версию CMake, необходимую для оценки текущего скрипта.
2. `project(hello_world)` запускает новый проект CMake. Это вызовет много внутренней логики CMake, особенно при обнаружении компилятора C и C ++ по умолчанию.
3. С помощью `add_executable(app main.cpp)` создается целевое `app` для сборки, которое будет вызывать сконфигурированный компилятор с некоторыми стандартными флагами для текущего параметра для компиляции исполняемого `app` из данного исходного файла `main.cpp` .

Командная строка *(In-Source-Build, не рекомендуется)*

```
> cmake .
...
> cmake --build .
```

`cmake .` обнаруживает компилятор, оценивает `CMakeLists.txt` в данном `.` и генерирует среду сборки в текущем рабочем каталоге.

`cmake --build .` команда является абстракцией для необходимого вызова build / make.

Командная строка *(рекомендуется использовать Out-of-Source)*

Чтобы ваш исходный код был чистым от каких-либо артефактов сборки, вы должны делать сборки «вне источника».

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

Или CMake также может абстрагировать основные команды оболочки платформы из

примера выше:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

«Hello World» с несколькими исходными файлами

Сначала мы можем указать каталоги файлов заголовков `include_directories()` , тогда нам нужно указать соответствующие исходные файлы целевого исполняемого файла с помощью `add_executable()` и убедиться, что в исходных файлах есть только одна функция `main()` .

Ниже приведен простой пример: все файлы предполагаются помещенными в каталог `PROJECT_SOURCE_DIR` .

main.cpp

```
#include "foo.h"

int main()
{
    foo();
    return 0;
}
```

foo.h

```
void foo();
```

foo.cpp

```
#include <iostream>
#include "foo.h"

void foo()
{
    std::cout << "Hello World!\n";
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_executable(app main.cpp foo.cpp)  # be sure there's exactly one main() function in the
source files
```

Мы можем следовать той же процедуре в приведенном выше примере, чтобы построить

наш проект. Затем выполнение `app` распечатает

```
>./app
Hello World!
```

«Привет мир» в качестве библиотеки

В этом примере показано, как развернуть программу «Hello World» в качестве библиотеки и как связать ее с другими объектами.

Скажем, у нас есть тот же набор исходных / заголовочных файлов, что и в примере http://www.riptutorial.com/cmake/example/22391/-hello-world--with-multiple-source-files . Вместо создания из нескольких исходных файлов мы можем сначала развернуть `foo.cpp` как библиотеку с помощью `add_library()` а затем связать ее с основной программой с помощью `target_link_libraries()` .

Мы модифицируем **CMakeLists.txt** для

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_library(applib foo.cpp)
add_executable(app main.cpp)
target_link_libraries(app applib)
```

и после тех же шагов мы получим тот же результат.

Прочитайте Начало работы с cmake онлайн: https://riptutorial.com/ru/cmake/topic/862/начало-работы-с-cmake

глава 2: Добавить каталоги в компилятор Включить путь

Синтаксис

- include_directories ([ПОСЛЕ | ПЕРЕД] [SYSTEM] dir1 [dir2 ...])

параметры

параметр	Описание
`dirN`	один или более относительных или абсолютных путей
`AFTER` , `BEFORE`	(необязательно), следует ли добавлять указанные каталоги в начало или конец текущего списка включенных путей; поведение по умолчанию определяется `CMAKE_INCLUDE_DIRECTORIES_BEFORE`
`SYSTEM`	(необязательно) сообщает компилятору протестировать указанные каталоги, поскольку *система включает dirs* , что может вызвать специальную обработку компилятором

Examples

Добавить подкаталог проекта

Учитывая следующую структуру проекта

```
include\
  myHeader.h
src\
  main.cpp
CMakeLists.txt
```

следующая строка в файле `CMakeLists.txt`

```
include_directories(${PROJECT_SOURCE_DIR}/include)
```

суммирует `include` каталог в *путь поиска* компилятора для всех целей , определенных в этой директории (и все его подкаталоги включены через `add_subdirectory()`).

Таким образом, файл `myHeader.h` в подкаталоге `include` проекта можно включить через `#include "myHeader.h"` в файле `main.cpp` .

Прочитайте Добавить каталоги в компилятор Включить путь онлайн:
https://riptutorial.com/ru/cmake/topic/5968/добавить-каталоги-в-компилятор-включить-путь

глава 3: Иерархический проект

Examples

Простой подход без пакетов

Пример, который создает исполняемый файл (редактор) и связывает библиотеку (выделение) с ней. Структура проекта проста, для этого требуется мастер CMakeLists и каталог для каждого подпроекта:

```
CMakeLists.txt
editor/
    CMakeLists.txt
    src/
        editor.cpp
highlight/
    CMakeLists.txt
    include/
        highlight.h
    src/
        highlight.cpp
```

Мастер CMakeLists.txt содержит глобальные определения и вызов `add_subdirectory` для каждого подпроекта:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

add_subdirectory(highlight)
add_subdirectory(editor)
```

CMakeLists.txt для библиотеки назначает источники и включает в себя каталоги. Используя `target_include_directories()` вместо `include_directories()` `target_include_directories()` dirs будут распространяться на пользователей библиотеки:

```
cmake_minimum_required(VERSION 3.0)
project(highlight)

add_library(${PROJECT_NAME} src/highlight.cpp)
target_include_directories(${PROJECT_NAME} PUBLIC include)
```

CMakeLists.txt для приложения назначает источники и связывает выделенную библиотеку. Пути к бинарнику и включают в себя автоматическое управление cmake:

```
cmake_minimum_required(VERSION 3.0)
project(editor)

add_executable(${PROJECT_NAME} src/editor.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC highlight)
```

Прочитайте Иерархический проект онлайн: https://riptutorial.com/ru/cmake/topic/1443/иерархический-проект

глава 4: Интеграция CMake в инструментах CI GitHub

Examples

Настройте Travis CI с запасом CMake

У Travis CI установлен предустановленный CMake 2.8.7.

Минимальный скрипт `.travis.yml` для сборки вне источника

```
language: cpp

compiler:
  - gcc

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Настройте Travis CI с новейшим CMake

Версия CMake, предварительно установленная на Travis, очень старая. Вы можете использовать официальные Linux-файлы для сборки с более новой версией.

Вот пример `.travis.yml` :

```
language: cpp

compiler:
  - gcc

# the install step will take care of deploying a newer cmake version
install:
  # first we create a directory for the CMake binaries
  - DEPS_DIR="${TRAVIS_BUILD_DIR}/deps"
  - mkdir ${DEPS_DIR} && cd ${DEPS_DIR}
  # we use wget to fetch the cmake binaries
  - travis_retry wget --no-check-certificate https://cmake.org/files/v3.3/cmake-3.3.2-Linux-
```

```
x86_64.tar.gz
  # this is optional, but useful:
  # do a quick md5 check to ensure that the archive we downloaded did not get compromised
  - echo "f3546812c11ce7f5d64dc132a566b749 *cmake-3.3.2-Linux-x86_64.tar.gz" > cmake_md5.txt
  - md5sum -c cmake_md5.txt
  # extract the binaries; the output here is quite lengthy,
  # so we swallow it to not clutter up the travis console
  - tar -xvf cmake-3.3.2-Linux-x86_64.tar.gz > /dev/null
  - mv cmake-3.3.2-Linux-x86_64 cmake-install
  # add both the top-level directory and the bin directory from the archive
  # to the system PATH. By adding it to the front of the path we hide the
  # preinstalled CMake with our own.
  - PATH=${DEPS_DIR}/cmake-install:${DEPS_DIR}/cmake-install/bin:$PATH
  # don't forget to switch back to the main build directory once you are done
  - cd ${TRAVIS_BUILD_DIR}

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Прочитайте Интеграция CMake в инструментах CI GitHub онлайн:
https://riptutorial.com/ru/cmake/topic/1445/интеграция-cmake-в-инструментах-ci-github

глава 5: Использование CMake для настройки тегов препроцессора

Вступление

Использование CMake в проекте C ++ при правильном использовании может позволить программисту меньше сконцентрироваться на платформе, номере версии программы и т. Д. На самой реальной программе. С помощью CMake вы можете определить теги препроцессора, которые позволяют легко проверить, какая платформа или любые другие теги препроцессора вам могут понадобиться в реальной программе. Например, номер версии, который можно использовать в системе журналов.

Синтаксис

- #define preprocessor_name "@ cmake_value @"

замечания

Важно понимать, что не каждый препроцессор должен быть определен в `config.h.in` . Теги препроцессора обычно используются только для облегчения работы программистов и должны использоваться с осторожностью. Вы должны исследовать, существует ли тег препроцессора, прежде чем определять его, поскольку вы можете столкнуться с неопределенным поведением в разных системах.

Examples

Использование CMake для определения номера версии для использования на C ++

Возможности безграничны. поскольку вы можете использовать эту концепцию, чтобы вытащить номер версии из вашей системы сборки; например, git и использовать этот номер версии в вашем проекте.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(project_name VERSION "0.0.0")

configure_file(${path to configure file 'config.h.in'}
include_directories(${PROJECT_BINARY_BIN}) // this allows the 'config.h' file to be used
throughout the program
```

```
...
```

config.h.in

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD

#define PROJECT_NAME "@PROJECT_NAME@"
#define PROJECT_VER  "@PROJECT_VERSION@"
#define PROJECT_VER_MAJOR "@PROJECT_VERSION_MAJOR@"
#define PROJECT_VER_MINOR "@PROJECT_VERSION_MINOR@"
#define PTOJECT_VER_PATCH "@PROJECT_VERSION_PATCH@"

#endif // INCLUDE_GUARD
```

main.cpp

```
#include <iostream>
#include "config.h"
int main()
{
    std::cout << "project name: " << PROJECT_NAME << " version: " << PROJECT_VER << std::endl;
    return 0;
}
```

выход

```
project name: project_name version: 0.0.0
```

Прочитайте Использование CMake для настройки тегов препроцессора онлайн: https://riptutorial.com/ru/cmake/topic/10885/использование-cmake-для-настройки-тегов-препроцессора

глава 6: Компилировать функции и стандартный выбор C / C ++

Синтаксис

- target_compile_features (*target* **PRIVATE | PUBLIC | INTERFACE** *feature1* [*feature2* ...])

Examples

Компиляция требований к характеристикам

Необходимые функции компилятора могут быть указаны в целевом режиме с помощью команды target_compile_features :

```
add_library(foo
    foo.cpp
)
target_compile_features(foo
    PRIVATE          # scope of the feature
    cxx_constexpr    # list of features
)
```

Эти функции должны быть частью CMAKE_C_COMPILE_FEATURES или CMAKE_CXX_COMPILE_FEATURES ; cmake сообщает об ошибке в противном случае. Cmake добавит все необходимые флаги, такие как `-std=gnu++11` в параметры компиляции цели.

В этом примере функции объявляются `PRIVATE` : требования будут добавлены к цели, но не к ее потребителям. Чтобы автоматически добавлять требования к целевому зданию против foo, вместо `PRIVATE` следует использовать `PUBLIC` или `INTERFACE` :

```
target_compile_features(foo
    PUBLIC    # this time, required as public
    cxx_constexpr
)

add_executable(bar
    main.cpp
)
target_link_libraries(bar
    foo       # foo's public requirements and compile flags are added to bar
)
```

Выбор версии C / C ++

Требуемая версия для C и C ++ может быть указана глобально, используя, соответственно,

переменные `CMAKE_C_STANDARD` (принятые значения: 98, 99 и 11) и `CMAKE_CXX_STANDARD` (принятые значения: 98, 11 и 14):

```
set(CMAKE_C_STANDARD 99)
set(CMAKE_CXX_STANDARD 11)
```

Они добавят необходимые параметры компиляции для целей (например, `-std=c++11` для gcc).

Версия может быть выполнена, установив для `ON` переменные `CMAKE_C_STANDARD_REQUIRED` и `CMAKE_CXX_STANDARD_REQUIRED` соответственно.

Переменные должны быть установлены до создания цели. Версия также может быть указана для каждой цели:

```
set_target_properties(foo PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED ON
)
```

Прочитайте Компилировать функции и стандартный выбор C / C ++ онлайн: https://riptutorial.com/ru/cmake/topic/5297/компилировать-функции-и-стандартный-выбор-c---c-plusplus

глава 7: Конфигурации сборки

Вступление

В этом разделе показано использование различных конфигураций CMake, таких как Debug или Release, в разных средах.

Examples

Настройка конфигурации отладки / отладки

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

# Configuration types
SET(CMAKE_CONFIGURATION_TYPES "Debug;Release" CACHE STRING "Configs" FORCE)
IF(DEFINED CMAKE_BUILD_TYPE AND CMAKE_VERSION VERSION_GREATER "2.8")
  SET_PROPERTY(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS  ${CMAKE_CONFIGURATION_TYPES})
ENDIF()

SET(${PROJ_NAME}_PATH_INSTALL     "/opt/project"                    CACHE PATH "This
directory contains installation Path")
SET(CMAKE_DEBUG_POSTFIX "d")

# Install
#-----------------------------------------------------#
INSTALL(TARGETS ${PROJ_NAME}
    DESTINATION  "${${PROJ_NAME}_PATH_INSTALL}/lib/${CMAKE_BUILD_TYPE}/"
    )
```

Выполнение следующих сборок создаст две разные ('/ opt / myproject / lib / Debug' / opt / myproject / lib / Release) папки с библиотеками:

```
$ cd /myproject/build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
$ sudo make install
$ cmake _DCMAKE_BUILD_TYPE=Release ..
$ make
$ sudo make install
```

Прочитайте Конфигурации сборки онлайн: https://riptutorial.com/ru/cmake/topic/8319/конфигурации-сборки

глава 8: Настроить файл

Вступление

configure_file - это функция CMake для копирования файла в другое место и изменения его содержимого. Эта функция очень полезна для создания файлов конфигурации с помощью путей, настраиваемых переменных, с использованием общего шаблона.

замечания

Скопируйте файл в другое место и измените его содержимое.

```
configure_file(<input> <output>
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Копирует файл в файл и заменяет значения переменных, указанные в содержимом файла. Если это относительный путь, он оценивается относительно текущего каталога источника. Должен быть файл, а не каталог. Если это относительный путь, он оценивается по отношению к текущей двоичной директории. Если имена существующего каталога, входной файл помещается в этот каталог с его исходным именем.

Если файл изменен, система сборки повторно запустит CMake для повторной настройки файла и создания системы сборки снова.

Эта команда заменяет любые переменные во входном файле, на которые ссылаются как $ {VAR} или @ VAR @, с их значениями, определенными CMake. Если переменная не определена, она будет заменена ничем. Если COPYONLY указан, то переменное расширение не произойдет. Если указано ESCAPE_QUOTES, любые замещенные кавычки будут экранированы в стиле C. Файл будет настроен с текущими значениями переменных CMake. Если задано значение @ONLY, будут заменены только переменные формы @ VAR @, и $ {VAR} будет проигнорирован. Это полезно для настройки скриптов, использующих $ {VAR}.

Строки входных файлов формы «#cmakedefine VAR ...» будут заменены на «#define VAR ...» или / * #undef VAR * / в зависимости от того, установлен ли VAR в CMake на любое значение, которое не считается ложным константа командой if (). (Содержимое «...», если оно есть, обрабатывается, как указано выше.) Строки входных файлов формы «# cmakedefine01 VAR» будут заменены либо «#define VAR 1», либо «#define VAR 0» аналогичным образом.

С помощью NEWLINE_STYLE можно завершить настройку строки:

```
'UNIX' or 'LF' for \n, 'DOS', 'WIN32' or 'CRLF' for \r\n.
```

COPYONLY нельзя использовать с NEWLINE_STYLE.

Examples

Создайте файл конфигурации с ++ с помощью CMake

Если у нас есть проект с ++, который использует конфигурационный файл config.h с некоторыми настраиваемыми путями или переменными, мы можем сгенерировать его с помощью CMake и общего файла config.h.in.

Config.h.in может быть частью репозитория git, в то время как сгенерированный файл config.h никогда не будет добавлен, поскольку он генерируется из текущей среды.

```
#CMakeLists.txt
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)

SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

SET(${PROJ_NAME}_DATA     ""     CACHE PATH "This directory contains all DATA and RESOURCES")
SET(THIRDPARTIES_PATH    ${CMAKE_CURRENT_SOURCE_DIR}/../thirdparties      CACHE PATH "This
directory contains thirdparties")

configure_file ("${CMAKE_CURRENT_SOURCE_DIR}/common/config.h.in"
            "${CMAKE_CURRENT_SOURCE_DIR}/include/config.h" )
```

Если у нас есть config.h.in:

```
cmakedefine PATH_DATA "@myproject_DATA@"
cmakedefine THIRDPARTIES_PATH "@THIRDPARTIES_PATH@"
```

Предыдущие CMakeLists будут генерировать заголовок с ++ следующим образом:

```
#define PATH_DATA "/home/user/projects/myproject/data"
#define THIRDPARTIES_PATH "/home/user/projects/myproject/thirdparties"
```

Экзамен на основе контрольной версии SDL2

Если у вас есть модуль `cmake` . Вы можете создать папку с именем `in` хранить все конфигурационные файлы.

Например, у вас есть проект под названием `FOO` , вы можете создать файл `FOO_config.h.in` например:

```
//============================================================================
//  CMake configuration file, based on SDL 2 version header
// ===========================================================================
```

```
#pragma once

#include <string>
#include <sstream>

namespace yournamespace
{
  /**
 *  \brief Information the version of FOO_PROJECT in use.
 *
 *  Represents the library's version as three levels: major revision
 *  (increments with massive changes, additions, and enhancements),
 *  minor revision (increments with backwards-compatible changes to the
 *  major revision), and patchlevel (increments with fixes to the minor
 *  revision).
 *
 *  \sa FOO_VERSION
 *  \sa FOO_GetVersion
 */
typedef struct FOO_version
{
    int major;        /**< major version */
    int minor;        /**< minor version */
    int patch;        /**< update version */
} FOO_version;

/* Printable format: "%d.%d.%d", MAJOR, MINOR, PATCHLEVEL
*/
#define FOO_MAJOR_VERSION   0
#define FOO_MINOR_VERSION   1
#define FOO_PATCHLEVEL      0

/**
 *  \brief Macro to determine FOO version program was compiled against.
 *
 *  This macro fills in a FOO_version structure with the version of the
 *  library you compiled against. This is determined by what header the
 *  compiler uses. Note that if you dynamically linked the library, you might
 *  have a slightly newer or older version at runtime. That version can be
 *  determined with GUCpp_GetVersion(), which, unlike GUCpp_VERSION(),
 *  is not a macro.
 *
 *  \param x A pointer to a FOO_version struct to initialize.
 *
 *  \sa FOO_version
 *  \sa FOO_GetVersion
 */
#define FOO_VERSION(x)                          \
{                                       \
    (x)->major = FOO_MAJOR_VERSION;             \
    (x)->minor = FOO_MINOR_VERSION;             \
    (x)->patch = FOO_PATCHLEVEL;                \
}

/**
 *  This macro turns the version numbers into a numeric value:
 *  \verbatim
    (1,2,3) -> (1203)
    \endverbatim
 *
```

```
 *  This assumes that there will never be more than 100 patchlevels.
 */
#define FOO_VERSIONNUM(X, Y, Z)                      \
    ((X)*1000 + (Y)*100 + (Z))

/**
 *  This is the version number macro for the current GUCpp version.
 */
#define FOO_COMPILEDVERSION \
    FOO_VERSIONNUM(FOO_MAJOR_VERSION, FOO_MINOR_VERSION, FOO_PATCHLEVEL)

/**
 *  This macro will evaluate to true if compiled with FOO at least X.Y.Z.
 */
#define FOO_VERSION_ATLEAST(X, Y, Z) \
    (FOO_COMPILEDVERSION >= FOO_VERSIONNUM(X, Y, Z))

}

// Paths
#cmakedefine FOO_PATH_MAIN "@FOO_PATH_MAIN@"
```

Этот файл создаст `FOO_config.h` в пути установки с переменной, определенной в с `FOO_PATH_MAIN` из переменной cmake. Чтобы сгенерировать его, вам нужно включить `in` файл в ваш CMakeLists.txt, как это (установить пути и переменные):

```
MESSAGE("Configuring FOO_config.h ...")
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/common/in/FOO_config.h.in"
"${FOO_PATH_INSTALL}/common/include/FOO_config.h" )
```

Этот файл будет содержать данные из шаблона и переменную с вашим реальным путем, например:

```
// Paths
#define FOO_PATH_MAIN "/home/YOUR_USER/Respositories/git/foo_project"
```

Прочитайте Настроить файл онлайн: https://riptutorial.com/ru/cmake/topic/8304/настроить-файл

глава 9: Переменные и свойства

Вступление

Простота основных переменных CMake противоречит сложности полного синтаксиса переменных. На этой странице представлены различные случаи с примерами и указаны недостатки, которых следует избегать.

Синтаксис

- set (значение переменной_имя [описание типа CACHE [FORCE]])

замечания

Имена переменных зависят от регистра. Их значения имеют тип string. Значение переменной ссылается через:

```
${variable_name}
```

и оценивается внутри цитируемого аргумента

```
"${variable_name}/directory"
```

Examples

Кэшированная (глобальная) переменная

```
set(my_global_string "a string value"
    CACHE STRING "a description about the string variable")
set(my_global_bool TRUE
    CACHE BOOL "a description on the boolean cache entry")
```

Если кэшированная переменная уже определена в кеше, когда CMake обрабатывает соответствующую строку (например, когда перезагружается CMake), она не изменяется. Чтобы перезаписать по умолчанию, добавьте `FORCE` в качестве последнего аргумента:

```
set(my_global_overwritten_string "foo"
    CACHE STRING "this is overwritten each time CMake is run" FORCE)
```

Локальная переменная

```
set(my_variable "the value is a string")
```

По умолчанию локальная переменная определяется только в текущем каталоге и любых подкаталогах, добавленных через команду `add_subdirectory` .

Чтобы расширить область действия переменной, существует две возможности:

1. `CACHE` , что сделает его доступным по всему миру
2. используйте `PARENT_SCOPE` , который сделает его доступным в родительской области. Родительская область - это либо файл `CMakeLists.txt` в родительском каталоге, либо вызывающая функция текущей функции.

   Технически родительский каталог будет файлом `CMakeLists.txt` который включает текущий файл с помощью команды `add_subdirectory` .

Строки и списки

Важно знать, как CMake различает списки и простые строки. Когда вы пишете:

```
set(VAR "ab c")
```

вы создаете **строку** со значением `"ab c"` . Но когда вы пишете эту строку без кавычек:

```
set(VAR abc)
```

Вместо этого вы создаете **список** из трех элементов: `"a"` , `"b"` и `"c"` .

Переменные не списка также являются списками (одного элемента).

Списки могут управляться командой `list()` , которая позволяет объединять списки, искать их, получать доступ к произвольным элементам и т. Д. (Документация по списку ()).

Немного запутанный, **список** также является **строкой** . Линия

```
set(VAR abc)
```

эквивалентно

```
set(VAR "a;b;c")
```

Поэтому, чтобы объединить списки, вы также можете использовать команду `set()` :

```
set(NEW_LIST "${OLD_LIST1};${OLD_LIST2})"
```

Переменные и глобальный кеш переменных

В основном вы будете использовать «обычные переменные» :

```
set(VAR TRUE)
set(VAR "main.cpp")
set(VAR1 ${VAR2})
```

Но CMake также знает глобальные «кэшированные переменные» (сохраняется в `CMakeCache.txt`). И если в текущей области существуют обычные и кешированные переменные с тем же именем, обычные переменные скрывают кэшированные:

```
cmake_minimum_required(VERSION 2.4)
project(VariablesTest)

set(VAR "CACHED-init" CACHE STRING "A test")
message("VAR = ${VAR}")

set(VAR "NORMAL")
message("VAR = ${VAR}")

set(VAR "CACHED" CACHE STRING "A test" FORCE)
message("VAR = ${VAR}")
```

Выход первого запуска

```
VAR = CACHED-init
VAR = NORMAL
VAR = CACHED
```

Выход второго запуска

```
VAR = CACHED
VAR = NORMAL
VAR = CACHED
```

Примечание. Параметр `FORCE` также отменяет / удаляет нормальную переменную из текущей области.

Использовать случаи для кэшированных переменных

Обычно используются два случая использования (пожалуйста, не используйте их для глобальных переменных):

1. Значение вашего кода должно быть модифицировано у пользователя вашего проекта, например, с помощью `cmakegui` , `ccmake` или с помощью `cmake -D ...` :

   CMakeLists.txt / MyToolchain.cmake

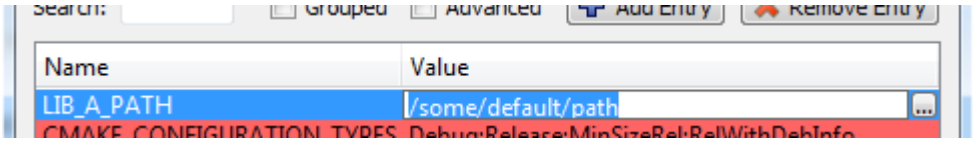
   ```
   set(LIB_A_PATH "/some/default/path" CACHE PATH "Path to lib A")
   ```

   Командная строка

   ```
   $ cmake -D LIB_A_PATH:PATH="/some/other/path" ..
   ```

Это предопределяет это значение в кеше, и указанная выше строка не будет изменять его.

CMake GUI



В графическом интерфейсе пользователь сначала запускает процесс настройки, затем может модифицировать любое кешированное значение и заканчивается запуском создания среды сборки.

2. Кроме того, CMake выполняет кэширование результатов поиска / тестирования / компилятора (поэтому ему не нужно делать это снова при повторном запуске шагов конфигурации / генерации)

```
find_path(LIB_A_PATH libA.a PATHS "/some/default/path")
```

Здесь `LIB_A_PATH` создается как кэшированная переменная.

Добавление профилей для CMake для использования gprof

Предполагается, что ряд событий будет работать следующим образом:

1. Компилировать код с опцией -pg
2. Код ссылки с опцией -pg
3. Запустить программу
4. Программа генерирует файл gmon.out
5. Запустить программу gprof

Чтобы добавить флаги профилирования, вы должны добавить в свой CMakeLists.txt:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")
SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pg")
SET(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -pg")
```

Это должно добавить флаги для компиляции и ссылки и использовать после выполнения программы:

gprof ./my_exe

Если вы получите сообщение об ошибке:

```
gmon.out: No such file or directory
```

Это означает, что компиляция не добавила информацию профилирования должным

образом.

Прочитайте Переменные и свойства онлайн: https://riptutorial.com/ru/cmake/topic/2091/переменные-и-свойства

глава 10: Поиск и использование установленных пакетов, библиотек и программ

Синтаксис

- find_package (pkgname [версия] [EXACT] [QUIET] [ТРЕБУЕТСЯ])
- включают в себя (FindPkgConfig)
- pkg_search_module (префикс [ТРЕБУЕТСЯ] [QUIET] pkgname [otherpkg ...])
- pkg_check_modules (префикс [ТРЕБУЕТСЯ] [QUIET] pkgname [otherpkg ...])

параметры

параметр	подробности
версия (необязательно)	Минимальная версия пакета, определяемая основным номером и необязательно небольшим, патчем и номером настройки, в формате major.minor.patch.tweak
EXACT (необязательно)	Укажите, что версия, указанная в `version` является точной версией, которая будет найдена
ТРЕБУЕТСЯ (необязательно)	Автоматически выдает ошибку и останавливает процесс, если пакет не найден
QUIET (необязательно)	Функция не будет отправлять какое-либо сообщение на стандартный вывод

замечания

- Способ `find_package` совместим со всей платформой, тогда как путь `pkg-config` доступен только на Unix-подобных платформах, таких как Linux и OSX.
- Полное описание многочисленных параметров и опций `find_package` можно найти в руководстве .
- Несмотря на то, что можно указать многие дополнительные параметры, такие как версия пакета, не все модули Find правильно используют все эти параметры. Если произойдет какое-либо неопределенное поведение, может потребоваться найти

модуль в пути установки CMake и исправить или понять его поведение.

Examples

Используйте find_package и найдите .cmake модули

По умолчанию для поиска установленных пакетов с CMake используется функция `find_package` в сочетании с файлом `Find<package>.cmake` . Цель файла - определить правила поиска для пакета и установить разные переменные, такие как `<package>_FOUND` , `<package>_INCLUDE_DIRS` и `<package>_LIBRARIES` .

Многие файлы `Find<package>.cmake` уже определены по умолчанию в CMake. Однако, если нет `${CMAKE_SOURCE_DIR}/cmake/modules` файла для пакета, вы всегда можете написать свой собственный и поместить его в `${CMAKE_SOURCE_DIR}/cmake/modules` (или любой другой каталог, если `CMAKE_MODULE_PATH` был переопределен)

Список модулей по умолчанию можно найти в руководстве (v3.6) . Необходимо проверить руководство в соответствии с версией CMake, используемой в проекте, или же там могут отсутствовать модули. Также можно найти установленные модули с помощью `cmake --help-module-list` .

Хороший пример для `FindSDL2.cmake` на Github

Вот базовый `CMakeLists.txt` , который потребует SDL2:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH} ${CMAKE_SOURCE_DIR}/cmake/modules")
find_package(SDL2 REQUIRED)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Используйте pkg_search_module и pkg_check_modules

В Unix-подобных операционных системах можно использовать программу `pkg-config` для поиска и настройки пакетов, которые предоставляют файл `<package>.pc` .

Чтобы использовать `pkg-config` , необходимо `include(FindPkgConfig)` в `CMakeLists.txt` . Тогда есть две возможные функции:

- `pkg_search_module` , который проверяет пакет и использует первый доступный.
- `pkg_check_modules` , которые проверяют все соответствующие пакеты.

Вот базовый `CMakeLists.txt` который использует `pkg-config` для поиска SDL2 с версией выше

или равной 2.0.1:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

include(FindPkgConfig)
pkg_search_module(SDL2 REQUIRED sdl2>=2.0.1)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Прочитайте Поиск и использование установленных пакетов, библиотек и программ онлайн: https://riptutorial.com/ru/cmake/topic/6752/поиск-и-использование-установленных-пакетов--библиотек-и-программ

глава 11: Пользовательские сборки

Вступление

Пользовательские шаги сборки полезны для запуска пользовательских целей в вашей сборке проектов или для простого копирования файлов, поэтому вам не нужно делать это вручную (возможно, dll?). Здесь я покажу вам два примера: первый - для копирования DLL (в частности, библиотек Qt5) в двоичный каталог проектов (Debug или Release), а второй - для запуска пользовательской цели (Doxygen в этом случае) в вашем решении (если вы используете Visual Studio).

замечания

Как вы можете видеть, вы можете много сделать с пользовательскими целями сборки и шагами в cmake, но вы должны быть осторожны при их использовании, особенно при копировании DLL. Хотя это удобно, оно иногда может привести к тому, что ласково называется «dll hell».

В основном это означает, что вы можете потеряться, в каких DLL-файлах зависит ваш исполняемый файл, какие из них загружаются и какие из них ему нужно запустить (возможно, из-за переменной пути вашего компьютера).

Помимо вышеперечисленного, не стесняйтесь делать индивидуальные цели, делая все, что хотите! Они мощные и гибкие и являются бесценным инструментом для любого проекта cmake.

Examples

Пример копирования dll Qt5

Предположим, у вас есть проект, который зависит от Qt5, и вам нужно скопировать соответствующие DLL в каталог сборки, и вы не хотите делать это вручную; вы можете сделать следующее:

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

# add the executable
add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
```

```
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )
```

Итак, каждый раз, когда вы строите свой проект, если целевые DLL-файлы были изменены, которые вы хотите скопировать, они будут скопированы после того, как ваша цель (в этом случае основной исполняемый файл) будет построена (обратите внимание на команду `copy_if_different`); в противном случае они не будут скопированы.

Кроме того, обратите внимание на использование выражений генератора . Преимущество использования этих компонентов заключается в том, что вам не нужно явно указывать, где копировать DLL или какие варианты использовать. Чтобы иметь возможность использовать их, проект, который вы используете (Qt5 в этом случае), должен иметь импортированные цели.

Если вы создаете отладочную версию, то CMake знает (на основе импортированной цели) скопировать Qt5Cored.dll, Qt5Guid.dll и Qt5Widgetsd.dll в папку Debug вашей папки. Если вы создаете в выпуске, то версии выпуска .dll будут скопированы в папку выпуска.

Запуск пользовательской цели

Вы также можете создать настраиваемую цель для запуска, когда хотите выполнить определенную задачу. Обычно это исполняемые файлы, которые вы запускаете для выполнения разных действий. Что-то, что может быть особенно полезно, - запустить Doxygen для создания документации для вашего проекта. Для этого вы можете сделать следующее в вашем `CMakeLists.txt` (для простоты мы рассмотрим пример проекта Qt5):

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )

#Add target to build documents from visual studio.
set(DOXYGEN_INPUT ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile)
#set the output directory of the documentation
set(DOXYGEN_OUTPUT_DIR ${CMAKE_CURRENT_SOURCE_DIR}/docs)
# sanity check...
message("Doxygen Output ${DOXYGEN_OUTPUT_DIR}")
find_package(Doxygen)
```

```
if(DOXYGEN_FOUND)
    # create the output directory where the documentation will live
    file(MAKE_DIRECTORY ${DOXYGEN_OUTPUT_DIR})
    # configure our Doxygen configuration file. This will be the input to the doxygen
    # executable
    configure_file(${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in
    ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile @ONLY)

# now add the custom target. This will create a build target called 'DOCUMENTATION'
# in your project
ADD_CUSTOM_TARGET(DOCUMENTATION
  COMMAND ${CMAKE_COMMAND} -E echo_append "Building API Documentation..."
  COMMAND ${CMAKE_COMMAND} -E make_directory ${DOXYGEN_OUTPUT_DIR}
  COMMAND ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND ${CMAKE_COMMAND} -E echo "Done."
  WORKING_DIRECTORY ${DOXYGEN_OUTPUT_DIR})

endif(DOXYGEN_FOUND)
```

Теперь, когда мы создаем наше решение (опять же, предполагая, что вы используете Visual Studio), у вас будет цель сборки, называемая `DOCUMENTATION` которую вы можете создать для восстановления документации вашего проекта.

Прочитайте Пользовательские сборки онлайн: https://riptutorial.com/ru/cmake/topic/9537/пользовательские-сборки

глава 12: Построение целей

Синтаксис

- add_executable (target_name [EXCLUDE_FROM_ALL] source1 [источник2 ...])
- add_library (lib_name [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 [источник2 ...])

Examples

исполняемые

Чтобы создать цель сборки, производящую исполняемый файл, следует использовать команду `add_executable` :

```
add_executable(my_exe
               main.cpp
               utilities.cpp)
```

Это создает цель сборки, например `make my_exe` для GNU make, с соответствующими `my_exe` сконфигурированного компилятора для создания исполняемого файла `my_exe` из двух исходных файлов `main.cpp` и `utilities.cpp` .

По умолчанию, все исполняемые цели добавляются к встроено `all` целям (`all` для GNU сделать, `BUILD_ALL` для MSVC).
Чтобы исключить возможность создания исполняемого файла с помощью `all` цели по умолчанию, можно добавить необязательный параметр `EXCLUDE_FROM_ALL` сразу после целевого имени:

```
add_executable(my_optional_exe EXCLUDE_FROM_ALL main.cpp)
```

Библиотеки

Чтобы создать цель сборки, которая создает библиотеку, используйте команду `add_library` :

```
add_library(my_lib lib.cpp)
```

Переменная CMake `BUILD_SHARED_LIBS` управляет всякий раз, когда создается статическая (`OFF`) или общая (`ON`) библиотека, используя, например, `cmake .. -DBUILD_SHARED_LIBS=ON` .
Однако вы можете явно настроить создание общей или статической библиотеки, добавив `STATIC` или `SHARED` после целевого имени:

```
add_library(my_shared_lib SHARED lib.cpp) # Builds an shared library
```

```
add_library(my_static_lib STATIC lib.cpp) # Builds an static library
```

Фактический выходной файл отличается между системами. Например, разделяемая библиотека в системах Unix обычно называется `libmy_shared_library.so` , но в Windows это будет `my_shared_library.dll` и `my_shared_library.lib` .

Как `add_executable` , добавить `EXCLUDE_FROM_ALL` перед списком исходных файлов , чтобы исключить его из `all` цели:

```
add_library(my_lib EXCLUDE_FROM_ALL lib.cpp)
```

Библиотеки, предназначенные для загрузки во время выполнения (например, плагины или приложения, использующие что-то вроде `dlopen`), должны использовать `MODULE` вместо `SHARED` / `STATIC` :

```
add_library(my_module_lib MODULE lib.cpp)
```

Например, в Windows не будет файла импорта (`.lib`), потому что символы экспортируются непосредственно в `.dll` .

Прочитайте Построение целей онлайн: https://riptutorial.com/ru/cmake/topic/3107/построение-целей

глава 13: Создание тестовых наборов с CTest

Examples

Базовый комплект тестов

```
# the usual boilerplate setup
cmake_minimum_required(2.8)
project(my_test_project
        LANGUAGES CXX)

# tell CMake to use CTest extension
enable_testing()

# create an executable, which instantiates a runner from
# GoogleTest, Boost.Test, QtTest or whatever framework you use
add_executable(my_test
               test_main.cpp)

# depending on the framework, you need to link to it
target_link_libraries(my_test
                      gtest_main)

# now register the executable with CTest
add_test(NAME my_test COMMAND my_test)
```

Макрос `enable_testing()` делает много магии. Прежде всего, он создает встроенный целевой `test` (для GNU make, `RUN_TESTS` для VS), который при запуске выполняет *CTest* .

Вызов `add_test()` наконец, регистрирует произвольный исполняемый файл с помощью *CTest* , поэтому исполняемый файл запускается всякий раз, когда мы вызываем `test` объект.

Теперь создайте проект как обычно и, наконец, запустите тестовый объект

GNU Make	Visual Studio
`make test`	`cmake --build . --target RUN_TESTS`

Прочитайте Создание тестовых наборов с CTest онлайн:
https://riptutorial.com/ru/cmake/topic/4197/создание-тестовых-наборов-с-ctest

глава 14: Тест и отладка

Examples

Общий подход к отладке при создании с помощью Make

Предположим, что `make` терпит неудачу:

```
$ make
```

Запустите его вместо `make VERBOSE=1` чтобы увидеть выполненные команды. Затем сразу запустите команду компоновщика или компилятора, которую вы увидите. Попробуйте сделать это, добавив необходимые флаги или библиотеки.

Затем выясните, что нужно изменить, поэтому сам CMake может передать правильные аргументы команде компилятора / компоновщика:

- что менять в системе (какие библиотеки устанавливать, какие версии, версии самого CMake)
- если предыдущий сбой, какие переменные среды для установки или параметры передаются в CMake
- в противном случае, что нужно изменить в `CMakeLists.txt` проекта или скриптах обнаружения библиотеки, таких как `FindSomeLib.cmake`

Чтобы помочь в этом, добавьте `message(${MY_VARIABLE})` в `CMakeLists.txt` или `*.cmake` для отладки переменных, которые вы хотите проверить.

Пусть CMake создает подробные Makefiles

Когда проект CMake инициализируется через `project()` , выходная многословность результирующего скрипта сборки может быть скорректирована с помощью:

```
CMAKE_VERBOSE_MAKEFILE
```

Эта переменная может быть задана с помощью командной строки CMake при настройке проекта:

```
cmake -DCMAKE_VERBOSE_MAKEFILE=ON <PATH_TO_PROJECT_ROOT>
```

Для GNU эта переменная имеет тот же эффект, что и работа `make VERBOSE=1` .

Ошибки отладки find_package ()

Примечание. Представленные сообщения об ошибках CMake уже включают исправление для «нестандартных» путей установки библиотеки / инструмента. Следующие примеры просто демонстрируют более подробные результаты CMake `find_package()` .

CMake внутренне поддерживаемый пакет / модуль

Если следующий код (замените модуль `FindBoost` на ваш модуль)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Boost REQUIRED)
```

дает некоторую ошибку, например

```
CMake Error at [...]/Modules/FindBoost.cmake:1753 (message):
  Unable to find the requested Boost libraries.

  Unable to find the Boost header files.  Please set BOOST_ROOT to the root
  directory containing Boost or BOOST_INCLUDEDIR to the directory containing
  Boost's headers.
```

И вам интересно, где он пытался найти библиотеку, вы можете проверить, есть ли в вашем пакете опция `_DEBUG` например, модуль `Boost` для получения более подробного вывода

```
$ cmake -D Boost_DEBUG=ON ..
```

CMake enabled Package / Library

Если следующий код (замените `Xyz` на соответствующую библиотеку)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Xyz REQUIRED)
```

дает некоторую ошибку, например

```
CMake Error at CMakeLists.txt:4 (find_package):
  By not providing "FindXyz.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Xyz", but
  CMake did not find one.

  Could not find a package configuration file provided by "Xyz" with any of
  the following names:
```

```
    XyzConfig.cmake
    xyz-config.cmake

  Add the installation prefix of "Xyz" to CMAKE_PREFIX_PATH or set "Xyz_DIR"
  to a directory containing one of the above files.  If "Xyz" provides a
  separate development package or SDK, be sure it has been installed.
```

И вам интересно, где он пытался найти библиотеку, вы можете использовать недокументированную глобальную переменную `CMAKE_FIND_DEBUG_MODE` для получения более подробного вывода

```
$ cmake -D CMAKE_FIND_DEBUG_MODE=ON ..
```

Прочитайте Тест и отладка онлайн: https://riptutorial.com/ru/cmake/topic/4098/тест-и-отладка

глава 15: Упаковка и распределение проектов

Синтаксис

- # Пакет каталога сборки
  pack [PATH]
- # Использовать определенный генератор
  cpack -G [GENERATOR] [PATH]
- # Предоставить дополнительные переопределения
- cpack -G [GENERATOR] -C [КОНФИГУРАЦИЯ] -P [ИМЯ ПАКЕТЫ] -R [ПАКЕТНАЯ ВЕРСИЯ] -B [КАТАЛОГ ПАКЕТОВ] - поставщик [ПАКЕТНЫЙ ВЫСТАВК]

замечания

CPack - это внешний инструмент, обеспечивающий быструю упаковку встроенных проектов CMake, путем сбора всех необходимых данных прямо из файлов `CMakeLists.txt` и используемых команд установки, таких как `install_targets()` .

Чтобы CPack правильно работал, `CMakeLists.txt` должен включать файлы или целевые объекты, которые должны быть установлены с использованием цели `install` .

Минимальный скрипт может выглядеть так:

```
# Required headers
cmake(3.0)

# Basic project setup
project(my-tool)

# Define a buildable target
add_executable(tool main.cpp)

# Provide installation instructions
install_targets(tool DESTINATION bin)
```

Examples

Создание пакета для проекта CMake

Чтобы создать распространяемый пакет (например, ZIP-архив или программу установки), обычно достаточно просто вызвать CPack, используя синтаксис, очень похожий на вызов CMake:

```
cpack path/to/build/directory
```

В зависимости от среды это соберет все необходимые / установленные файлы для проекта и поместит их в сжатый архив или самораспаковывающийся установщик.

Выбор используемого генератора CPack

Чтобы создать пакет с использованием определенного формата, можно выбрать **генератор,** который будет использоваться.

Подобно CMake, это может быть сделано с использованием аргумента **-G** :

```
cpack -G 7Z .
```

Используя эту командную строку, пакет будет построен в текущем каталоге с использованием формата архива 7-Zip.

На момент написания статьи CPack версии 3.5 поддерживал следующие генераторы по умолчанию:

- `7Z` Формат файла 7-Zip (архив)
- `IFW` Qt Installer Framework (исполняемый файл)
- `NSIS` Null Soft Installer (исполняемый файл)
- `NSIS64` Null Soft Installer (64-разрядный, исполняемый)
- `STGZ` Самораспаковывающееся сжатие tar GZip (архив)
- `TBZ2` Tap BZip2 сжатие (архив)
- `TGZ` Tar GZip компрессия (архив)
- `TXZ` Tar XZ компрессия (архив)
- `TZ` Tar Сжатие сжатия (архив)
- Формат файла `WIX` MSI через инструменты WiX (исполняемый архив)
- `ZIP` формат ZIP-архива (архив)

Если явный генератор не предоставляется, CPack будет пытаться определить наилучший доступный в зависимости от реальной среды. Например, он предпочтет создать самораспаковывающийся исполняемый файл в Windows и создать только ZIP-архив, если подходящий набор инструментов не найден.

Прочитайте Упаковка и распределение проектов онлайн:
https://riptutorial.com/ru/cmake/topic/4368/упаковка-и-распределение-проектов

глава 16: Функции и макросы

замечания

Основное различие между *макросами* и *функциями* заключается в том, что *макросы* оцениваются в текущем контексте, а *функции* открывают новую область в текущем контексте. Таким образом, переменные, определенные внутри *функций* , не известны после того, как функция была оценена. Напротив, переменные внутри *макросов* все еще определены после оценки макроса.

Examples

Простой макрос для определения переменной на основе ввода

```
macro(set_my_variable _INPUT)
  if("${_INPUT}" STREQUAL "Foo")
    set(my_output_variable "foo")
  else()
    set(my_output_variable "bar")
  endif()
endmacro(set_my_variable)
```

Используйте макрос:

```
set_my_variable("Foo")
message(STATUS ${my_output_variable})
```

распечатает

```
-- foo
```

в то время как

```
set_my_variable("something else")
message(STATUS ${my_output_variable})
```

распечатает

```
-- bar
```

Макрос для заполнения переменной данного имени

```
macro(set_custom_variable _OUT_VAR)
  set(${_OUT_VAR} "Foo")
endmacro(set_custom_variable)
```

Используйте его с

```
set_custom_variable(my_foo)
message(STATUS ${my_foo})
```

который будет печатать

```
-- Foo
```

Прочитайте Функции и макросы онлайн: https://riptutorial.com/ru/cmake/topic/2096/функции-и-макросы

кредиты

S. No	Главы	Contributors
1	Начало работы с cmake	Amani, arrowd, ComicSansMS, Community, Daniel Schepler, dontloo, Fantastic Mr Fox, fedepad, Florian, greatwolf, Mario, Neui, OliPro007, Torbjörn, Ziv
2	Добавить каталоги в компилятор Включить путь	kiki, Torbjörn
3	Иерархический проект	Adam Trhon, Anedar, Clare Macrae, Robert
4	Интеграция CMake в инструментах CI GitHub	ComicSansMS
5	Использование CMake для настройки тегов препроцессора	JVApen, Matthew
6	Компилировать функции и стандартный выбор C / C ++	wasthishelpful
7	Конфигурации сборки	Jav_Rock
8	Настроить файл	Jav_Rock, Shihe Zhang, vgonisanz
9	Переменные и свойства	arrowd, CivFan, Florian, Torbjörn, Trilarion, Trygve Laugstøl, vgonisanz
10	Поиск и использование установленных пакетов, библиотек и программ	OliPro007
11	Пользовательские	Developer Paul

	сборки	
12	Построение целей	arrowd, Neui, Torbjörn
13	Создание тестовых наборов с CTest	arrowd, ComicSansMS, Torbjörn
14	Тест и отладка	Florian, Torbjörn, Velkan
15	Упаковка и распределение проектов	Mario, Meysam, Neui
16	Функции и макросы	Torbjörn