FREE eBook

LEARNING cmake

Free unaffiliated eBook created from
Stack Overflow contributors.

#cmake

Table of Contents

About

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

Chapter 1: Getting started with cmake

Remarks

CMake is a tool for defining and managing code builds, primarily for C++.

CMake is a cross-platform tool; the idea is to have a single definition of how the project is built - which translates into specific build definitions for any supported platform.

It accomplishes this by pairing with different platform-specific buildsystems; CMake is an intermediate step, that generates build input for different specific platforms. On Linux, CMake generates Makefiles; on Windows, it can generate Visual Studio projects, and so on.

Build behavior is defined in `CMakeLists.txt` files - one in every directory of the source code. Each directory's `CMakeLists` file defines what the buildsystem should do in that specific directory. It also defines which subdirectories CMake should handle as well.

Typical actions include:

- Build a library or an executable out of some of the source files in this directory.
- Add a filepath to the include-path used during build.
- Define variables that the buildsystem will use in this directory, and in its subdirectories.
- Generate a file, based on the specific build configuration.
- Locate a library which is somewhere in the source tree.

The final `CMakeLists` files can be very clear and straightforward, because each is so limited in scope. Each only handles as much of the build as is present in the current directory.

For official resources on CMake, see CMake's Documentation and Tutorial.

Versions

Version	Release Date
3.9	2017-07-18
3.8	2017-04-10
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

Version	Release Date
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

CMake Installation

Head over to CMake download page and get a binary for your operating system, e.g. Windows, Linux, or Mac OS X. On Windows double click the binary to install. On Linux run the binary from a terminal.

On Linux, you can also install the packages from the distribution's package manager. On Ubuntu 16.04 you can install the command-line and graphical application with:

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

On FreeBSD you can install the command-line and the Qt-based graphical application with:

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

On Mac OSX, if you use one of the package managers available to install your software, the most notable being MacPorts (MacPorts) and Homebrew (Homebrew), you could also install CMake via one of them. For example, in case of MacPorts, typing the following

```
sudo port install cmake
```

will install CMake, while in case you use the Homebrew package manger you will type

```
brew install cmake
```

Once you have installed CMake you can check easily by doing the following

```
cmake --version
```

You should see something similar to the following

```
cmake version 3.5.1

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

Switching between build types, e.g. debug and release

CMake knows several build types, which usually influence default compiler and linker parameters (such as debugging information being created) or alternative code paths.

By default, CMake is able to handle the following build types:

- **Debug**: Usually a classic debug build including debugging information, no optimization etc.
- **Release**: Your typical release build with no debugging information and full optimization.
- **RelWithDebInfo:**: Same as *Release*, but with debugging information.
- **MinSizeRel**: A special *Release* build optimized for size.

How configurations are handled depends on the generator that is being used.

Some generators (like Visual Studio) support multiple configurations. CMake will generate all configurations at once and you can select from the IDE or using `--config CONFIG` (with `cmake --build`) which configuration you want to build. For these generators CMake will try its best to

generate a build directory structure such that files from different configurations do not step on each other.

Generators that do only support a single configuration (like Unix Makefiles) work differently. Here the currently active configuration is determined by the value of the CMake variable `CMAKE_BUILD_TYPE`.

For example, to pick a different build type one could issue the following command line commands:

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

A CMake script should avoid setting the `CMAKE_BUILD_TYPE` itself, as it's generally considered the users responsibility to do so.

For single-config generators switching the configuration requires re-running CMake. A subsequent build is likely to overwrite object files produced by the earlier configuration.

Simple "Hello World" Project

Given a C++ source file `main.cpp` defining a `main()` function, an accompanying `CMakeLists.txt` file (with the following content) will instruct *CMake* to generate the appropriate build instructions for the current system and default C++ compiler.

main.cpp *(C++ Hello World Example)*

```
#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)
```

See it live on Coliru

1. `cmake_minimum_required(VERSION 2.4)` sets a minimum CMake version required to evaluate the current script.

2. `project(hello_world)` starts a new CMake project. This will trigger a lot of internal CMake logic, especially the detection of the default C and C++ compiler.

3. With `add_executable(app main.cpp)` a build target `app` is created, which will invoke the

configured compiler with some default flags for the current setting to compile an executable `app` from the given source file `main.cpp`.

Command Line *(In-Source-Build, not recommended)*

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

`cmake .` does the compiler detection, evaluates the `CMakeLists.txt` in the given `.` directory and generates the build environment in the current working directory.

The `cmake --build .` command is an abstraction for the necessary build/make call.

Command Line *(Out-of-Source, recommended)*

To keep your source code clean from any build artifacts you should do "out-of-source" builds.

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

Or CMake can also abstract your platforms shell's basic commands from above's example:

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

"Hello World" with multiple source files

First we can specify the directories of header files by `include_directories()`, then we need to specify the corresponding source files of the target executable by `add_executable()`, and be sure there's exactly one `main()` function in the source files.

Following is a simple example, all the files are assumed placed in the directory `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
```

We can follow the same procedure in the above example to build our project. Then executing `app` will print

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

"Hello World" as a library

This example shows how to deploy the "Hello World" program as a library and how to link it with other targets.

Say we have the same set of source/header files as in the http://www.riptutorial.com/cmake/example/22391/-hello-world--with-multiple-source-files example. Instead of building from multiple source files, we can first deploy `foo.cpp` as a library by using `add_library()` and afterwards linking it with the main program with `target_link_libraries()`.

We modify **CMakeLists.txt** to

```
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)
```

and following the same steps, we'll get the same result.

Read Getting started with cmake online: https://riptutorial.com/cmake/topic/862/getting-started-with-cmake

Chapter 2: Add Directories to Compiler Include Path

Syntax

- include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])

Parameters

Parameter	Description
`dirN`	one ore more relative or absolute paths
`AFTER`, `BEFORE`	(optional) whether to add the given directories to the front or end of the current list of include paths; default behaviour is defined by `CMAKE_INCLUDE_DIRECTORIES_BEFORE`
`SYSTEM`	(optional) tells the compiler to tread the given directories as *system include dirs*, which might trigger special handling by the compiler

Examples

Add a Project's Subdirectory

Given the following project structure

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

the following line in the `CMakeLists.txt` file

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

adds the `include` directory to the *include search path* of the compiler for all targets defined in this directory (and all its subdirectories included via `add_subdirectory()`).

Thus, the file `myHeader.h` in the project's `include` subdirectory can be included via `#include "myHeader.h"` in the `main.cpp` file.

Read Add Directories to Compiler Include Path online:
https://riptutorial.com/cmake/topic/5968/add-directories-to-compiler-include-path

Chapter 3: Build Configurations

Introduction

This topic shows the uses of different CMake configurations like Debug or Release, in different environments.

Examples

Setting a Release/Debug configuration

```
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}/"
    )
```

Performin the following builds will generate two different ('/opt/myproject/lib/Debug' '/opt/myproject/lib/Release') folders with the libraries:

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

Read Build Configurations online: https://riptutorial.com/cmake/topic/8319/build-configurations

Chapter 4: Build Targets

Syntax

- add_executable(target_name [EXCLUDE_FROM_ALL] source1 [source2...])
- add_library(lib_name [STATIC|SHARED|MODULE] [EXCLUDE_FROM_ALL] source1 [source2 ...])

Examples

Executables

To create a build target producing an executable, one should use the `add_executable` command:

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

This creates a build target, e.g. `make my_exe` for GNU make, with the appropriate invocations of the configured compiler to produce an executable `my_exe` from the two source files `main.cpp` and `utilities.cpp`.

By default, all executable targets are added to the builtin `all` target (`all` for GNU make, `BUILD_ALL` for MSVC).
To exclude an executable from being built with the default `all` target, one can add the optional parameter `EXCLUDE_FROM_ALL` right after the target name:

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

Libraries

To create an build target that creates an library, use the `add_library` command:

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

The CMake variable `BUILD_SHARED_LIBS` controls whenever to build an static (`OFF`) or an shared (`ON`) library, using for example `cmake .. -DBUILD_SHARED_LIBS=ON`. However, you can explicitly set to build an shared or an static library by adding `STATIC` or `SHARED` after the target name:

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

The actual output file differs between systems. For example, an shared library on Unix systems is usually called `libmy_shared_library.so`, but on Windows it would be `my_shared_library.dll` and `my_shared_library.lib`.

Like `add_executable`, add `EXCLUDE_FROM_ALL` before the list of source files to exclude it from the `all` target:

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

Libraries, that are designed to be loaded at runtime (for example plugins or applications using something like `dlopen`), should use `MODULE` instead of `SHARED`/`STATIC`:

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

For example, on Windows, there won't be a import (`.lib`) file, because the symbols are directly exported in the `.dll`.

Read Build Targets online: https://riptutorial.com/cmake/topic/3107/build-targets

Chapter 5: CMake integration in GitHub CI tools

Examples

Configure Travis CI with stock CMake

Travis CI has CMake 2.8.7 pre-installed.

A minimal `.travis.yml` script for an out-of source build

```
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
```

Configure Travis CI with newest CMake

The CMake version preinstalled on Travis is very old. You can use the official Linux binaries to build with a newer version.

Here is an example `.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
```

Read CMake integration in GitHub CI tools online: https://riptutorial.com/cmake/topic/1445/cmake-integration-in-github-ci-tools

Chapter 6: Compile features and C/C++ standard selection

Syntax

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

Examples

Compile Feature Requirements

Required compiler features can be specified on a target using the command target_compile_features:

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

The features must be part of CMAKE_C_COMPILE_FEATURES or CMAKE_CXX_COMPILE_FEATURES; cmake reports an error otherwise. Cmake will add any necessary flags such as `-std=gnu++11` to the compile options of the target.

In the example, the features are declared `PRIVATE`: the requirements will be added to the target, but not to its consumers. To automatically add the requirements to a target building against foo, `PUBLIC` or `INTERFACE` should be used instead of `PRIVATE`:

```
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
)
```

C/C++ version selection

Wanted version for C and C++ can be specified globally using respectively variables `CMAKE_C_STANDARD` (accepted values are 98, 99 and 11) and `CMAKE_CXX_STANDARD` (accepted values are 98, 11 and 14):

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

These will add the needed compile options on targets (e.g. `-std=c++11` for gcc).

The version can be made a requirement by setting to `ON` the variables `CMAKE_C_STANDARD_REQUIRED` and `CMAKE_CXX_STANDARD_REQUIRED` respectively.

The variables must be set before target creation. The version can also be specified per-target:

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

Read Compile features and C/C++ standard selection online:
https://riptutorial.com/cmake/topic/5297/compile-features-and-c-cplusplus-standard-selection

Chapter 7: Configure file

Introduction

configure_file is a CMake function for copying a file to another location and modify its contents. This function is very useful for generating configuration files with paths, custom variables, using a generic template.

Remarks

Copy a file to another location and modify its contents.

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

Copies a file to file and substitutes variable values referenced in the file content. If is a relative path it is evaluated with respect to the current source directory. The must be a file, not a directory. If is a relative path it is evaluated with respect to the current binary directory. If names an existing directory the input file is placed in that directory with its original name.

If the file is modified the build system will re-run CMake to re-configure the file and generate the build system again.

This command replaces any variables in the input file referenced as ${VAR} or @VAR@ with their values as determined by CMake. If a variable is not defined, it will be replaced with nothing. If COPYONLY is specified, then no variable expansion will take place. If ESCAPE_QUOTES is specified then any substituted quotes will be C-style escaped. The file will be configured with the current values of CMake variables. If @ONLY is specified, only variables of the form @VAR@ will be replaced and ${VAR} will be ignored. This is useful for configuring scripts that use ${VAR}.

Input file lines of the form “#cmakedefine VAR ...” will be replaced with either “#define VAR ...” or /* #undef VAR */ depending on whether VAR is set in CMake to any value not considered a false constant by the if() command. (Content of ”...”, if any, is processed as above.) Input file lines of the form “#cmakedefine01 VAR” will be replaced with either “#define VAR 1” or “#define VAR 0” similarly.

With NEWLINE_STYLE the line ending could be adjusted:

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

COPYONLY must not be used with NEWLINE_STYLE.

Examples

Generate a c++ configure file with CMake

If we have a c++ project that uses a config.h configuration file with some custom paths or variables, we can generate it using CMake and a generic file config.h.in.

The config.h.in can be part of a git repository, while the generated file config.h will never be added, as it is generated from the current environment.

```
#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" )
```

If we have a config.h.in like this:

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

The previous CMakeLists will generate a c++ header like this:

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

Examble based on SDL2 control version

If you have a `cmake` module . You can create a folder called `in` to store all config files.

For example,you have a project called `FOO`, you can create a `FOO_config.h.in` file like:

```
//=============================================================================
//  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@"
```

This file will create a `FOO_config.h` in the install path, with a variable defined in c `FOO_PATH_MAIN` from cmake variable. To generate it you need to include `in` file in your CMakeLists.txt,like this (set paths and variables):

```
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" )
```

That file will contain the data from template, and variable with your real path, for example:

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

Read Configure file online: https://riptutorial.com/cmake/topic/8304/configure-file

Chapter 8: Create test suites with CTest

Examples

Basic Test Suite

```
# 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)
```

The macro `enable_testing()` does a lot of magic. First and foremost, it creates a builtin target `test` (for GNU make; `RUN_TESTS` for VS), which, when run, executes *CTest*.

The call to `add_test()` finally registers an arbitrary executable with *CTest*, thus the executable gets run whenever we call the `test` target.

Now, build the project as usual and finally run the test target

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

Read Create test suites with CTest online: https://riptutorial.com/cmake/topic/4197/create-test-suites-with-ctest

Chapter 9: Custom Build-Steps

Introduction

Custom build steps are useful to run custom targets in your project build or for easily copying files so you don't have to do it manually (maybe dlls?). Here I'll show you two examples, the first is for copying dlls (in particular Qt5 dlls) to your projects binary directory (either Debug or Release) and the second is for running a custom target (Doxygen in this case) in your solution (if you're using Visual Studio).

Remarks

As you can see, you can do a lot with custom build targets and steps in cmake, but you should be careful in using them, especially when copying dlls. While it is convinient to do so, it can at times result in what is affectionately called "dll hell".

Basically this means you can get lost in which dlls your executable actually depends on, which ones its loading, and which ones it needs to run (maybe because of your computer's path variable).

Other than the above caveat, feel free to make custom targets do whatever you want! They're powerful and flexible and are an invaluable tool to any cmake project.

Examples

Qt5 dll copy example

So let's say you have a project that depends on Qt5 and you need to copy the relevant dlls to your build directory and you don't want to do it manually; you can do the following:

```
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>
    )
```

So now everytime you build your project, if the target dlls have changed that you want to copy,

then they will be copied after your target (in this case the main executable) is built (notice the `copy_if_different` command); otherwise, they will not be copied.

Additionally, note the use of generator expressions here. The advantage with using these is that you don't have to explicitly say where to copy dlls or which variants to use. To be able to use these though, the project you're using (Qt5 in this case) must have imported targets.

If you're building in debug, then CMake knows (based on the imported target) to copy the Qt5Cored.dll, Qt5Guid.dll, and Qt5Widgetsd.dll to the Debug folder of you build folder. If you're building in release, then the release versions of the .dlls will be copied to the release folder.

Running a Custom Target

You can also create a custom target to run when you want to perform a particular task. These are typically executables that you run to do different things. Something that may be of particular use is to run Doxygen to generate documentation for your project. To do this you can do the following in your `CMakeLists.txt` (for the sake of simplicity we'll contiue our Qt5 project example):

```
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)
```

Now when we create our solution (again assuming you're using Visual Studio), you'll have a build target called `DOCUMENTATION` that you can build to regenerate your project's documentation.

Read Custom Build-Steps online: https://riptutorial.com/cmake/topic/9537/custom-build-steps

Chapter 10: Functions and Macros

Remarks

The main difference between *macros* and *functions* is, that *macros* are evaluated within the current context, while *functions* open a new scope within the current one. Thus, variables defined within *functions* are not known after the function has been evaluated. On the contrary, variables within *macros* are still defined after the macro has been evaluated.

Examples

Simple Macro to define a variable based on input

```
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)
```

Use the macro:

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

will print

```
-- foo
```

while

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

will print

```
-- bar
```

Macro to fill a variable of given name

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

Use it with

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

which will print

```
-- Foo
```

Read Functions and Macros online: https://riptutorial.com/cmake/topic/2096/functions-and-macros

Chapter 11: Hierarchical project

Examples

Simple approach without packages

Example that builds an executable (editor) and links a library (highlight) to it. Project structure is straightforward, it needs a master CMakeLists and a directory for each subproject:

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

The master CMakeLists.txt contains global definitions and `add_subdirectory` call for each subproject:

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

add_subdirectory(highlight)
add_subdirectory(editor)
```

CMakeLists.txt for the library assigns sources and include directories to it. By using `target_include_directories()` instead of `include_directories()` the include dirs will be propagated to library users:

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

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

CMakeLists.txt for the application assigns sources and links the highlight library. Paths to hightlighter's binary and includes are handled automaticaly by cmake:

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

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

Read Hierarchical project online: https://riptutorial.com/cmake/topic/1443/hierarchical-project

Chapter 12: Packaging and Distributing Projects

Syntax

- # Package a build directory
  pack [PATH]
- # Use a specific generator
  cpack -G [GENERATOR] [PATH]
- # Provide optional overrides
- cpack -G [GENERATOR] -C [CONFIGURATION] -P [PACKAGE NAME] -R [PACKAGE VERSION] -B [PACKAGE DIRECTORY] --vendor [PACKAGE VENDOR]

Remarks

CPack is an external tool allowing the fast packaging of built CMake projects by gathering all the required data straight from the `CMakeLists.txt` files and the utilized installation commands like `install_targets()`.

For CPack to properly work, the `CMakeLists.txt` must include files or targets to be installed using the `install` build target.

A minimal script could look like this:

```
# 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

Creating a package for a built CMake project

To create a redistributable package (e.g. a ZIP archive or setup program), it's usually enough to simply invoke CPack using a syntax very similar to calling CMake:

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

Depending on the environment this will gather all required/installed files for the project and put them into a compressed archive or self-extracting installer.

Selecting a CPack Generator to be used

To create a package using a specific format, it is possible to pick the **Generator** to be used.

Similar to CMake this may be done using the **-G** argument:

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

Using this command line would package the built project in the current directory using the 7-Zip archive format.

At the time of writing, CPack version 3.5 supports the following generators by default:

- `7Z` 7-Zip file format (archive)
- `IFW` Qt Installer Framework (executable)
- `NSIS` Null Soft Installer (executable)
- `NSIS64` Null Soft Installer (64-bit, executable)
- `STGZ` Self extracting Tar GZip compression (archive)
- `TBZ2` Tar BZip2 compression (archive)
- `TGZ` Tar GZip compression (archive)
- `TXZ` Tar XZ compression (archive)
- `TZ` Tar Compress compression (archive)
- `WIX` MSI file format via WiX tools (executable archive)
- `ZIP` ZIP file format (archive)

If no explicit generator is provided, CPack will try to determine the best available depending on the actual environment. For example, it will prefer creating a self-extracting executable on Windows and only create a ZIP archive if no suitable toolset is found.

Read Packaging and Distributing Projects online:
https://riptutorial.com/cmake/topic/4368/packaging-and-distributing-projects

Chapter 13: Search and use installed packages, libraries and programs

Syntax

- find_package(pkgname [version] [EXACT] [QUIET] [REQUIRED])
- include(FindPkgConfig)
- pkg_search_module(prefix [REQUIRED] [QUIET] pkgname [otherpkg...])
- pkg_check_modules(prefix [REQUIRED] [QUIET] pkgname [otherpkg...])

Parameters

Parameter	Details
version (optional)	Minimum version of the package defined by a major number and optionally a minor, patch and tweak number, in the format major.minor.patch.tweak
EXACT (optional)	Specify that the version specified in `version` is the exact version to be found
REQUIRED (optional)	Automatically throws an error and stop the process if the package is not found
QUIET (optional)	The function won't send any message to the standard output

Remarks

- The `find_package` way is compatible on all platform, whereas the `pkg-config` way is available only on Unix-like platforms, like Linux and OSX.
- A full description of the `find_package` numerous parameters and options can be found in the manual.
- Even though it is possible to specify many optional parameters such as the version of the package, not all Find modules properly uses all those parameters. If any undefined behaviour occur, it could be necessary to find the module in CMake's install path and fix or understand its behaviour.

Examples

Use find_package and Find.cmake modules

The default way to find installed packages with CMake is the use the `find_package` function in conjunction with a `Find<package>.cmake` file. The purpose of the file is to define the search rules for the package and set different variables, such as `<package>_FOUND`, `<package>_INCLUDE_DIRS` and `<package>_LIBRARIES`.

Many `Find<package>.cmake` file are already defined by default in CMake. However, if there is no file for the package you need, you can always write your own and put it inside `${CMAKE_SOURCE_DIR}/cmake/modules` (or any other directory if `CMAKE_MODULE_PATH` was overridden)

A list of default modules can be found in the manual (v3.6). It is essential to check the manual according to the version of CMake used in the project or else there could be missing modules. It is also possible to find the installed modules with `cmake --help-module-list`.

There is a nice example for a `FindSDL2.cmake` on Github

Here's a basic `CMakeLists.txt` that would require 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})
```

Use pkg_search_module and pkg_check_modules

On Unix-like operating systems, it is possible to use the `pkg-config` program to find and configure packages that provides a `<package>.pc` file.

In order to use `pkg-config`, it is necessary to call `include(FindPkgConfig)` in a `CMakeLists.txt`. Then, there are 2 possible functions:

- `pkg_search_module`, which checks for the package and uses the first available.
- `pkg_check_modules`, which check for all the corresponding packages.

Here's a basic `CMakeLists.txt` that uses `pkg-config` to find SDL2 with version above or equal to 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})
```

Read Search and use installed packages, libraries and programs online:

https://riptutorial.com/cmake/topic/6752/search-and-use-installed-packages--libraries-and-programs

Chapter 14: Test and Debug

Examples

General approach to debug when building with Make

Suppose the `make` fails:

```
$ make
```

Launch it instead with `make VERBOSE=1` to see the commands executed. Then directly run the linker or compiler command that you'll see. Try to make it work by adding necessary flags or libraries.

Then figure out what to change, so CMake itself can pass correct arguments to the compiler/linker command:

- what to change in the system (what libraries to install, which versions, versions of CMake itself)
- if previous fails, what environment variables to set or parameters to pass to CMake
- otherwise, what to change in the `CMakeLists.txt` of the project or the library detection scripts like `FindSomeLib.cmake`

To help in that, add `message(${MY_VARIABLE})` calls into `CMakeLists.txt` or `*.cmake` to debug variables that you want to inspect.

Let CMake create verbose Makefiles

Once a CMake project is initialized via `project()`, the output verbosity of the resulting build script can be adjusted via:

```
CMAKE_VERBOSE_MAKEFILE
```

This variable can be set via CMake's command line when configuring a project:

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

For GNU make this variable has the same effect as running `make VERBOSE=1`.

Debug find_package() errors

Note: The shown CMake error messages already include the fix for "non-standard" library/tool installation paths. The following examples just demonstrate more verbose CMake `find_package()` outputs.

CMake internally supported Package/Module

If the following code (replace the `FindBoost` module with your module in question)

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

find_package(Boost REQUIRED)
```

gives some error like

```
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.
```

And you're wondering where it tried to find the library, you can check if your package has an `_DEBUG` option like the `Boost` module has for getting more verbose output

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

CMake enabled Package/Library

If the following code (replace the `Xyz` with your library in question)

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

find_package(Xyz REQUIRED)
```

gives the some error like

```
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.
```

And you're wondering where it tried to find the library, you can use the undocumented

`CMAKE_FIND_DEBUG_MODE` global variable for getting a more verbose output

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

Read Test and Debug online: https://riptutorial.com/cmake/topic/4098/test-and-debug

Chapter 15: Using CMake to configure preproccessor tags

Introduction

The use of CMake in a C++ project if used correctly can allow the programmer to focus less on the platform, program version number and more on the actual program itself. With CMake you can define preprocessor tags that allow for easy checking of which platform or any other preprocessor tags you might need in the actual program. Such as the version number which could be leveraged in a log system.

Syntax

- #define preprocessor_name "@cmake_value@"

Remarks

It is important to understand not every preprocessor should be defined in the `config.h.in`. Preprocessor tags are generally used only to make the programmers life easier and should be used with discretion. You should research if a preprocessor tag already exists before defining it as you may run into undefined behavior on different system.

Examples

Using CMake to define the version number for C++ usage

The possibilities are endless. as you can use this concept to pull the version number from your build system; such as git and use that version number in your project.

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;
}
```

output

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

Read Using CMake to configure preproccessor tags online:
https://riptutorial.com/cmake/topic/10885/using-cmake-to-configure-preproccessor-tags

Chapter 16: Variables and Properties

Introduction

The simplicity of basic CMake variables belies the complexity of the full variable syntax. This page documents the various variable cases, with examples, and points out the pitfalls to avoid.

Syntax

- set(variable_name value [CACHE type description [FORCE]])

Remarks

Variable names are case-sensitive. Their values are of type string. The value of a variable is referenced via:

```
${variable_name}
```

and is evaluated inside a quoted argument

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

Examples

Cached (Global) Variable

```
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")
```

In case a cached variable is already defined in the cache when CMake processes the respective line (e.g. when CMake is rerun), it is not altered. To overwrite the default, append `FORCE` as the last argument:

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

Local Variable

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

By default, a local variable is only defined in the current directory and any subdirectories added

through the `add_subdirectory` command.

To extend the scope of a variable there are two possibilities:

1. `CACHE` it, which will make it globally available
2. use `PARENT_SCOPE`, which will make it available in the parent scope. The parent scope is either the `CMakeLists.txt` file in the parent directory or caller of the current function.

   Technically the parent directory will be the `CMakeLists.txt` file that included the current file via the `add_subdirectory` command.

Strings and Lists

It's important to know how CMake distinguishes between lists and plain strings. When you write:

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

you create a **string** with the value `"a b c"`. But when you write this line without quotes:

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

You create a **list** of three items instead: `"a"`, `"b"` and `"c"`.

Non-list variables are actually lists too (of a single element).

Lists can be operated on with the `list()` command, which allows concatenating lists, searching them, accessing arbitrary elements and so on (documentation of list()).

Somewhat confusing, a **list** is also a **string**. The line

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

is equivalent to

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

Therefore, to concatenate lists one can also use the `set()` command:

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

Variables and the Global Variables Cache

Mostly you will use "normal variables":

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

But CMake does also know global "cached variables" (persisted in `CMakeCache.txt`). And if normal and cached variables of the same name exist in the current scope, normal variables do hide the cached ones:

```
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}")
```

First Run's Output

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

Second Run's Output

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

Note: The `FORCE` option does also unset/remove the normal variable from the current scope.

Use Cases for Cached Variables

There are typically two use cases (please don't misuse them for global variables):

1. An value in your code should be modifiable from your project's user e.g. with the `cmakegui`, `ccmake` or with `cmake -D ...` option:

   ### CMakeLists.txt / MyToolchain.cmake

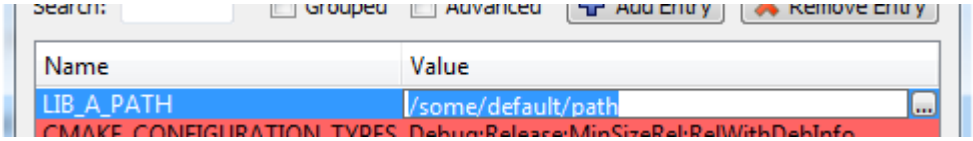
   ```
   set(LIB_A_PATH "/some/default/path" CACHE PATH "Path to lib A")
   ```

   ### Command Line

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

   This does pre-set this value in the cache and the above line will not modify it.

   ### CMake GUI

   In the GUI the user first starts the configuration process, then can modify any cached value

and finishes with starting the build environment generation.

2. Additionally CMake does cache search/test/compiler identification results (so it does not need to do it again whenever re-running the configuration/generation steps)

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

Here `LIB_A_PATH` is created as a cached variable.

Adding profiling flags to CMake to use gprof

The series of events here is supposed to work as follows:

1. Compile code with -pg option
2. Link code with -pg option
3. Run program
4. Program generates gmon.out file
5. Run gprof program

To add profiling flags, you must add to your 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")
```

That must add flags to compile and link, and use after execute the program:

gprof ./my_exe

If you get an error like:

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

That means that compilation didn't add profiling info properly.

Read Variables and Properties online: https://riptutorial.com/cmake/topic/2091/variables-and-properties

Credits

S. No	Chapters	Contributors
1	Getting started with cmake	Amani, arrowd, ComicSansMS, Community, Daniel Schepler, dontloo, Fantastic Mr Fox, fedepad, Florian, greatwolf, Mario, Neui, OliPro007, Torbjörn, Ziv
2	Add Directories to Compiler Include Path	kiki, Torbjörn
3	Build Configurations	Jav_Rock
4	Build Targets	arrowd, Neui, Torbjörn
5	CMake integration in GitHub CI tools	ComicSansMS
6	Compile features and C/C++ standard selection	wasthishelpful
7	Configure file	Jav_Rock, Shihe Zhang, vgonisanz
8	Create test suites with CTest	arrowd, ComicSansMS, Torbjörn
9	Custom Build-Steps	Developer Paul
10	Functions and Macros	Torbjörn
11	Hierarchical project	Adam Trhon, Anedar, Clare Macrae, Robert
12	Packaging and Distributing Projects	Mario, Meysam, Neui
13	Search and use installed packages, libraries and programs	OliPro007
14	Test and Debug	Florian, Torbjörn, Velkan
15	Using CMake to configure	JVApen, Matthew

	preproccessor tags	
16	Variables and Properties	arrowd, CivFan, Florian, Torbjörn, Trilarion, Trygve Laugstøl, vgonisanz