CMake Tutorial

Onur Dündar Follow
Feb 6, 2018 · 22 min read

Macbook Pro screen with a few dozen lines of code" by Luca Bravo on Unsplash

implement and design a solid build process. I hope this article helps other people who is learning about CMake.

. . .

Introduction

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called *CMakeLists.txt* files) are used to generate standard build files (e.g., Makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way.
CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executable binaries in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree.
CMake has supports for static and dynamic library builds. Another nice feature of CMake is that it can generates a cache file that is designed to be used with a graphical editor. For example, while CMake is running, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.
CMake scripts also make source management easier because it simplifies build script into one file and more organized, readable format.

Popular Open Source Project with CMake

Here is a list of popular open source projects using CMake for build purposes:

- OpenCV: https://github.com/opencv/opencv

https://en.wikipedia.org/wiki/CMake#Appli
It is always a good practice to read open sourc
best practices.

CMake as a Scripting Language

Photo by Markus Spiske on Unsplash

invoked with the script file to interpret and generate actual build file.
A developer can write either simple or complex building scripts using CMake language for the projects.
Build logic and definitions with CMake language is written either in `CMakeLists.txt` or a file ends with `<project_name>.cmake`. But as a best practice, main script is named as `CMakeLists.txt` instead of cmake.

- `CMakeLists.txt` file is placed at the source of the project you want to build.
- `CMakeLists.txt` is placed at the root of the source tree of any application, library it will work for.
- If there are multiple modules, and each module can be compiled and built separately, `CMakeLists.txt` can be inserted into the sub folder.
- `.cmake` files can be used as scripts, which runs `cmake` command to prepare environment pre-processing or split tasks which can be written outside of `CMakeLists.txt`.
- `.cmake` files can also define modules for projects. These projects can be separated build processes for libraries or extra methods for complex, multi-module projects.

Writing *Makefiles* might be harder than writing CMake scripts. CMake scripts by syntax and logic have similarity to high level languages so it makes easier for developers to create their *cmake* scripts with less effort and without getting lost in Makefiles.

CMake Commands

CMake commands are similar to C++/Java methods or functions, which take parameters as a list and perform certain tasks accordingly.
CMake commands are case insensitive.
There are built-in commands, which can be found from cmake documentation:

Photo by Hannah Joshua on Unsplash

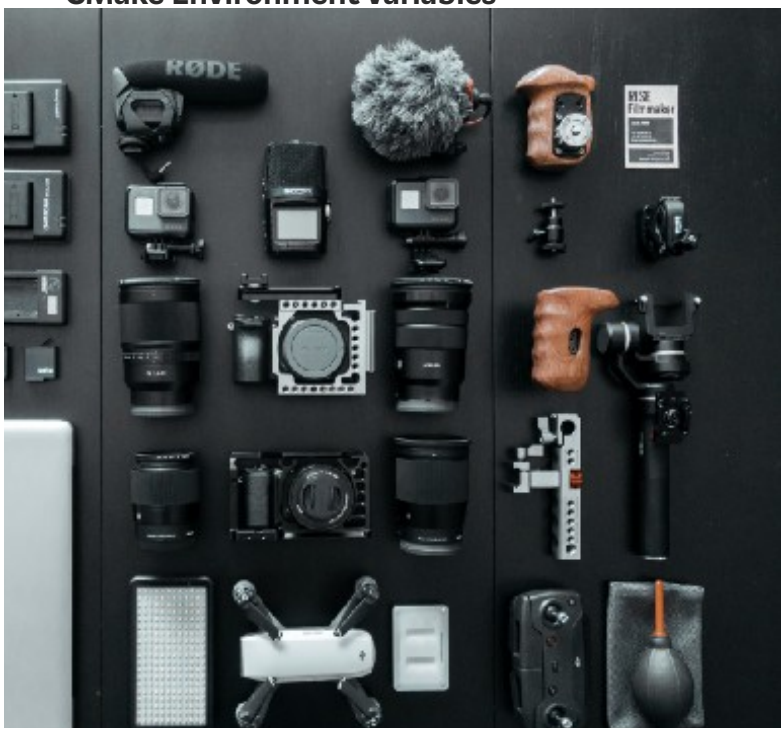
There are also commands to enable developers write out conditional statements, loops, iterate on list, assignments:

- `if, endif`
- `elif, endif`
- `while, endwhile`
- `foreach, endforeach`
- `list`
- `return`
- `set_property` (assign value to variable.)

Indentation is not mandatory but suggested while writing CMake scripts. CMake doesn't use ';' to understand end of statement.
All conditional statements should be ended with its corresponding end command (`endif`, `endwhile`, `endforeach` etc)
All these properties of CMake help developers to program complex build processes including multiple modules, libraries and platforms.
For example, KDE has its own CMake style guideline as in following URL:

- https://community.kde.org/Policies/CMake_Coding_Style

CMake Environment Variables



Environment variables are used to configure compiler flags, linker flags, test configurations for a regular build process. Compiler have to be guided to search for given directories for libraries.
A detailed list of environment variables can be seen from following URL:

- https://cmake.org/cmake/help/latest/manual/cmake-env-variables.7.html

Some of the environment variables are overriden by predefined CMake Variables. e.g. CXXFLAGS is overriden when CMAKE_CXX_FLAGS is defined.
Below is an example use case, when you want to enable all warnings during

CMake Variables

CMake includes predefined variables which are set by default as location of source tree and system components.
Variables are case-sensitive, not like commands. You can only use *alpha numeric chars* and *underscore*, *dash* (`_, -`) in definition of variable.
You can find more details about CMake variables in the following URLs

- https://cmake.org/cmake/help/v3.0/manual/cmake-language.7.html#variables
- https://cmake.org/cmake/help/v3.0/manual/cmake-variables.7.html#manual:cmake-variables(7)

Some of the variables can be seen as below, these are predefined according to root folder:

- `CMAKE_BINARY_DIR` : Full path to top level of build tree and binary output folder, by default it is defined as top level of build tree.
- `CMAKE_HOME_DIRECTORY` : Path to top of source tree
- `CMAKE_SOURCE_DIR` : Full path to top level of source tree.
- `CMAKE_INCLUDE_PATH` : Path used to find file, path

Variable values can be accessed with `${<variable_name>}` .

```
message("CXX Standard: ${CMAKE_CXX_STANDARD}")
set(CMAKE_CXX_STANDARD 14)
```

Just like above variables, you can define your own variables. You can call set command to set a value to a new variable or change value of existing variable like below:

```
set(TRIAL_VARIABLE "VALUE")
message("${TRIAL_VARIABLE}")
```

CMake Lists

All values in CMake are stored as string but a string can be treated as list in certain context.
A list of elements represented as a string by concatenating elements separated by semi-column ‘;’.

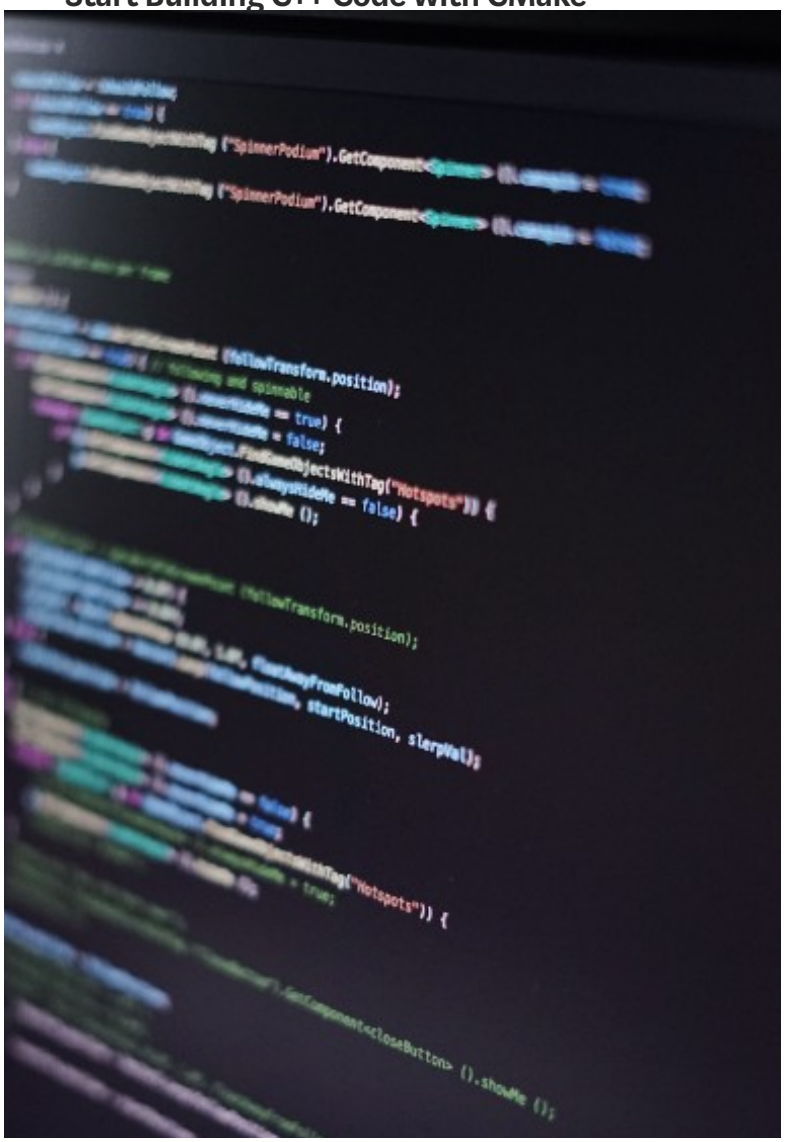
```
set(files a.txt b.txt
c.txt)
```

Photo by Kelly Sikkema on Unsplash

CMake Generator Expressions

Generator expressions are evaluated during build system generation to produce information specific to each build configuration.
Generator expressions are allowed in the context of many target properties, such as `LINK_LIBRARIES`, `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` and others. They may also be used when using commands to populate those properties, such as `target_link_libraries()`, `target include directories()`, `target compile definitions()` and others.
https://cmake.org/cmake/help/v3.3/manual/cmake-generator-expressions.7.html

Start Building C++ Code with CMake



In previous sections, we have covered the core principles of writing CMake scripts. Now, we can continue to write actual scripts to start building C++ code.
We can just start with a basic “Hello World!” example with CMake so we wrote the following “Hello CMake!” the main.cpp file as following:

```
#include <iostream>
int main() {
        std::cout<<"Hello
CMake!"<<std::endl;
}
```

Our purpose is to generate a binary to print “Hello CMake!”.
If there is no CMake we can run compiler to generate us a target basically with only following commands.

```
$ g++ main.cpp -o
cmake_hello
```

CMake helps to generate bash commands with the instructions you gave, for this

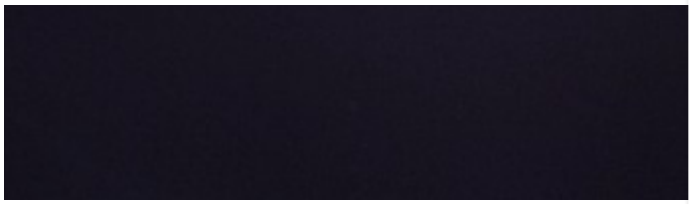


Photo by Oscar Nord on Unsplash

cmake version requirement as below:

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
add_executable(cmake_hello main.cpp)
```

When the script ready, you can run cmake command to generate Makefile/s for the project. You will notice that, cmake is identifying compiler versions and configurations with default information.

```
$ cmake CMakeLists.txt
-- The C compiler identification is AppleClang 9.0.0.9000039
-- The CXX compiler identification is AppleClang 9.0.0.9000039
-- Check for working C compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xc
toolchain/usr/bin/cc
-- Check for working C compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xc
toolchain/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xc
toolchain/usr/bin/c++
-- Check for working CXX compiler:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xc
toolchain/usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/Users/User/Projects/CMakeTutorial
```

When cmake finishes its job, Makefile will be generated together with CMakeCache.txt and some other artefects about build configuration. You can run `make` command to build project.

Let's make things little more complicated.
What if your code is depended on C++14 or l
that, return type deduction has been introduc
main.cpp with auto return type as below.

```
#include <iostream>
auto sum(int a, int b){
        return a + b;
}
int main() {
        std::cout<<"Hello CMake!"<<std::endl;
        std::cout<<"Sum of 3 + 4 :"<<sum(3, 4)<<std::endl;
        return 0;
}
```

If you try to build above code with default settings, you could get an error about auto return type because most hosts configured to work C++99 config by default. Therefore you should point your compiler to build with C++14 with setting CMAKE_CXX_STANDARD variable to 14. If you want to add C++14 on command line, you can set `-std=c++14`.
CMake generates Makefile with defaults settings which is mostly lower than C++14, so you should add 14 flag as seen below.

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
add_executable(cmake_hello main.cpp)
```

Now, you should be able to build it correctly. (cmake command would work even though you didn't add 14 standard but make command would return error.)
CMake eases building process for you; especially when you do cross compile to make sure you are working with correct version of compiler with correct configurations, this will enable multiple developers to use same building configurations across all machines including build server and developer PC.
What if, you want to build for multiple platforms:

- generate executables for Windows, Mac and Linux separetely.
- add different macros for Linux Kernel version later than X.

Following variables can be used to check for system related information: Pasted from:
https://cmake.org/Wiki/CMake_Checking_Platform

- `CMAKE_SYSTEM` the complete system name, e.g. "Linux-2.4.22", "FreeBSD-5.4-RELEASE" or "Windows 5.1"
- `CMAKE_SYSTEM_NAME` The name of the system targeted by the build. The three common values are Windows, Darwin, and Linux, though several others exist, such as Android, FreeBSD, and CrayLinuxEnvironment. Platforms without an operating system, such as embedded devices, are given Generic as a system name.

- `CMAKE_HOST_SYSTEM_NAME` The name of th

values as CMAKE_SYSTEM_NAME.

Let's check if build system is Unix or Windows

```
cmake_minimum_required(VERSION 3
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
# UNIX, WIN32, WINRT, CYGWIN, AP
flags set by default system
if(UNIX)
    message("This is a ${CMAKE_SYSTEM_NAME} system")
elseif(WIN32)
    message("This is a Windows System")
endif()
# or use MATCHES to see if actual system name
# Darwin is Apple's system name
if(${CMAKE_SYSTEM_NAME} MATCHES Darwin)
    message("This is a ${CMAKE_SYSTEM_NAME} system")
elseif(${CMAKE_SYSTEM_NAME} MATCHES Windows)
    message("This is a Windows System")
endif()
add_executable(cmake_hello main.cpp)
```

System information checks your build system other than building correct binary, like using macros.

You can define compiler macros to send to code during build process to change behavior.
Large code bases are implemented to be system agnostic with macros to use certain methods only for the correct system. That also prevents errors, next section shows how to define a macro and use in code.

Defining Macros using CMake

s of code on a computer screen" by Ilya Pavlov on Unsplash

Macros help engineers to build code conditionally to discard or include certain methods according to running system configurations.
You can define macros in CMake with `add_definitions` command, using **-D** flag before the macro name.
Lets define macro named `CMAKEMACROSAMPLE` and print it in the code.

```
cmake_minimum_required(VERS
ION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
# or use MATCHES to see if
actual system name
# Darwin is Apple's system
```

Below is the new main.cpp with printing macro.

```
#include <iostream>
#ifndef CMAKEMACROSAMPLE
    #define CMAKEMACROSAMPLE "NO SYSTEM NAME"
#endif
auto sum(int a, int b){
        return a + b;
}
int main() {
        std::cout<<"Hello CMake!"<<std::endl;
                std::cout<<CMAKEMACROSAMPLE<<std::endl;
        std::cout<<"Sum of 3 + 4 :"<<sum(3, 4)<<std::endl;
        return 0;
}
```

CMake Folder Organization

While building applications, we try to keep source tree clean and separate auto-generated files and binaries.
For CMake, many developers prefer to create a build folder under root tree and start CMake command inside, as below. Make sure you did cleaned all previously generate files (CMakeCache.txt), otherwise it doesn't creates files inside build.

```
$ mkdir build
$ cmake ..
$ ls -all
-rw-r--r--   1 onur  staff  13010 Jan 25 18:40 CMakeCache.txt
drwxr-xr-x  15 onur  staff    480 Jan 25 18:40 CMakeFiles
-rw-r--r--   1 onur  staff   4964 Jan 25 18:40 Makefile
-rw-r--r--   1 onur  staff   1256 Jan 25 18:40 cmake_install.cmake
$ make all
```

Above commands creates build files inside build directory. It is called *out-source* build.
You can add **build/** folder to your **.gitignore** to disable tracking.
At this time, you can not force CMake with variables to force creating artefacts into another folder with setting variables so if you don't want to create a directory and *cd* into it, you can also use below command to create folder and generate files inside it. **-H** and **-B** flags will help it.

```
# H indicates source directory
# B indicates build directory
# For CLion, you can navigate to
Execution and Deployment -> CMak
```

However, you can write script in CMake to ma

be.

Let's edit our `CMakeLists.txt` to generate bin

`CMAKE_RUNTIME_OUTPUT_DIRECTORY` or `EXECUT`

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
add_executable(cmake_hello main.cpp)
```

After building the project, `cmake_hello` binary will be generated inside build/bin folder.
Same procedure can be done for library paths as well for shared libraries *(.dll or .so)* with following variables.

- `LIBRARY_OUTPUT_PATH`
- `CMAKE_LIBRARY_OUTPUT_DIRECTORY`

Or you can use archive output paths for static libraries *(.a or .lib)*

- `CMAKE_ARCHIVE_OUTPUT_DIRECTORY`
- `ARCHIVE_OUTPUT_PATH`

In some cases, setting single path can be enough, but for larger projects with multiple modules, you may need to write cmake scripts to manage folders to easily organize build structure. This will help you to pack and deploy, install the generated executables, libraries, documents with certain folders.
It is a common practice to disable *in-source* building. Following script can be used to block in-source building.

- Source of script OpenCV Project:

  https://github.com/opencv/opencv/blob/master/CMakeLists.txt

```
# Disable in-source builds to prevent source tree corruption.
if(" ${CMAKE_SOURCE_DIR}" STREQUAL " ${CMAKE_BINARY_DIR}")
  message(FATAL_ERROR "
FATAL: In-source builds are not allowed.
       You should create a separate directory for build files.
")
endif()
```

Building a Library with CMake

Let's expand the CMakeHello project with additional sub folder and a class.
To keep things simple, we added a template class which is able to do 4 math operations, sum, subtraction, division, multiplication only on integers.

- First, we created lib/math folder inside source tree.
- Then, added class files, `operations.cpp, operations.hpp` as following

```
    public:
        int sum(const int &a, co
        int mult(const int &a, c
        int div(const int &a, co
        int sub(const int &a, co
    };
}
#endif //CMAKEHELLO_OPERATIONS_H
#include <exception>
#include <stdexcept>
#include <iostream>
#include "operations.hpp"
int math::operations::sum(const int &a, const int &b){
    return a + b;
}
int math::operations::mult(const int &a, const int &b){
    return a * b;
}
int math::operations::div(const int &a, const int &b){
    if(b == 0){
        throw std::overflow_error("Divide by zero exception");
    }
    return a/b;
}
int math::operations::sub(const int &a, const int &b){
    return a - b;
}
```

- Then, we modified `main.cpp` to include `operations.hpp` and use its `sum` method instead of previously written sum.

```
#include <iostream>
#include "lib/math/operations.hpp"
int main() {
        std::cout<<"Hello CMake!"<<std::endl;
        math::operations op;
        int sum = op.sum(3, 4);
        std::cout<<"Sum of 3 + 4 :"<<sum<<std::endl;
        return 0;
}
```

Quick Note: Shared vs Static Library

Before going more with library building with C++, lets get a quick brief about shared and static libraries.
Shared Library File Extensions:

- Windows: **.dll**
- Mac OS X: **.dylib**
- Linux: **.so**

Static Library File Extensions:

covers on shelves in a library" by Jamie Taylor on Unspl

shared libraries during executions. That requirement also decrease the performance because at each execution it tries to load instructions from shared objects.
Static libraries are used to fetch instructions directly into application binary by compiler, so all the code required from library are already injected into final application binary. That increase the size of object but increase the size of binary but performance get increased. Applications build with static library will also don't need dependencies on the running platform.

Building Library with Target

If you just want to build these files together with main.cpp, you can just add source files next to add_executable command. This will compile all together and create a single binary file. Leading to 15.076 bytes of exe file.

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
add_executable(cmake_hello main.cpp lib/math/operations.cpp
lib/math/operations.hpp)
```

As an alternate, you can create a variable named ${SOURCES} as a list to include target sources. It can be done in a lot of different ways, depending on your methodology.

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
set(SOURCES main.cpp
            lib/math/operations.cpp
            lib/math/operations.hpp)
add_executable(cmake_hello ${SOURCES})
```

Building Library Separate than Target

We can build library separately either as **shared** or **static**.
If we do that, we also need to *link library* to executable in order to enable executable file to make calls from operations library. We also want to generate library binaries inside **lib** directory of build folder.

```
cmake_minimum_required(VERSION 3
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CX
set(EXECUTABLE_OUTPUT_PATH ${CMA
set(LIBRARY_OUTPUT_PATH  ${CMAKE
message(${CMAKE_BINARY_DIR})
add_library(math SHARED lib/math
#add_library(math STATIC lib/mat
add_executable(cmake_hello main.
target_link_libraries(cmake_hello
```

You will see that, in mac OS, libmath.dylib is generated and binary size decreased to 14.876 bytes. *(No significant change because code is already very small)*.

Build Library as Sub-Module CMake

Another library building process is that, you can write a new `CMakeLists.txt` file inside `lib/operations` folder, to build it independently just before building exe file.
This kind of situations are mostly needed when there is an optional build required to generate module. In our case, above solution is better since they all dependend. However, if you want to add a case to only build libraries and skipping executable build process, below example can work as well.
Generate a new `CMakeLists.txt` inside `lib/math` folder as shown in below code snippet to build library.

```
cmake_minimum_required(VERSION 3.9.1)
set(LIBRARY_OUTPUT_PATH  ${CMAKE_BINARY_DIR}/lib)
add_library(math SHARED operations.cpp)
```

- You should delete `add_library` command and setting `LIBRARY_OUTPUT_PATH` from main `CMakeLists.txt`.
- Now, you should add new build path with `add_subdirectoy` . This command makes cmake to go for the folder and build it as well with the `CMakeLists.txt` inside it.

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
message(${CMAKE_BINARY_DIR})
add_subdirectory(lib/math)
add_executable(cmake_hello main.cpp)
target_link_libraries(cmake_hello math)
```

- Now, you will see the libmath.diylib inside build/lib folder again.

Above, examples shows how to deal with additional sources inside source tree. Either build them all together or separately according to your release plans.

Finding Existing Library with CMake

If a library installed to system with its `.cmake` system default library locations to find that lib Boost library can be installed via package mar cmake's `find_package` command to check if l Let's change our example little more. I want to random samples. Therefore, I included `boost/` `boost::random::normal_distribution` to ge

```
#include <iostream>
#include "lib/math/operations.hpp"
#include <boost/random.hpp>
int main() {
        std::cout<<"Hello CMake!"<<std::endl;
        math::operations op;
        int sum = op.sum(3, 4);
        std::cout<<"Sum of 3 + 4 :"<<sum<<std::endl;
    //Boost Random Sample
    boost::mt19937 rng;
    double mean = 2.3;
    double std = 0.34;
    auto normal_dist = boost::random::normal_distribution<double>
(mean, std);
    boost::variate_generator<boost::mt19937&,
            boost::normal_distribution<> > random_generator(rng,
normal_dist);
    for(int i = 0; i < 2; i++){
        auto rand_val = random_generator();
        std::cout<<"Random Val "<<i+1<<" :"<<rand_val<<std::endl;
    }
        return 0;
}
```

How `CMakeLists.txt` should be written now? It should check for the Boost library, if it can find includes headers and links libraries.
Note: In most cases, default library and include paths are defined in cmake default configurations, so it may find these libraries without any modification, but it is not suggested to leave CMakeLists.txt with trusting to system.

```
cmake_minimum_required(VERSION 3.9.1)
project(CMakeHello)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
#set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin)
message(${CMAKE_BINARY_DIR})
add_executable(cmake_hello main.cpp)
add_subdirectory(lib/math)
find_package(Boost 1.66)
```

```
        include_directories(${Boost_
        target_link_libraries(cmake_
    elseif(NOT Boost_FOUND)
        error("Boost Not Found")
    endif()
    target_link_libraries(cmake_hell
```

`include_directories` will include library hea
Above code will not continue to build if packa
add `REQUIRED` flag after package to raise error.
Note: Boost name already recognized by system because cmake has already defined commands to check for Boost library. If it is not a common library like boost, you should write your own scripts to enable this feature.
When a package found, following variables will be initialized automatically.
`<NAME>_FOUND` : Flag to show if it is found `<NAME>_INCLUDE_DIRS or <NAME>_INCLUDES` : Header directories `<NAME>_LIBRARIES or <NAME>_LIBRARIES or <NAME>_LIBS` : library files
`<NAME>_DEFINITIONS`
So what happens, if a library is in a custom folder and outside of source tree. If there won't be any CMake, your command line build would look like below.

```
g++ main.cpp -o cmake_hello -I/home/onur/libraries/boost/include -
L/home/onur/libraries/boost -lBoost
```

With this logic, you should add include and library folder with following commands.

```
include_directories(/Users/User/Projects/libraries/include)
link_directories(/Users/User/Projects/libraries/libs)
# elseif case can be
elseif(NOT Boost_FOUND)
message("Boost Not Found")
        include_directories(/Users/User/Projects/libraries/include)
        link_directories(/Users/User/Projects/libraries/libs)
        target_link_libraries(cmake_hello Boost)
endif()
```

Many different methodologies can be followed while including custom libraries for your project. You can also write custom cmake methods to search for given folders to check libraries etc. Most important thing is to understand linkage logic of compiler. You should point the headers and library (.so, .a) folder and link library for compile process.
Above case is not very safe, because you can't be sure about if it exists in the folders of host. Safest method is to fetch sources of library and build before going forward. In order to provide this property, dependent library should be added with its sources. If library is closed source, you should include binary to your source tree.

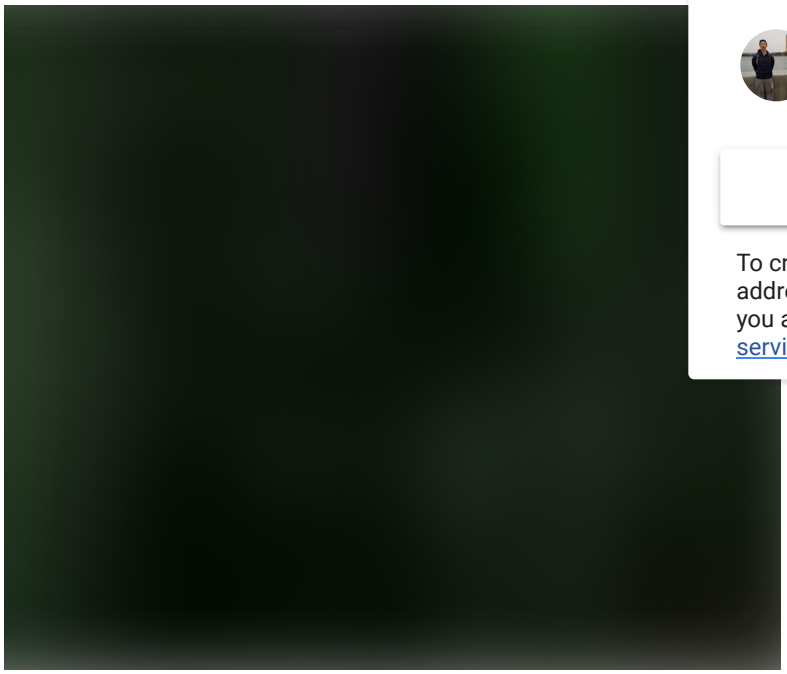


Photo by Markus Spiske on Unsplash

all possible checks into your cmake as well as your code as macros.

Change Compiler and Linker for Build

Let's say you will cross-compile your project for a different target system. In such cases, your host system should include the target system compiler and linker installed. A basic example is shown in official documentation of CMake:

https://cmake.org/cmake/help/v3.6/manual/cmake-toolchains.7.html#cross-compiling-for-linux

Official example is enough for this article's context, in order to cover it basically. It is an example of building for Raspberry Pi with its C/C++ compiler and tools (linker etc.)

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_SYSROOT /home/devel/rasp-pi-rootfs)
set(CMAKE_STAGING_PREFIX /home/devel/stage)
set(tools /home/devel/gcc-4.7-linaro-rpi-gnueabihf)
set(CMAKE_C_COMPILER ${tools}/bin/arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER ${tools}/bin/arm-linux-gnueabihf-g++)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

Most important part is to point compiler and tools (linker) paths correctly. Rest is configuration of build process, including optimisations.

It is also important to know your target system's compiler properties, it would always be different than the host system. For example, intel system has different level of instructions than ARM. Even arm cpu's would differ for instruction sets so for optimisations. Make sure you had covered those logic to apply for your cross-compile.

In the next section, we tried to cover some of the basic compiler and linker flags for GNU GCC and Clang.

Compiler/Linker Flags with CMake

flags used by Clang or GNU compilers. But, let

what they have been used for.
See, user manuals:

- https://gcc.gnu.org/onlinedocs/gcc-2.95
- https://clang.llvm.org/docs/ClangComm

Compiler Flags

Setting Compiler Flags

Compiler flags are crucial when it comes to co

optimisation values, behaviour of compiler etc

different ways but it is rather easier to define t

```
set(CMAKE_CXX_FLAGS "-std=c++0x -Wall")
# suggested way is to keep previous flags in mind and append new ones
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x -Wall")
# Alternatively, you can use generator expressions, which are
conditional expressions. Below says that, if compiper is c++ then set
it to c++11
add_compile_options("$<$<STREQUAL:$<TARGET_PROPERTY:LINKER_LANGUAGE>,
CXX>:-std=c++11>")
```

It is quite important to know, how these flags works and what flags do you want to use than setting flags with CMake.

- Optimisation flags set certain compiler flags or disables them. These flags are defined to make easier to switch on/off certain optimisation flags.
- Please see manual:
- https://clang.llvm.org/docs/ClangCommandLineReference.html#optimization-level
- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

-O0, -O1, -O2, -O3, -Os, -Oz, -Ofast

- Warning flags set warning properties during build process to see any warning during build to report them. Some may be turned off to fasten compile process because each warning process would require to make analysis. See manual and tree.
- https://gist.github.com/d0k/3608547
- https://clang.llvm.org/docs/DiagnosticsReference.html

-Wstring-conversion, -Wall, -Wswitch-enum …

Set Source File Properties

This is a complex property of CMake if there are multiple targets, it can be needed to change one target's certain behavior. In case, you would like to build main.cpp with C++11 and if you building only library, you may want to build it with C++14. In such cases, you may want to configure certain source's properties with using set_source_files_properties command like below:

```
set_source_files_properties(${CMAKE_CURRENT_SOURCE_DIR}/*.cpp
PROPERTIES COMPILE_FLAGS "-std=c++11")
```

A large set of properties can be seen from following manual.

- https://cmake.org/cmake/help/v3.3/manual/cmake-properties.7.html#source-file-properties

Each can be used for a specific case needed for your purpose.

Linker Flags

Here is a list of linker flags of GNU GCC Linker.

Most common flag is `-l` to link desired librar
Additional flags, help you to change behavior
Below are the variables which you can add lin

- CMAKE_EXE_LINKER_FLAGS: Flags used
- CMAKE_EXE_LINKER_FLAGS_RELEASE: executable
- CMAKE_EXE_LINKER_FLAGS_DEBUG: F executable
- CMAKE_STATIC_LINKER_FLAGS: Flags u (.a, .lib)
- CMAKE_SHARED_LINKER_FLAGS: Flags used by linker during the creation of static libraries (.a, .lib)
- CMAKE_MODULE_LINKER_FLAGS: Flags used by linker during the creation of static libraries (.a, .lib)

```
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -
fsanitize=address")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl")
```

Debug and Release Configuration

It is highly recommended to create CMakeLists.txt with multiple configurations according to your needs. If you intend to deliver binaries, you should make a `Release`, which doesn't include debug flags in binary, which would be a way faster and ready to run. However, debug version of executable files include many of other flags which exposes the memory, method names etc for debuggers to help identify the errors. It is not a good practice and not safe to deliver debug version of an application.
CMake helps you to write script to separate final builds of both type of outputs. There are additional build types like Release with Debug Flags (RELWITHDEBINFO) or Minium Release Size (MINSIZEREL). In this example, we will show both.

- You have to create Debug and Release folders for both type of builds under your build folder. build/Debug and build/Release. cmake command line will change as below:

```
$ cmake -H. -Bbuild/Debug
$ cmake -H. -Bbuild/Release
```

- Above configuration is not enough to create different binaries. You should also set build type with CMAKE_BUILD_TYPE variable on the command line. (CLion handles this process by itself.)

```
$ cmake -DCMAKE_BUILD_TYPE=Debug -H.  -Bbuild/Debug
$ cmake -DCMAKE_BUILD_TYPE=Release -H. -Bbuild/Release
```

- CMAKE_BUILD_TYPE is accessible inside CMakeLists.txt. You can easily check for build type in CMakeLists.txt

```
    message("Release Build")
endif()
```

- You can also, set compiler and linker flags
shown in the previous section.
- CMAKE_EXE_LINKER_FLAGS_RELEASE:
executable
- CMAKE_EXE_LINKER_FLAGS_DEBUG: F
executable
- CMAKE_CXX_FLAGS_RELEASE
- CMAKE_CXX_FLAGS_DEBUG

CMake Installation/Deployment Configurations

Photo by rawpixel on Unsplash

CMake helps create commands for installation process as well, not only the build process.
There is a good coverage of the parameters of install command of CMake in:

- https://cmake.org/cmake/help/v3.0/command/install.html

Main procedure in this process is to copy the files generated by build process to a destination folder on the host. Therefore:

- First, think about destination folder. `CMAKE_INSTALL_PREFIX` is the variable to define host destination. it is set to `/usr/local` by default. During the build process, you should point the destination folder in command-line.

```
$ cmake -
DCMAKE_BUILD_TYPE=Release -
DCMAKE_INSTALL_PREFIX=/usr/
local/test/with/cmake -H. -
Bbuild/Release
```

- Keeping in mind that, you are getting prefix destination from terminal you should think about library and executable destinations accordingly.

CMake Best Practices

- Always remember the previous configurations, make sure you append new flag instead of overwriting it. It is better to implement, add_flag/remove method of yours, to achieve easier

- Always check system information carefull
  to prevent faulty binaries.
- Always, check required libraries to contin

```
if(Boost_FOUND)
    message("Boost Found")
else()
    error("Boost Not Found")
endif()
```

Resources

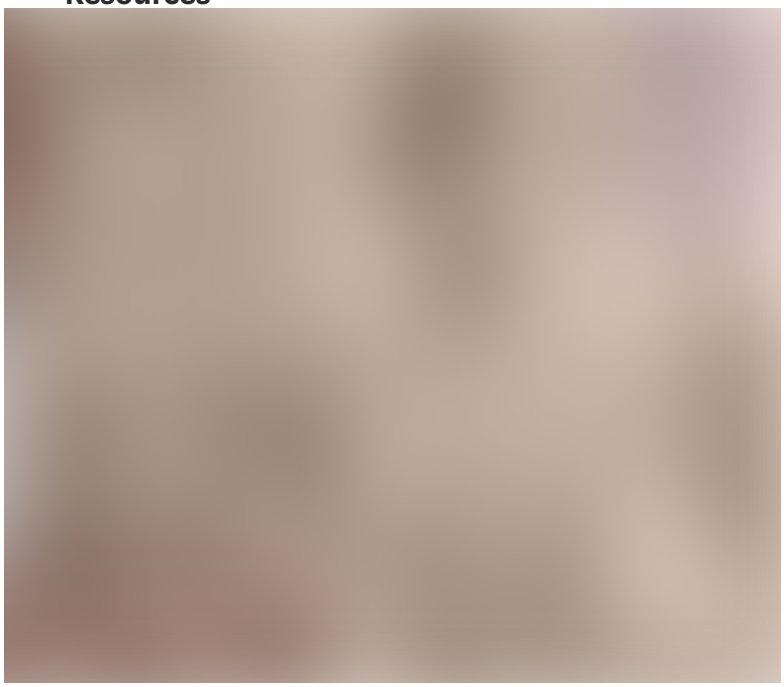


Photo by Patrick Tomasso on Unsplash

- https://cmake.org/cmake-tutorial/
- https://tuannguyen68.gitbooks.io/learning-cmake-a-beginner-s-guide/content/chap1/chap1.html
- https://cmake.org/cmake/help/latest
- https://cmake.org/cmake/help/v3.10/
- https://www.vtk.org/Wiki/CMake/Examples
- https://cmake.org/cmake/help/v3.0/module/FindBoost.html
- https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html
- https://clang.llvm.org/docs/ClangCommandLineReference.html
- http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/gcc-compiler-flags/

Programming Cmake Cpp C Programming Cross Platform

WRITTEN BY
Onur Dündar

Follow

More From Medium

More from Onur Dündar

MQTT — Part I: Understanding MQTT

Onur... Jul 21,... 68

Related reads

Understanding Compilers — For Humans

Luke ... Jun 1... 3.6K

Related reads

Splitting a string in C++

Ben Key Oct 24,... 57