MAKE - TUTORIAL

Make & Makefiles are used to automate the compiling of a small to medium project

This session is based on this makefile tutorial:

https://makefiletutorial.com/

Feel free to check it out

GETTING STARTED

```
$ unzip /CECI/proj/training/cmake.zip -d .
$ cd cmake-3.27.9/Step0
$ make
```

BASICS

A makefile is a bit like a "todo list".

```
target: prerequisite1 prerequisite2
	command1
	command2
	...
```

Careful with the syntax, makefile uses **tabs**, not spaces

Make is meant for smart compiling, so it will try to compile only what needs to be. How so ?

TERMINOLOGY

- **Rule :** a "task" of the makefile
- **Target :** the "name of a rule". It should produce a file with the same name (or be a special rule)
- **Prerequisite :** either the name of an existing file, or the target of a rule to be executed

TIMESTAMPS

Each file has a (set of) **timestamps** associated to it. Make uses the last time a file has been modified in order to determine if a rule has to be executed.

```
someExec: file1.c file2.h
    gcc file1.c -o someExec
```

This rule will be executed if :

- There is no file named 'someExec'
- 'file1.c' or 'file2.h' has a timestamp more recent than 'someExec'

GOOD PRACTICE

- You should have one rule per .c/.cc/.cpp/.cxx
- You should have the source file as a prerequisite
- The first rule is the default. Use it as your 'main'
- Write a rule 'clean' that removes all intermediary files (.o)

If no file bearing the name of the target is produced, the target will never be considered "up to date"

Sometimes, that's the point (*e.g.* the 'clean' target)

TO GO FURTHER

Make is very useful and easy to learn. If you are interested, have a look here: https://makefiletutorial.com/

There are several notions not covered here such as:

- Variables
- Rule templates
- Recursive makes
- Functions
- String formatting

CMAKE - TUTORIAL

CMake documentation:

https://cmake.org/documentation/

Tutorial based on official CMake tutorial:

https://cmake.org/cmake/help/v3.27/guide/tutorial/A%20Basic%20Starting%20Point.html

GETTING STARTED

```
$ ml releases/2023b
$ ml CMake GCC
$ cd cmake-3.27.9
```

STEP 1 - BASIC PROJECT

```
$ ls Step1/
CMakeLists.txt  TutorialConfig.h.in  tutorial.cxx
```

We can see three files:

CMakeLists.txt : This is the file read by CMake to build your project

TutorialConfig.h.in : This file is ingested by CMake to produce an actual header file (*i.e.* TutorialConfig.h)

tutorial.cxx : This is the source code of our project

CMAKE VERSION

Target: CMakeLists.txt (TODO 1)

```
cmake_minimum_required(VERSION <min>[...<policy_max>])
```

Do not set the minimal version of cmake to an arbitrarily low value.

Set it to the version included in the release you use (2023b)

```
cmake_minimum_required(VERSION 3.27)
```

NAME YOUR PROJECT

Target: CMakeLists.txt (TODO 2)

```
project(<PROJECT-NAME> [<language-name>...])
project(<PROJECT-NAME>
        [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
        [DESCRIPTION <project-description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

```
project(Tutorial)
```

ADD AN EXECUTABLE

Target: CMakeLists.txt (TODO 3)

```
add_executable(<name>
               [source1]
               [source2 ...])
```

```
add_executable(Tutorial tutorial.cxx)
```

BUILD AND RUN

Build your project

```
$ mkdir build1
$ cmake -S Step1 -B build1
# -S tells cmake where to find the sources
# -B tells cmake where to build the project
$ cmake --build build1
```

Run your program

```
$ ./build1/Tutorial 9
The square root of 9 is 3
$ ./build1/Tutorial 10
The square root of 10 is 3.16228
$ ./build1/Tutorial
Usage: ./build1/Tutorial number
```

STEP1 - SO FAR...

What did we learn ?

- Write a minimal working example of CMakeLists.txt
- Build a project by automatically generating a Makefile

What next ?

- Set the version of C++
- Automate variables/macros definition *in the code*

NEWER C++ CODE

Target: tutorial.cxx (TODO 4-5)

```
// convert input to double
// TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
const double inputValue = atof(argv[1]);
```

We want to use C++11 features

```
// convert input to double
// TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
const double inputValue = std::stod(argv[1]);
```

Don't forget TODO 5

Try and build your project, now

std::stod() does not exist in C++98

UPGRADE C++ VERSION

Target: CMakeLists.txt (TODO 6)

```
set(CMAKE_CXX_STANDARD 98)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

set() allows you to set the value of a variable

CMake variables are prefixed by CMAKE_

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

Try and build your project

IT'S ALL ABOUT VERSIONING

Target: CMakeLists.txt (TODO 7)

Add a version to your project.

```
project(Tutorial VERSION 1.0)
```

Now we can use the version as a macro in the rest of the project

FIRST CONFIGURED (HEADER) FILE

Target: TutorialConfig.h.in (TODO 8)

You can use variables defined by CMake at compile-time.

```
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

First, you need to write a *"template"* of a header file using the @ character to surround the variables you need

PRODUCE THE ACTUAL HEADER

Target: CMakeLists.txt (TODO 9)

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

Replaces the CMake variables in TutorialConfig.h.in with their value and produce a new file

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

Build your project again and inspect the file TutorialConfig.h

USE THE GENERATED MACROS

Target: tutorial.cxx

Include the header (TODO 10)

```
#include "TutorialConfig.h"
```

Print the version of your program (TODO 11)

```
    std::cout << argv[0] << " version: "
        << Tutorial_VERSION_MAJOR << "."
        << Tutorial_VERSION_MINOR << std::endl;
```

Now try and compile your project

ALLOW CMAKE TO *SEE* THE HEADER FILE

Target: CMakeLists.txt (TODO 12)

```
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

```
target_include_directories(Tutorial
        PUBLIC "${PROJECT_BINARY_DIR}")
```

STEP 2 - LIBRARIES AND SUBPROJECTS

Our project is now (a bit) larger and includes a library we are developing.

```
$ tree Step2
Step2
├── CMakeLists.txt
├── MathFunctions
│   ├── CMakeLists.txt
│   ├── MathFunctions.cxx
│   ├── MathFunctions.h
│   ├── mysqrt.cxx
│   └── mysqrt.h
├── TutorialConfig.h.in
└── tutorial.cxx
```

ADD A LIBRARY

Target: MathFunctions/CMakeLists.txt (TODO 1)

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [<source>...])
```

```
add_library(MathFunctions MathFunctions.cxx mysqrt.cxx)
```

You can think of add_library() as an extension of add_executable() already present in your project...

... But cmake needs to:

- Know where to find the library
- Know how to link the library

INCLUDE A LIBRARY

Target: CMakeLists.txt (TODO 2-3)

```
add_subdirectory(MathFunctions)
```

"Make cmake aware" of a directory containing sources

```
target_include_directories(Tutorial PUBLIC
        "${PROJECT_BINARY_DIR}"
        "${PROJECT_SOURCE_DIR}/MathFunctions/")
```

Add the include directory to the search paths of cmake (where are the headers of the library)

USE OUR LIBRARY IN OUR PROJECT

Target: tutorial.cxx (TODO 4-5)

```
#include "MathFunctions.h"
```

```
const double outputValue = mathfunctions::sqrt(inputValue);
```

Try and build your project

What causes "Undefined references" ?

Functions declared in a header are used, but the linker cannot find them

How to deal with them ?

Tell the linker it has to link the library

LINK THE LIBRARY

A compiler builds a project in two steps:

- Translating source code to binary
- Linking together the "binary parts" of the program

Target: CMakeLists.txt (TODO 6)

```
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

Tell cmake to link the library to our project

STEP2 - SO FAR ...

What did we learn ?

- A cleaner way to organise more complex projects
- How to develop a library for our project
- A better understanding of the compiling process
  - Translate
  - Link

What next ?

- Define compile-time options
- Conditionals in cmake

DEFINE AN OPTION

Target: MathFunctions/CMakeLists.txt (TODO 7)

```
option(<variable> "<help_text>" [value])
```

```
option(USE_MYMATH
       "Whether or not to use MathFunctions implementation"
       ON)
```

Pro-tip: Level up your game using ccmake (instead of cmake)

```
$ ccmake -S Step2/ -B build2/
$ cmake --build build2/
```

EXPORT CMAKE OPTIONS IN C/C++

Target: MathFunctions/CMakeLists.txt (TODO 8)

```
target_compile_definitions(<target>
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

```
if (USE_MYMATH)
    target_compile_definitions(MathFunctions
        PRIVATE "USE_MYMATH")
endif()
```

The macro USE_MYMATH is defined in the code only if it is ON

USE CMAKE OPTIONS IN C/C++

Target: MathFunctions/MathFunctions.cxx (TODO 9-11)

```
#include <cmath>
```

```
#ifdef USE_MYMATH
  #include "mysqrt.h"
#endif
```

```
#ifdef USE_MYMATH
  return detail::mysqrt(x);
#else
  return std::sqrt(x);
#endif
```

SKIP UNNECESSARY COMPILING

Make CMake compile MathFunctions/mysqrt.cxx only when we actually use it

Target: MathFunctions/CMakeLists.txt (TODO 12-14)

```
add_library(MathFunctions MathFunctions.cxx)
```

```
if (USE_MYMATH)
  target_compile_definitions(MathFunctions
    PRIVATE "USE_MYMATH")

  add_library(SqrtLibrary STATIC mysqrt.cxx)
  target_link_libraries(MathFunctions PUBLIC SqrtLibrary)
endif()
```

TO GO FURTHER...

Now you know how to add a library to your project...

but you need to specify some include/link paths in the "main" CMakeLists.txt. You know your own code, so you know what paths to add.

CMake allows to automatically add those paths when including a library in the main CMake.

If you are interested, Step3 covers this subject, head to the official tutorial to learn about it

https://cmake.org/cmake/help/v3.27/guide/tutorial

STEP2 - EXTENDED

What if we wanted to use a library installed on the cluster ?

Let's say **Eigen/3.4.0**

Target: MathFunctions/CMakeLists.txt (TODO 15-16)

```
find_package(Eigen3 3.4 REQUIRED)
```

```
else()
  target_link_libraries(MathFunctions PUBLIC Eigen3::Eigen)
```

BUILD YOUR PROJECT

As usual :

```
$ rm -rf build2/  # (optional) remove the build directory
$ ccmake -S Step2/ -B build2/
```

ccmake fails to configure our project !

```
$ ml Eigen  # load Eigen3 module
$ ccmake -S Step2/ -B build2/
$ cmake --build build2/
```

USE EIGEN3 IN THE C++ CODE

Target: MathFunctions/MathFunctions.cxx

Include Eigen and iostream (TODO 17)

Play around with Eigen and compute the square root (TODO 18)

```
Eigen::Array<double, 2, 2> arr;
arr << x, x+1, x+2, x+3;
std::cout << "Testing with Eigen. arr :" << std::endl << ar
std::cout << "Testing with Eigen. first row : " << arr(0, 
std::cout << "Testing with Eigen. first column :" << std::
std::cout << "Testing with Eigen. arr.sqrt() :" << std::en
return arr.sqrt()(0, 0);
```

STEP 5 - INSTALLING AND TESTING

So far, we were running the program by directly executing the binary produced in the build directory

What if we'd like to keep things a bit cleaner and install the program in another directory ?

What about a program that you can call from anywhere, like you would for any other program (python, ls, cmake, vim, nano, *etc.*)

HOW DOES LINUX RUN PROGRAMS ?

Unix-like systems need to "see" your program.

So you must move them in one of the directories linux "looks into", they are collectively called the **PATH**

```
$ echo $PATH
[... lots of paths separated by colons ...]
```

BTW, this is how lmod works. It changes the value of PATH to load or unload modules on the fly without the need to install them

WHERE TO INSTALL OUR PROGRAM ?

You can install it locally in your home *~/.local/*

```
$ tree ~/.local/
/home/ulb/operations/npotvin/.local/
├── bin
│   └── Tutorial
├── include
│   ├── MathFunctions.h
│   └── TutorialConfig.h
├── lib
│   ├── libMathFunctions.a
│   └── libSqrtLibrary.a
└── share
```

TELL CMAKE WHAT TO DO WITH THE LIBRARY

Target: MathFunctions/CMakeLists.txt (TODO 1-2)

```
set(installable_libs MathFunctions tutorial_compiler_flags)

if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
```

```
install(TARGETS ${installable_libs} DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

TELL CMAKE WHAT TO DO WITH THE MAIN PROGRAM

Target: CMakeLists.txt (TODO 3-4)

```
install(TARGETS Tutorial DESTINATION bin)

install(FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  DESTINATION include
  )
```

INSTALL YOUR PROGRAM

First, configure and build:

```
$ mkdir build5
$ ccmake -S Step5/ -B build5/
$ cmake --build build5/
```

Be careful to configure the right install path.

Your program has been built successfully, it is now time to install it.

```
$ cmake --install build5/
```

As simple as that

STEP5 - SO FAR ...

We have learned :

- How to run a program from anywhere
- How to install a program after build

To go further, you can have a look to step 9 the official tutorial to package an installer

---

What's next ?

- Add some tests to your code
- Automate testing

FIRST DUMMY TESTS

Target: CMakeLists.txt (TODO 5-8)

```
enable_testing()
```

```
add_test(NAME Runs COMMAND Tutorial 25)
```

Ok, but how do we check the output ?

```
add_test(NAME StandardUse COMMAND Tutorial 4)
set_tests_properties(StandardUse
  PROPERTIES PASS_REGULAR_EXPRESSION "4 is 2")
```

```
add_test(NAME Usage COMMAND Tutorial)
set_tests_properties(Usage
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage:.*number")
```

TEST YOUR PROGRAM

```
$ cmake --build build5/ && ctest --test-dir build5/
Internal ctest changing into directory: /home/ulb/operations/
npotvin/cmake-3.27.9-tutorial-source/build5
Test project /home/ulb/operations/npotvin/cmake-3.27.9-tutori
al-source/build5

    Start 1: Runs
1/3 Test #1: Runs .....................   Passed    0.00 sec
    Start 2: StandardUse
2/3 Test #2: StandardUse ..............   Passed    0.00 sec
    Start 3: Usage
3/3 Test #3: Usage ....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 3
```

Try with a test that will fail, see the output

CMAKE FUNCTION

Target: CMakeLists.txt

```
function(do_test target arg result)
  add_test(NAME Comp${arg} COMMAND ${target} ${arg})
  set_tests_properties(Comp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endfunction()
```

```
do_test(Tutorial 4 "4 is 2")
do_test(Tutorial 9 "9 is 3")
do_test(Tutorial 5 "5 is 2.236")
do_test(Tutorial 7 "7 is 2.645")
do_test(Tutorial 25 "25 is 5")
do_test(Tutorial -25 "-25 is (-nan|nan|0)")
do_test(Tutorial 0.0001 "0.0001 is 0.01")
```

MORE ADVANCED TESTING ? (1/2)

What if we wanted to use C/C++ code to test parts of our project ?

1. Write your test as an executable
2. Add your executable in CMake
3. Add tests using your new executable

```
add_executable(TestThings testThings.cxx)
add_test(NAME TestThing1 COMMAND TestThings arg1)
add_test(NAME TestThing2 COMMAND TestThings arg2)
```

MORE ADVANCED TESTING ? (2/2)

What if we do not want to compile the test targets every time ?

1. Use CMake options
2. Add test targets in a if-then-else block
3. Switch the test option ON/OFF depending on your needs

See ? you know enough to be autonomous

TO GO FURTHER

From here you can use a testing dashboard to monitor your tests (Step 6 of the official tutorial).

You can experiment with other test frameworks (doctest, gtest, CxxTest, *etc.*).

Build and test as github actions.