CMake & Ninja

by István Papp

istvan.papp@ericsson.com

Hello & Disclaimer

- I don't know everything (surprise!), if I stare blankly after a question, go to https://cmake.org/
- Spoiler alert: or https://ninja-build.org/

Contents

Definitions

- A practical view from my perspective, some of these are debatable
- Send me feedback, so 2.0 will be better

What is the goal of a build system?

- Get from source* to binary*

- *Source: source code, text file, assets (textures, audio)
- *Binary: executable, zip file, text file

Requirements

- Speed
- Reliability
- Flexibility

Requirements - Speed

- Fast feedback
  - Catch errors ASAP
  - Avoid breaking stuff for others
  - Conserve resources
- No effect on the compiler*
  - Avoid work
  - Parallel execution

*Build step: zip, upload/download, compilation

Requirements - Reliability

- Umbrella term

- Deterministic
- Stable
- No unexpected behaviour

Requirements - Flexibility

- Large variety of tasks
- Easy to modify
- Easy to read

Sources of complexity

- Source code in multiple directories
- External libraries
- Targeting different platforms
  - Compilers
  - Operating systems
  - Hardware
- Test code
- Mixing languages

Make

- Make is very generic
- Mostly conforms to the requirements
- Designed in 1977 (40 years old!)

- We can do better now

Contents

CMake

- “Cross-Platform Makefile Generator” (source: man cmake)
- Created by a company called Kitware about 17 years ago
- Gained popularity in the last 3-4 years
- Open source software, like most good development tools
- Popular = StackOverflow compatible
- Replaces configuration utilities like autotools

Capabilities – cross-platform

- Runs on Linux, Windows, Mac OSX
- Can compile for Linux, Windows, Mac OSX

- Executable/binary format
- Path separators
- Platform-dependent libraries

Capabilities – in-place & out-of-place

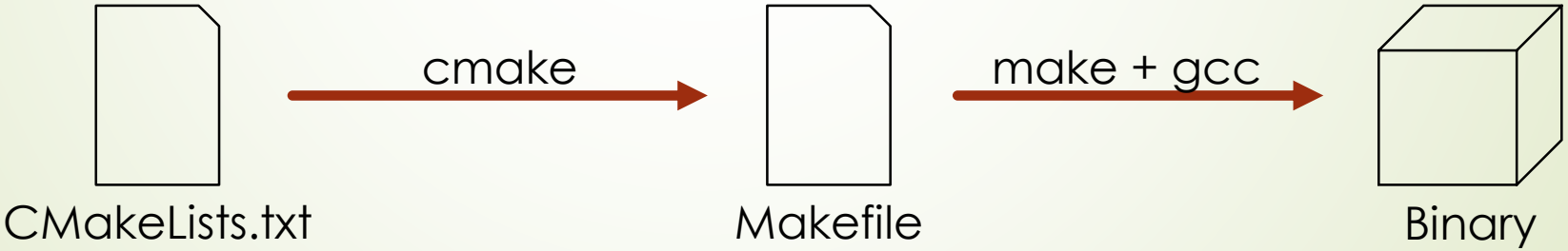
- In-place (in-tree): objects files and binaries mixed with source
  - Easy to do
- Out-of-place (out-of-tree): build artifacts gathered in a dedicated directory
  - Easy to force a clean build
  - Multiple builds in same repo

Capabilities

- Mostly C/C++, supports other languages
- Supports using multiple toolkits
- Supports static and dynamic library builds
- Uses build tools native to the environment
- Has a graphical interface
- Extendable via macros, functions and modules

Build process

1. Generate standard build files from platform independent configuration files.
   - CMakeLists.txt files in every directory.
2. Perform the actual build using native tools.
   - Usually make, gcc, msvc++, whatever the platform has.



Contents

A simple example

```
<project_root>
|--build
|--inc
|   `--<header files>
`--src
    |--main.cc
    `--CMakeLists.txt
```

A simple example

<project_root>/src/CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.8)

project(BestProjectEver)

include_directories(../inc)

add_executable(BestProjectEver main.cc)
```

Adding a library

```
<project_root>
|--build
|--inc
|  `--<header files>
|--src
|  |--main.cc
|  `--CMakeLists.txt
`--graphics
   |--inc
   |  `--<library header files>
   |--src
   |  |--bells.cc
   |  |--whistes.cc
   |  `--CMakeLists.txt
   `--CMakeLists.txt
```

Adding a library

<project_root>/src/CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.8)
project(BestProjectEver)

add_subdirectory(../graphics)

include_directories(../inc)
include_directories(../graphics/inc)

add_executable(BestProjectEver main.cc)

target_link_libraries(BestProjectEver Graphics)
```

Adding a library

<project_root>/graphics/CMakeLists.txt:

```
add_subdirectory(src)

# this file could be skipped by pointing right at
# <project_root>/graphics/src/CMakeLists.txt in the
# first file
```

Adding a library

<project_root>/graphics/src/CMakeLists.txt:

```
include_directories(../inc)

add_library(Graphics bells.cc whistles.cc)
# the name will be libGraphics.a or Graphics.lib,
# depending on the platform
```

Using the example

```
cd <project_root>/build
cmake ../src && make
```

- Binaries by default go into the directory where you start cmake
- The argument is the directory where the starting CMakeLists.txt lives

Contents

Variables

```
1  set(FOO "bar")
2  set(RESULT "Progress ${FOO}...")
3
4
5  set(SRC file1.cc file2.cc file3.cc)
6  file(GLOB SRC "*.cc")
7
8  add_executable(MyLittleProject ${SRC})
```

Variables

```
1  set(CMAKE_CXX_STANDARD 11)
2  set(CMAKE_CXX_FLAGS
3      "${CMAKE_CXX_FLAGS} -std=c++11 -W -Wall -pedantic")
```

Lists

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


# empty string, FALSE, NO, OFF, or any string
# ending in -NOTFOUND all evaluate to false
```

Lists

```
set(FOO a b c)
list(APPEND FOO a b c)


list(LENGTH FOO result)
message(${result}) # prints 3
```

Conditionals

```
1   set(CMAKE_CXX_COMPILER "/usr/bin/g++")
2   project(MyLittleProject)
3
4
5   if("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
6       set(WARNING_FLAGS
7           "${WARNING_FLAGS} -Wsomething-only-gcc-knows")
8   elseif("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
9       set(WARNING_FLAGS
10          "${WARNING_FLAGS} -Wsomething-only-clang-knows")
11  endif()
12  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${WARNING_FLAGS}")
```

Conditionals

```
find_package(Boost 1.53.0 REQUIRED COMPONENTS filesystem regex)
if(Boost_FOUND)
    include_directories(${Boost_INCLUDE_DIR})
endif()

# looks for a FindBoost.cmake
```

Formatting

```
# you guessed right, these are comments

find_package(
    Boost
        1.53.0
    REQUIRED
    COMPONENTS
        filesystem
        regex
)
```

Other rules

```
MESSAGE(hi) # prints "hi"
message(hi) # prints "hi" again
message(HI) # prints "HI"

math(EXPR x "3 + 3") # x will be 6

# no declarations
# variables are global from the current directory down
```

Everything else

- Iteration: foreach(), while()
- Platform inspection: check_function_exists()
- Reuse: add_custom_command(), macro(), function()
- Extension: include() files from CMAKE_MODULE_PATH

Now you know how to read the documentation

Contents

CTest

Test driver for unit and component tests

1. Add enable_testing() to your listfile
2. Add testcases with add_test()
3. Run your tests with ctest
4. ???
5. Profit!

CPack

- Installation: install_*() commands
- Distribution: include(CPack), cpack_*() commands
  - tar.gz, zip, deb, rpm, etc.
- cpack --config <your_config>.cmake

Contents

Ninja

Small build system with a focus on speed

- Generated input
  - Still human-readable
- Prefer speed over convenience
- Do one thing, and do it well

How?

- Dependency of files as input
- No unnecessary decisions
  - Compilers?
  - Compiler flags?
  - Debug or release?
- The bare minimum to describe dependency graphs
  - Ninja doesn't know about your language

Features

- Multiplatform
- Very fast when there's nothing to do
  - Think “incremental build”
- One environment variable: NINJA_STATUS
  - Controls the output’s format

Some more nice features

- Outputs depend on the command line
  - Changing the compilation flags will cause a rebuild
- Builds are parallel by default
  - Need correct dependencies
  - Run `ninja` with `nice`
- Command output is buffered

How to write your own build.ninja files

- Don't

build.ninja syntax

- variables (aliases for strings)

```
<variable> = <value>
```

- build statements (how to do things)

```
build <outputs>: <rulename> <inputs>
```

- rules (what things to do)

```
rule <rulename>
    <variable> = <value>
    <variable> = <value>
```

Example build.ninja

```
cflags = -Wall

rule cc
  command = gcc $cflags -c $in -o $out

build foo.o: cc foo.c
```

Contents

Tying it all together

- CMake supports multiple generators

```
cmake –G “Unix Makefiles”
cmake –G “Ninja”
```

- Makefiles work well, but Ninja was designed for this

Summary

- Speed: handled by Ninja
- Flexibility: provided by CMake
- Reliability: both seem to be reliable so far

- Use CMake with Ninja
- Look for better alternatives for existing tools

Thanks for listening & Questions

Contact me at istvan.papp@ericsson.com