CMAKE

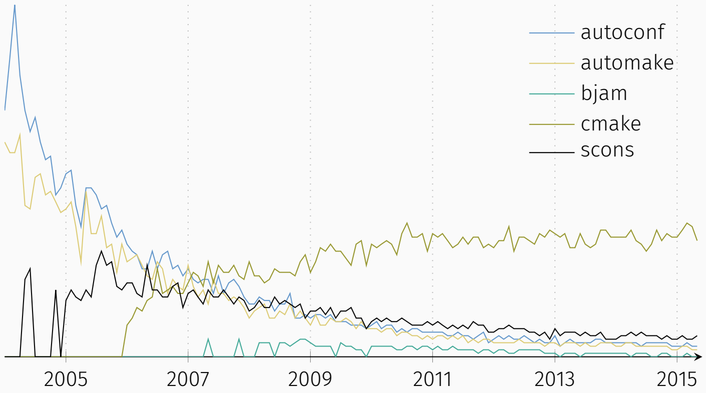
Introduction and best practices

Daniel Pfeifer

May 21, 2015

daniel@pfeifer-mail.de

INTEREST OVER TIME



[0] https://www.google.com/trends/explore#q=autoconf,automake,bjam,cmake,scons

WHAT IS CMAKE?

ACCORDING TO ITS WEBSITE

Welcome to CMake, the **cross-platform, open-source build system**.

CMake is a **family of tools** designed to build, test and package software. CMake is used to control the software compilation process using simple **platform and compiler independent** configuration files.

CMake **generates** native makefiles and workspaces that can be used in the compiler environment of your choice.[1]

[1]http://www.cmake.org/

cmake's position among build systems

standalone	Make, NMake, SCons, Jom, BJam, Ninja
integrated	Visual Studio, Xcode, Eclipse
generators	Autotools, CMake, GYP

CMAKE IS SIMPLE

```
> ls zlib-1.2.8
amiga/          as400/        contrib/          doc/
examples/       msdos/        nintendods/       old/
qnx/            test/         watcom/           win32/
adler32.c       ChangeLog     CMakeLists.txt    compress.c
configure       crc32.c       crc32.h           deflate.c
deflate.h       FAQ           gzclose.c         gzguts.h
gzlib.c         gzread.c      gzwrite.c         INDEX
infback.c       inffast.c     inffast.h         inffixed.h
inflate.c       inflate.h     inftrees.c        inftrees.h
make_vms.com    Makefile      Makefile.in       README
treebuild.xml   trees.c       trees.h           uncompr.c
zconf.h         zconf.h.in    zlib.3            zlib.3.pdf
zlib.h          zlib.map      zlib.pc.in        zlib2ansi
zutil.c         zutil.h
```

REDUCE YOUR MAINTENANCE ...

You	write a single configuration that CMake understands.
CMake	takes care that it works on all compilers and platforms.

… unless you break it!

Don’t make *any* assumption about the platform or compiler!

CMAKE IS FAST!

I was quite surprised with the speed of building Quantum GIS codebase in comparison to Autotools.[2]

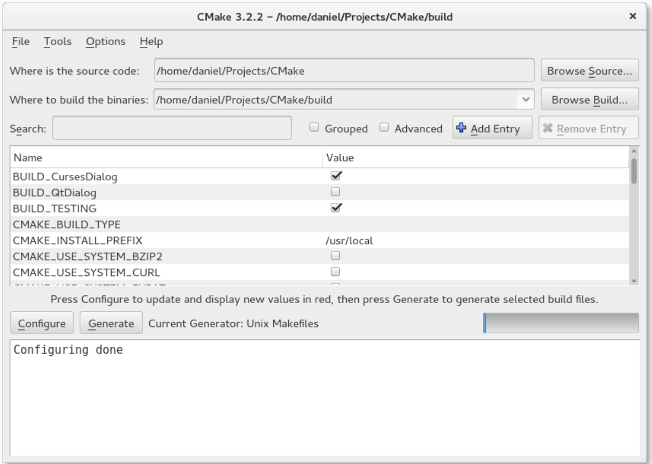
Task	CMake	Autotools
Configure	0:08	0:61
Make	12:15	21:16
Install	0:20	0:36
Total	12:43	22:43

[2] http://blog.qgis.org/node/16.html

THE CMAKE CACHE

- Compiler and platform test results are cached in `CMakeCache.txt`.
- You can modify this file and re-run CMake.
- CMake also provides graphical cache editors.

CMAKE'S GRAPHICAL CACHE EDITOR



Where is the source code: /home/daniel/Projects/CMake

Where to build the binaries: /home/daniel/Projects/CMake/build

Search:

Name	Value
BUILD_CursesDialog	✓
BUILD_QtDialog	☐
BUILD_TESTING	✓
CMAKE_BUILD_TYPE	
CMAKE_INSTALL_PREFIX	/usr/local
CMAKE_USE_SYSTEM_BZIP2	☐
CMAKE_USE_SYSTEM_CURL	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
Configuring done
```

CMAKE'S POSITION ON THE TIME AXIS

past	CMake generates build systems.
today	CMake is integrated into IDEs.
future	IDEs run CMake as a background process.[3]

[3] http://public.kitware.com/pipermail/cmake-developers/2015-April/025012.html

MODES OF OPERATION

CMAKE AS A BUILD SYSTEM GENERATOR

```
> cmake [<options>] <path>
```

- If the specified path contains a CMakeCache.txt, it is treated as a build directory where the build system is reconfigured and regenerated.
- Otherwise the specified path is treated as the source directory and the current working directory is treated as build directory.

```
> mkdir build
> cd build
> time cmake ..
> time cmake .
> time make
```

CMAKE AS A BUILD SYSTEM GENERATOR

Issues:

- The commands `mkdir` and `time` are not portable.
- With `cmake .` you may accidentally perform an in-source configure.
- Who said that `make` is the right build command?

```
> mkdir build
> cd build
> time cmake ..
> time cmake .
> time make mylibrary
```

cmake's command-line tool mode

```
> cmake -E <command> [<options>...]
```

Run `cmake -E help` for a summary of commands.

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

USE CMAKE TO RUN THE BUILD SYSTEM

```
> cmake --build build \
        --target mylibrary \
        --config Release \
        --clean-first
```

Just don't:

```
> ‘grep ’CMAKE_MAKE_PROGRAM:’ CMakeCache.txt \
    | awk -F= ’{print $2}’‘ mylibrary
```

CMAKE AS A SCRIPTING LANGUAGE

```
> cat hello_world.cmake
foreach(greet Hello Goodbye)
  message(”${greet}, World!”)
endforeach()

> cmake -P hello\_world.cmake
Hello, World!
Goodbye, World!
```

No configure or generate step is performed and the cache is not modified.

Resist the bash/perl/python temptation!

COMMANDS FOR PROJECTS

CMAKE_MINIMUM_REQUIRED()

Sets the mimimum required version of CMake.

```
cmake_minimum_required(VERSION 3.2 FATAL_ERROR)
```

- Prefer the latest version of CMake.
- Please don't set 2.8 as the minimum.
- If you use 2.6 or 2.4, God kills a kitten.

PROJECT()

Set the name and version; enable languages.

```
project(<name> VERSION <version> LANGUAGES CXX)
```

- CMake sets several variables based on project().
- Call to project() must be direct, not through a function/macro/include.
- CMake will add a call to project() if not found on the top level.

ADD_SUBDIRECTORY()

Embed projects as sub-projects.

```
add_subdirectory(<sourcedir> [<binarydir>])
```

- CMake creates a Makefile/Solution for each subproject.
- The subproject does not have to reside in a subfolder.

HINT!

Make sure that all your projects can be built both **standalone** and as a **subproject** of another project:

- Don't assume that your project root is the build root.
- Don't modify global compile/link flags.
- Don't make any global changes!

FIND_PACKAGE()

Finds preinstalled dependencies

```
find_package(Qt5 REQUIRED COMPONENTS Widgets)
```

- Can set some variables and define imported targets.
- Supports components.

ADD_EXECUTABLE()

Add an executable target.

```
add_executable(tool
  main.cpp
  another_file.cpp
  )
add_executable(my::tool ALIAS tool)
```

ADD_LIBRARY()

Add a library target.

```
add_library(foo STATIC
  foo1.cpp
  foo2.cpp
  )
add_library(my::foo ALIAS foo)
```

- Libraries can be `STATIC`, `SHARED`, `MODULE`, or `INTERFACE`.
- Default can be controlled with `BUILD_SHARED_LIBS`.

THINGS TO REMEMBER

- Always add namespaced aliases for libraries.
- Dont't make libraries `STATIC`/`SHARED` unless they cannot be built otherwise.
- Leave the control of `BUILD_SHARED_LIBS` to your clients!

USAGE REQUIREMENTS

USAGE REQUIREMENTS SIGNATURE

```
target_<usage requirement>(<target>
  <PRIVATE|PUBLIC|INTERFACE> <lib> ...
  [<PRIVATE|PUBLIC|INTERFACE> <lib> ... ] ...]
  )
```

PRIVATE	Only used for this target.
PUBLIC	Used for this target and all targets that link against it.
INTERFACE	Only used for targets that link against this library.

TARGET_LINK_LIBRARIES()

Set libraries as dependency.

```
target_link_libraries(foobar
  PUBLIC  my::foo
  PRIVATE my::bar
  )
```

THINGS TO REMEMBER

- Prefer to link against namespaced targets.
- Specify the dependencies are private or public.
- Avoid the `link_libraries()` command.
- Avoid the `link_directories()` command.
- No need to call `add_dependencies()`.

TARGET_INCLUDE_DIRECTORIES()

Set include directories.

```
target_include_directories(foo
  PUBLIC  include
  PRIVATE src
  )
```

THINGS TO REMEMBER

- Avoid the `include_directories()` command.

TARGET_COMPILE_DEFINITIONS()

Set compile definitions (preprocessor constants).

```
target_compile_definitions(foo
  PRIVATE SRC_DIR=${Foo_SOURCE_DIR}
  )
```

THINGS TO REMEMBER

- Avoid the `add_definitions()` command.
- Avoid adding definitions to `CMAKE_<LANG>_FLAGS`.

TARGET_COMPILE_OPTIONS()

Set compile options/flags.

```
if(CMAKE_COMPILER_IS_GNUCXX)
  target_compile_options(foo
    PUBLIC  -fno-elide-constructors
    )
endif()
```

THINGS TO REMEMBER

- Wrap compiler specific options in an appropriate condition.
- Avoid the `add_compile_options()` command.
- Avoid adding options to `CMAKE_<LANG>_FLAGS`.

TARGET_COMPILE_FEATURES()

Set required compiler features.

```
target_compile_features(foo
  PUBLIC
    cxx_auto_type
    cxx_range_for
  PRIVATE
    cxx_variadic_templates
  )
```

- CMake will add required compile flags.
- Errors if the compiler is not supported.

THINGS TO REMEMBER

- Don’t add `-std=c++11` to `CMAKE_<LANG>_FLAGS`.
- Don’t pass `-std=c++11` to `target_compile_options()`.

BEST PRACTICES

GOAL: NO CUSTOM VARIABLES

VARIABLES ARE FRAGILE

```
set(PROJECT foobar)
set(LIBRARIES foo)

if(WIN32)
  list(APPEND LIBRARIES bar)
endif()

target_link_libraries(${Project}
  ${LIBRARIES}
  )
```

VARIABLES ARE FRAGILE

Hard to diagnose:

bar links against *foo*! But only on Windows!

```
set(PROJECT foobar)
set(LIBRARIES foo)

if(WIN32)
  list(APPEND LIBRARIES bar)
endif()

target_link_libraries(${Project}
  ${LIBRARIES}
  )
```

VARIABLES ARE FRAGILE

This is better:

Without variables the code is more robust and easier to read.

```
target_link_libraries(foobar PRIVATE my::foo)

if(WIN32)
  target_link_libraries(foobar PRIVATE my::bar)
endif()
```

DON'T USE FILE(GLOB)!

Not recommended:

```
file(GLOB sources "*.cpp")
add_library(mylib ${sources})
```

- `file(GLOB)` is useful in scripting mode.
- Don't use it in configure mode!

LIST ALL SOURCES EXPLICITLY

```cmake
add_library(mylib
  main.cpp
  file1.cpp
  file2.cpp
  )
```

Explicit is better than implicit

GOAL: NO CUSTOM FUNCTIONS

PICKING ON MYSELF

- I wrote many custom functions.
- Experience tought me: Most of the time it is a bad idea.

JUST DON'T

```
magic_project(<name> ...)
```

WHY NO CUSTOM FUNCTIONS?

- Remember what I said about the `project()` command?
- The signature of `project()` was extended recently.

JUST DON'T

```
magic_add_library(<name>
  SOURCES
    <list of sources>
  LIBRARIES
    <list of libraries>
  )
```

WHY NO CUSTOM FUNCTIONS?

- How to add platform specific libraries?
- Harder to get started for contributors.
- Maintenance!

SO YOU WROTE A CUSTOM FUNCTION ...

- Contribute it to CMake!
- If it is accepted, it is no longer a *custom function*.
- Otherwise, the reason for rejection should give you a hint.
- The maintainers are very helpful and experienced.

SUMMARY

SUMMARY

- CMake is widely used, fast, cross-platform.
- Build system generator plus platform abstraction.
- Manage build targets with usage requirements.
- Goal: No custom variables.
- Goal: No custom functions.

Thank you!