Introduction to CMake

Florent Castelli
florent.castelli@gmail.com

SwedenCpp 0xC
April 26th 2018

https://bit.ly/swedencpp-cmake-slides

Introduction

CMake features

Build file generator

Compiler independent configuration files

Uses CMake language

Enables building, testing and packaging of software

CMake features (2)

Cross-platform (Windows, macOS, Linux, ...)

Open source

Backwards compatibility with older scripts

Generates projects for major IDEs / build tools

- Microsoft Visual Studio
- Xcode
- Ninja
- Make

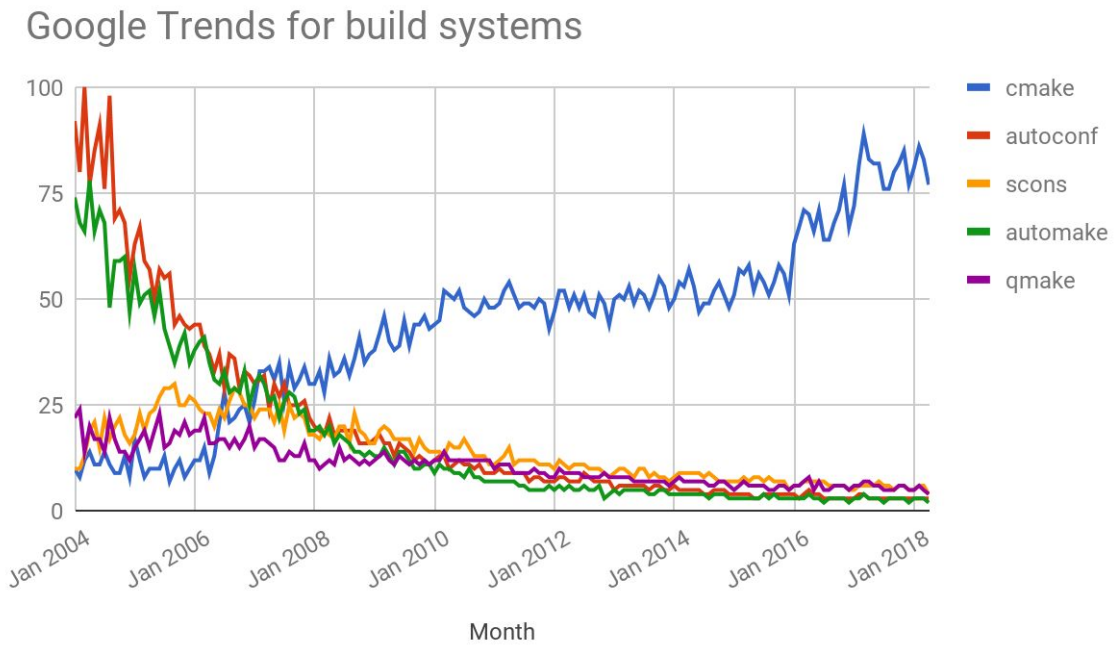
Direct CMake integration

- Microsoft Visual Studio 2017
- QtCreator
- JetBrains CLion
- Android Studio / Gradle

Supported platform targets (not exhaustive)

- Windows
- macOS / iOS
- Linux
- Android

Why CMake?


Google Trends for build systems
100
75
50
25
0
Jan 2004
Jan 2006
Jan 2008
Jan 2010
Jan 2012
Jan 2014
Jan 2016
Jan 2018
Month
cmake
autoconf
scons
automake
qmake

How to install

Prebuilt binaries for Windows, macOS and Linux on CMake.org

macOS: Use homebrew "`brew install cmake`"

Linux: Use your package manager, but they do not all have the latest version. Remember that it's fine to use a prebuilt binary from cmake.org, for example in a closed enterprise context to have a more recent version.

Bundled with Visual Studio 2017 or Android Studio.

Hello CMake!

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)
project(hello-world)

add_executable(hello-world
  hello.cpp
)
```

To build and run

```
> mkdir -p out
> cd out
> cmake ..
> cmake --build .
[100%] Built target hello-world
> ./hello-world
Hello world!
```

Interpreter mode

```
> cmake -P script.cmake
```

CMake language

Command based, one per line

```
set(FOO "bar")
add_executable(foo bar.cpp)
if(FOO)
```

Commands don't return values: no nesting

Commands may have arguments and overloads

```
file(WRITE <file> <content>)
file(READ <file> <variable>)
```

See the documentation

Variables are set with command `set()`, removed with `unset()`

Variables are all strings

```
set(FOO "bar")
set(FOO bar)
set(FOO 42)
```

Lists are strings too, semicolon separated

```
set(FOO "1;2;3")
set(FOO 1 2 3)
```

Variables are read using `${<var>}`

```
set(FOO ${BAR})
```

CMake language (2)

Control flow

`if()` / `elseif()` / `else()` / `endif()`

`foreach()` / `endforeach()`

`while()` / `endwhile()`

`break()` / `continue()` / `return()`

Comments starts with **#**

```
# I am a comment
```

Others

include(<file>)
Read another CMake files in the same context

add_subdirectory(<dir>)
Read another `CMakeLists.txt` file in <dir> in a new context

message(<text>)
Print a message, useful for displaying status, progress, warnings or errors

CMake variable scope

Variables are scoped

Each new scope creates a local copy of all variables

Scopes created by **`add_subdirectory`**`()` or custom function call

Top level scope is the **`CACHE`**, can serve as global variables. Values are kept between runs.

Can prepopulate the cache variables with `-D<var>=<val>` on the command line

Examples

```
set(FOO "bar")

set(FOO "bar" PARENT_SCOPE)

set(GFOO "42" CACHE STRING "doc")

set(GFOO "42" CACHE STRING "doc" FORCE)

> cmake -DGFOO=42 .
```

Fizzbuzz

```
function(fizzbuzz n)
  foreach(i RANGE 1 ${n})
    set(result "")
    math(EXPR fizz "${i} % 3")
    math(EXPR buzz "${i} % 5")
    if(NOT fizz)
      string(APPEND result "fizz")
    endif()
    if(NOT buzz)
      string(APPEND result "buzz")
    endif()
    if(NOT result)
      set(result "${i}")
    endif()
    message("${result}")
  endforeach()
endfunction()
fizzbuzz(15)
```

```
>  cmake -P fizzbuzz.cmake
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
```

Creating a CMake project

Boilerplate

CMake entry point: **`CMakeLists.txt`**

Put it at your project's root

Checks the CMake version and detects the compiler

Recurse in sub-folders of each executable or library

May have project wide options

Try to keep it simple!

```
cmake_minimum_required(VERSION x.y)
project(<name>)

option(SECRET_FEATURE "Enable secret feature" OFF)

add_subdirectory(mylib)
add_subdirectory(myexe)
```

Creating a target

Executable

```
add_executable(<name>
    [WIN32]  [MACOSX_BUNDLE]
    [source1] [source2 ...])
```

Library

```
add_library(<name>
    [STATIC | SHARED | INTERFACE]
    [source1] [source2 ...])
```

Custom target

```
add_custom_target(<name>
    [COMMAND command] [args]
    [DEPENDS depends]
    [WORKING_DIRECTORY dir])
```

Creating a target (2)

Library types

- SHARED
  Shared library (.dll, .so, .dylib)
- STATIC
  Static library (.lib, .a)
- INTERFACE
  Virtual target, usually used for header only libraries
  Doesn’t show up in IDEs (no target or files)

INTERFACE libraries are not real targets, they don’t have any file

For easier development in IDEs, it’s best to have header-only libraries in a STATIC library with a dummy source file to silence archiver warnings.

Target configuration

For both executable and libraries, use:

```
target_include_directories(<target>
    [PUBLIC | INTERFACE | PRIVATE ]
    <dir>)

target_compile_definitions(<target>
    [PUBLIC | INTERFACE | PRIVATE ]
    <name>=<value>)

target_compile_options(<target>
    [PUBLIC | INTERFACE | PRIVATE ]
    <option>)
```

Example:

```
target_include_directories(mylib
    PUBLIC include
    PRIVATE src)

target_compile_definitions(mylib
    PRIVATE FASTPATH=1)

target_compile_options(mylib
    PRIVATE -Wall)
```

Target configuration (2)

Adding a dependency on another target

```
target_link_libraries(<target>
    [PUBLIC | PRIVATE | INTERFACE]
    <lib>)
```

Example

```
target_link_libraries(myexe
    PRIVATE mylib)
```

Three types of dependencies

PRIVATE
Options are not propagated to users of the library, but users to build the current target.

INTERFACE
Options are propagated to users of the library, but not used to build the current target.

PUBLIC
Options are propagated to users of the library and used to build the current target.

Library example

CMakeLists.txt

```
add_library(mylib STATIC
    include/mylib/mylib.h
    src/mylib_impl.cpp
    src/mylib_impl.h
)
target_include_directories(mylib
    PUBLIC include
    PRIVATE src
)
target_link_libraries(mylib
    PUBLIC Boost::boost
)
```

Files

```
mylib/

  CMakeLists.txt

  include/mylib/
    mylib.h

  src/
    mylib_impl.cpp
    mylib_impl.h
```

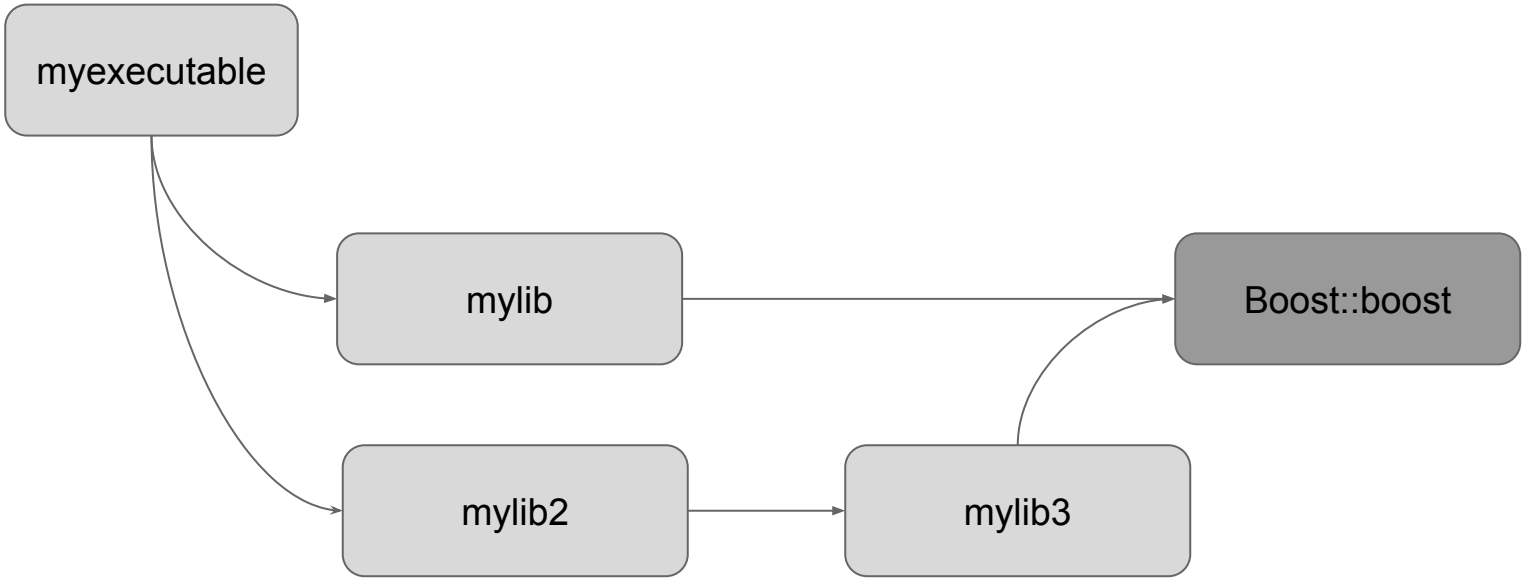
Legacy CMake commands

They have global effect and are highly discouraged for configuring regular targets

```
include_directories()

add_definitions()

add_dependencies()

add_compile_options()
```

Those have lots of side-effects and break the composability of CMake projects.

Example project



External libraries

Multiple ways to use 3rd party code

“Vendor” the library and copy its sources in your repository. Best if built with modern CMake.

Use an external CMake dependency manager (see Conan, Hunter or vcpkg)

Use CMake’s **`find_package`**`(<name>)`.
Uses `Find<name>.cmake` or `<name>Config.cmake` scripts, which you can write to locate the libraries and headers or use the ones bundled with CMake (see list).

Use CMake’s **`ExternalProject*`**`()` or **`FetchContent*`**`()`

When targeting multiple platforms, or using ABI changing compilation options, it is easier to either “vendor” a library or to use a smarter dependency manager.

Otherwise, you risk mixing incompatible binaries together by accident.
Management of prebuilt binaries is quite error prone.

Testing with CMake

```
add_executable(mytest mytest.cpp)

enable_testing()

add_test(NAME mytest
    COMMAND mytest --option)

> make mytest

> ctest .
```

Choosing a configuration

CMake supports multiple configurations by default

Debug: No optimization, debug information

Release: Optimizations, no debug information

RelWithDebInfo: Optimizations, debug information

MinSizeRel: Optimization for size, no debug information

Some generators support multiple configuration at once (Visual Studio, Xcode), others only one (Make, Ninja)

```
> cmake .. -G”Visual Studio 15 2017 x64”

> msbuild project.sln /p:Configuration=Release

> cmake .. -GNinja -DCMAKE_BUILD_TYPE=Debug

> ninja
```

Why modern CMake?

Legacy CMake

```
find_package(Boost REQUIRED
    COMPONENTS thread system)

# Everyone can use Boost now!
include_directories(${Boost_INCLUDE_DIRS})
# The order is important!
# thread uses system and needs to be first
target_link_libraries(mylib
     ${Boost_THREAD_LIBRARY}
     ${Boost_SYSTEM_LIBRARY})
```

Modern CMake

```
find_package(Boost REQUIRED
    COMPONENTS thread system)

target_link_libraries(mylib
    Boost::thread)
```

Concise, no passing of variables, correct.

Other libraries might require to pass defines or add dependencies manually, which is easy to forget.

Best practices for modern CMake

Each project should be embeddable in another with `add_subdirectory()`.

Targets should be self-sufficient and declare all their direct dependencies.

Targets should not have any cyclic dependency.

Targets should not rely on global commands (e.g. `include_directories()`).

Use targets, avoid variables!

Keep it declarative

List all your files, including headers in your targets

Only use files that are in the same folder or below, ".." leads to bad separation

Wrap your target definitions in a function, allowing factorization of code and easily make large scale changes. (See example and usage)

Advanced CMake Concepts & Examples

CMake predefined variables

Lots of variables automatically set by CMake.

Some noteworthy ones:

CMAKE_CURRENT_BINARY_DIR
CMAKE_CURRENT_LIST_DIR
Current source folder and bin folder in the build tree

CMAKE_SYSTEM_NAME
Name of the platform you are targeting

CMAKE_{C/CXX}_COMPILER_ID
String identifying the compiler

CMAKE_{C/CXX}_FLAGS
Global compiler flags

Example

```
add_library(mylib STATIC
  mylib.h
  mylib.cpp
)
if(CMAKE_SYSTEM_NAME STREQUAL "Android")
  target_sources(mylib
    PRIVATE mylib_android.cpp)
endif()
```

See full list of variables

CMake properties

CMake stores state in properties on objects

Main objects are global scope, directories, targets, tests, source files.

Most commands will just update properties of targets

Can be used for introspection or enabling some advanced features.

See property list

Example

```
get_property(mylib_sources
    TARGET mylib
    PROPERTY SOURCES)

message("mylib: ${mylib_sources}")

set_property(TEST mytest
    PROPERTY WILL_FAIL TRUE)

set_property(SOURCE test.cpp
    PROPERTY COMPILE_FLAGS "-Wall")
```

Generator expressions

CMake execution is in 2 phases, first running the scripts, then generating the build files

Some decisions can't be made while running the scripts, so you want to delay them in the second phase.

Allows full declarative CMake usage

See generator expressions

Example

```
set_property(TARGET mylib
  PROPERTY OUTPUT_NAME "mylib$< CONFIG>")
> cmake -DCMAKE_BUILD_TYPE=Debug ..
> make
[100%] Linking CXX static library
libmylibDebug.a

add_library(mylib STATIC
  mylib.cpp
  $<IF:$<PLATFORM_ID:Android>, android.cpp>
  $<IF:$<PLATFORM_ID:Windows>, windows.cpp>
)
```

CMake toolchains

Declares a toolchain: compiler, options, build flags, linker flags, file extensions…

Can also reuse a built-in toolchain and customize it

Enables cross-compilation

See toolchain documentation

Example: android.cmake

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 21) # API level
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_STL_TYPE gnustl_static)

> cmake .. -DCMAKE_TOOLCHAIN_FILE=android.cmake
```

Clang-Tidy / IWYU integration

Runs clang-tidy alongside compilation with Make or Ninja generators.

“Clang-Tidy is a linter tool. It checks for typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis.”

Set clang-tidy documentation

Include-What-You-Use is a linter tool checking for includes in your files, suggesting fewer includes or forward declarations for faster compilation time.

```
find_program(CLANG_TIDY_BIN
    NAMES “clang-tidy”)

set(CLANG_TIDY_RUN “${CLANG_TIDY_BIN}”
“-checks=-*,bugprone-*,cppcoreguidelines-*”)

set_property(TARGET mylib
  PROPERTY CXX_CLANG_TIDY ${CLANG_TIDY_RUN}”)

find_program(IWYU_BIN
    NAMES “iwyu”)

set_property(TARGET mylib
  PROPERTY CXX_INCLUDE_WHAT_YOU_USE
  “${IWYU_BIN}” “--transitive_includes_only”)
```

Faster builds with Ninja & CCache

Ninja generator usually faster than alternatives

Uses parallelism by default, for the whole project

Has better parallelism opportunities: can build several targets in parallel that depend on each other, for example if they are static libraries with no generated files.

Allows “ccache” or “sccache” usage, compiler caches for faster rebuilds.

```
find_program(CCACHE_BIN
    NAMES ccache sccache)

if(CCACHE_BIN)
  set_property(GLOBAL
    PROPERTY RULE_LAUNCH_COMPILE ${CCACHE_BIN})
endif()

> cmake .. -GNinja
> ninja -v
ccache /usr/bin/clang++-5.0 ...
```

Library importing with FetchContent

Downloads a project from an archive or repository at configuration time.

Allows using **`add_subdirectory`**`()` to use an external library in your project.

Allows downloading tools before usage (for example a linter like clang-tidy).

See FetchContent documentation

```
include(FetchContent)
FetchContent_Declare(
  gtest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG release-1.8.0
)
FetchContent_GetProperties(gtest)
if(NOT gtest_POPULATED)
  FetchContent_Populate(gtest)
  add_subdirectory(${gtest_SOURCE_DIR}
                   ${gtest_BINARY_DIR})
endif()
```

Force linking a library

Static libraries are only linked if any symbol is used

For “plugin” libraries that only have global objects that register themselves, it doesn’t work

We can use a workaround to automatically create a file with a symbol that is referenced in the library, forcing the linker to link it without any special flag.

```
function(force_link lib)
  set(FL_FILE
“${CMAKE_CURRENT_BINARY_DIR}/fl_${lib}”)

  file(WRITE “${FL_FILE}_ext.cpp“
             “extern int fl_${lib}; fl_${lib} = 0;”)
  file(WRITE “${FL_FILE}_decl.cpp”
             “int fl_${lib} = 0;”)

  target_sources(${lib}
    PRIVATE “${FL_FILE}_decl.cpp”
    INTERFACE “${FL_FILE}_ext.cpp”)
endfunction()
```

So much more to learn!

Packaging software

So many modules

And variables

And 3rd party CMake modules:

cotire: Unity builds from CMake

polly: Collection of toolchain files

Questions ?

Workshop material

https://bit.ly/swedencpp-cmake-slides
https://bit.ly/swedencpp-cmake-workshop