Modern C++ Programming

17. C++ Ecosystem

CMake and Other Tools

Federico Busato

2025-02-06

Table of Contents

4 Other Tools

CMake

CMake Overview

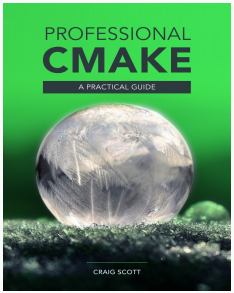
`CMake` is an *open-source*, *cross-platform* family of tools designed to build, test and package software

`CMake` is used to control the software compilation process using simple platform and compiler independent configuration files, and *generate* native `Makefile/Ninja` and workspaces that can be used in the compiler environment of your choice

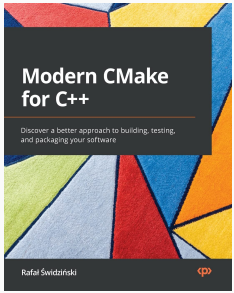
`CMake` features:

- Turing complete language (if/else, loops, functions, etc.)
- Multi-platform (Windows, Linux, etc.)
- Open-Source
- Generate: `makefile, ninja, etc.`
- Supported by many IDEs: `Visual Studio, Clion, Eclipse, etc.`

CMake Books



Professional CMake: A Practical Guide (14th)
C. Scott, 2023



Modern CMake for C++ (2nd)
R. Świdziński, 2024

CMake - References

- 19 reasons why CMake is actually awesome
- An Introduction to Modern CMake
- Effective Modern CMake
- Awesome CMake
- Useful Variables

Install CMake

Using PPA repository

```
$ wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null |
  gpg --dearmor - | sudo tee /etc/apt/trusted.gpg.d/kitware.gpg >/dev/null
$ sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ focal main' # bionic, xenial
$ sudo apt update
$ sudo apt install cmake cmake-curses-gui
```

Using the installer or the pre-compiled binaries: `cmake.org/download/`

```
# download the last cmake package, e.g. cmake-x.y.z-linux-x86_64.sh
$ sudo sh cmake-x.y.z-linux-x86_64.sh
```

A Minimal Example

CMakeLists.txt:

```
project(my_project)                 # project name

add_executable(program program.cpp) # compile command
```

```
# we are in the project root dir
$ mkdir build     # 'build' dir is needed to isolate temporary files
$ cd build
$ cmake ..        # search for CMakeLists.txt directory
$ cmake --build . # makefile automatically generated, -j to parallelize the build

Scanning dependencies of target program
[100%] Building CXX object CMakeFiles/out_program.dir/program.cpp.o
Linking CXX executable program
[100%] Built target program
```

Parameters and Message

CMakeLists.txt:

```
project(my_project)
add_executable(program program.cpp)

if (VAR)
    message("VAR is set, NUM is ${NUM}")
else()
    message(FATAL_ERROR "VAR is not set")
endif()
```

```
$ cmake ..
VAR is not set
$ cmake -DVAR=ON -DNUM=4 ..
VAR is set, NUM is 4
...
[100%] Built target program
```

Language Properties

```
project(my_project
        DESCRIPTION  "Hello World"
        HOMEPAGE_URL "github.com/"
        LANGUAGES     CXX)

cmake_minimum_required(VERSION 3.15)

set(CMAKE_CXX_STANDARD           14) # force C++14
set(CMAKE_CXX_STANDARD_REQUIRED  ON)
set(CMAKE_CXX_EXTENSIONS         OFF) # no compiler extensions

add_executable(program ${PROJECT_SOURCE_DIR}/program.cpp)  #$
# PROJECT_SOURCE_DIR is the root directory of the project
```

Target Commands

```
add_executable(program) # also add_library(program)

target_include_directories(program
                           PUBLIC  include/
                           PRIVATE src/)
# target_include_directories(program SYSTEM ...) for system headers

target_sources(program                      # best way for specifying
               PRIVATE src/program1.cpp     # program sources and headers
               PRIVATE src/program2.cpp
               PUBLIC  include/header.hpp)

target_compile_definitions(program PRIVATE MY_MACRO=ABCEF)

target_compile_options(program PRIVATE -g)

target_link_libraries(program PRIVATE boost_lib)

target_link_options(program PRIVATE -s)
```

Build Types

```
project(my_project)                          # project name
cmake_minimum_required(VERSION 3.15)         # minimum version

add_executable(program program.cpp)

if (CMAKE_BUILD_TYPE STREQUAL "Debug")       # "Debug" mode
                                             # cmake already adds "-g -O0"
    message("DEBUG mode")
    if (CMAKE_COMPILER_IS_GNUCXX)            # if compiler is gcc
        target_compile_options(program "-g3")
    endif()
elseif (CMAKE_BUILD_TYPE STREQUAL "Release") # "Release" mode
    message("RELEASE mode")                   # cmake already adds "-O3 -DNDEBUG"
endif()
```

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Custom Targets and File Managing

```
project(my_project)
add_executable(program)

add_custom_target(echo_target            # makefile target name
                  COMMAND echo "Hello"   # real command
                  COMMENT "Echo target")

# find all .cpp file in src/ directory
file(GLOB_RECURSE SRCS ${PROJECT_SOURCE_DIR}/src/*.cpp)
# compile all *.cpp file
target_sources(program PRIVATE ${SRCS}) # prefer the explicit file list instead
```

```
$ cmake ..
$ make echo_target
```

Local and Cached Variables

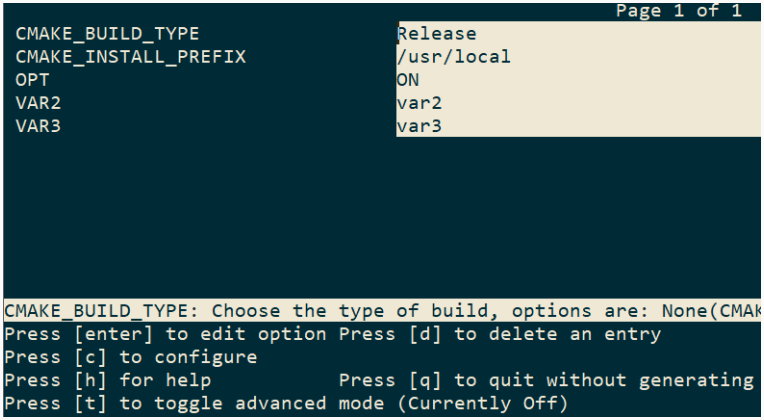
Cached variables can be reused across multiple runs, while *local variables* are only visible in a single run. Cached `FORCE` variables can be modified only after the initialization

```
project(my_project)

set(VAR1 "var1")                                    # local variable
set(VAR2 "var2" CACHE STRING "Description1")        # cached variable
set(VAR3 "var3" CACHE STRING "Description2" FORCE) # cached variable
option(OPT "This is an option" ON)                  # boolean cached variable
                                                    # same of var2
message(STATUS "${VAR1}, ${VAR2}, ${VAR3}, ${OPT}")
```

```
$ cmake .. # var1, var2, var3, ON
$ cmake -DVAR1=a -DVAR2=b -DVAR3=c -DOPT=d .. # var1, b, var3, d
```

Manage Cached Variables

```
$ ccmake .  # or 'cmake-gui'
```



Find Packages

```
project(my_project)                  # project name
cmake_minimum_required(VERSION 3.15) # minimum version

add_executable(program program.cpp)

find_package(Doxygen REQUIRED)  # compile only if Doxygen is found
find_package(Boost 1.87.0)      # search for a specific version

if (Boost_FOUND)
    target_include_directories("${PROJECT_SOURCE_DIR}/include" PUBLIC ${Boost_INCLUDE_DIRS})
else()
    message(FATAL_ERROR "Boost Lib not found")
endif()
```

Compile Commands

Generate JSON compilation database (`compile_commands.json`)

It contains the exact compiler calls for each file that are used by other tools

```
project(my_project)
cmake_minimum_required(VERSION 3.15)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)  # <--

add_executable(program program.cpp)
```

Change the C/C++ compiler:

```
CC=clang CXX=clang++ cmake ..
```

CTest is a testing tool (integrated in CMake) that can be used to automate updating, configuring, building, testing, performing memory checking, performing coverage

```
project(my_project)
cmake_minimum_required(VERSION 3.5)
add_executable(program program.cpp)

enable_testing()

add_test(NAME Test1         # check if "program" returns 0
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>) # command can be anything

add_test(NAME Test2         # check if "program" print "Correct"
         WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/build
         COMMAND ./program <args>)

set_tests_properties(Test2
                     PROPERTIES PASS_REGULAR_EXPRESSION "Correct")
```

Basic usage (call `ctest`):

```
$ make test       # run all tests
```

`ctest` usage:

```
$ ctest -R Python      # run all tests that contains 'Python' string
$ ctest -E Iron        # run all tests that not contain 'Iron' string
$ ctest -I 3,5         # run tests from 3 to 5
```

Each `ctest` command can be combined with other tools (e.g. `valgrind`)

`ctest` with Different Compile Options

It is possible to combine a custom target with `ctest` to compile the same code with different compile options

```
add_custom_target(program-compile
                  COMMAND mkdir -p test-release test-ubsan test-asan # create dirs
                  COMMAND cmake .. -B test-release                   # -B change working dir
                  COMMAND cmake .. -B test-ubsan   -DUBSAN=ON
                  COMMAND cmake .. -B test-asan    -DASAN=ON
                  COMMAND make -C test-release -j20 program          # -C run make in a
                  COMMAND make -C test-ubsan   -j20 program          # different dir
                  COMMAND make -C test-asan    -j20 program)

enable_testing()

add_test(NAME Program-Compile
         COMMAND make program-compile)
```

CMake Alternatives - xmake

xmake is a cross-platform build utility based on Lua.

Compared with makefile/CMakeLists.txt, the configuration syntax is more concise and intuitive. It is very friendly to novices and can quickly get started in a short time. Let users focus more on actual project development

Comparison: xmake vs cmake

Code Documentation

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources

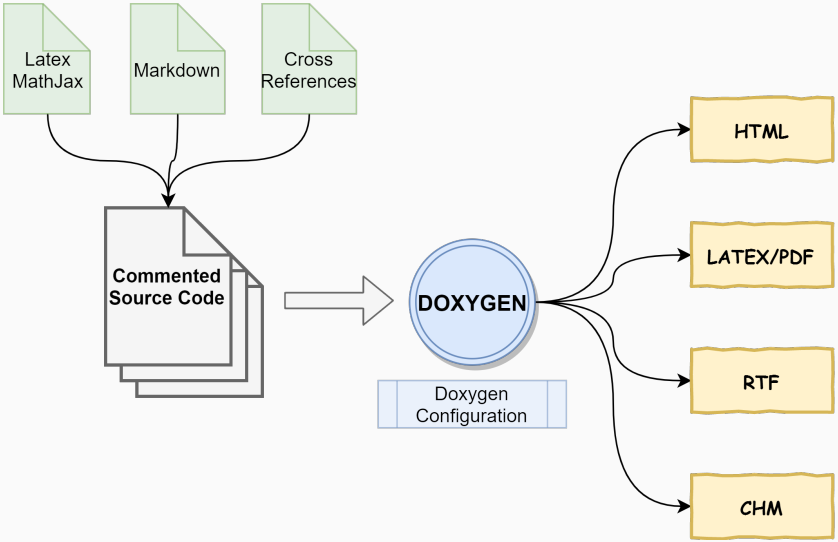
Doxygen usage

- comment the code with `///` or `/** comment */`
- generate `doxygen` base configuration file

```
$ doxygen -g
```

- modify the configuration file `Doxyfile`
- generate the documentation

```
$ doxygen <config_file>
```


Latex
MathJax
Markdown
Cross
References
Commented
Source Code
DOXYGEN
Doxygen
Configuration
HTML
LATEX/PDF
RTF
CHM

Doxygen requires the following tags for generating the documentation:

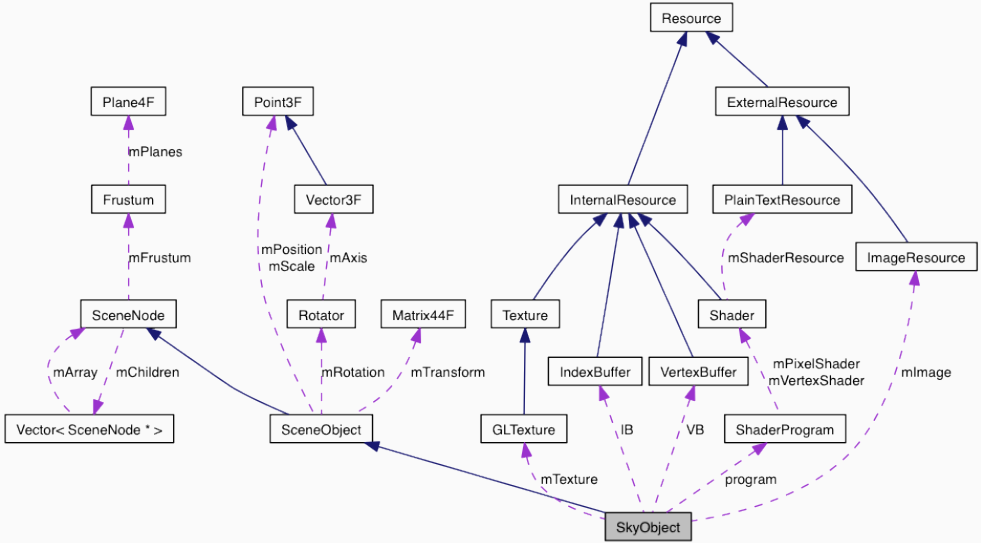
- `@file` Document a file
- `@brief` Brief description for an entity
- `@param` Run-time parameter description
- `@tparam` Template parameter description
- `@return` Return value description

- *Automatic cross references* between functions, variables, etc.
- *Specific highlight*. Code `` `<code>` ``, input/output parameters `@param[in] <param>`
- *Latex/MathJax* `$<code>$`
- *Markdown* (Markdown Cheatsheet link), Italic text `*<code>*`, bold text `**<code>**`, table, list, etc.
- Call/Hierarchy graph can be useful in large projects (requires `graphviz`)

```
HAVE_DOT = YES
GRAPHICAL_HIERARCHY = YES
CALL_GRAPH = YES
CALLER_GRAPH = YES
```

```
/**
 * @file
 * @copyright MyProject
 * license BSD3, Apache, MIT, etc.
 * @author MySelf
 * @version v3.14159265359
 * @date March, 2018
 */

/// @brief Namespace brief description
namespace my_namespace {

/// @brief "Class brief description"
/// @tparam R "Class template for"
template<typename R>
class A {
```

```
/**
 * @brief "What the function does?"
 * @details "Some additional details",
 *          Latex/MathJax: $\sqrt a$
 * @tparam T Type of input and output
 * @param[in] input Input array
 * @param[out] output Output array
 * @return `true` if correct,
 *         `false` otherwise
 * @remark it is *useful* if ...
 * @warning the behavior is **undefined** if
 *          @p input is `nullptr`
 * @see related_function
 */
template<typename T>
bool my_function(const T* input, T* output);

/// @brief
void related_function();
```


Resource
Plane4F
Point3F
ExternalResource
mPlanes
Frustum
Vector3F
InternalResource
PlainTextResource
mFrustum
mPosition
mScale
mAxis
mShaderResource
ImageResource
SceneNode
Rotator
Matrix44F
Texture
Shader
mArray
mChildren
mRotation
mTransform
IndexBuffer
VertexBuffer
mPixelShader
mVertexShader
mImage
Vector< SceneNode * >
SceneObject
GLTexture
IB
VB
ShaderProgram
mTexture
program
SkyObject

Doxygen Alternatives

`M.CSS`	Doxygen C++ theme
`Doxypress`	Doxygen fork
`clang-doc`	LLVM tool
`Sphinx`	`Clear, Functional C++ Documentation with Sphinx + Breathe + Doxygen + CMake`
`standardese`	The nextgen Doxygen for C++ (experimental)
`HDoc`	The modern documentation tool for C++ (alpha)
`Adobe Hyde`	Utility to facilitate documenting C++

Code Statistics

Count Lines of Code - `cloc`

`cloc` counts blank lines, comment lines, and physical lines of source code in many programming languages

```
$cloc my_project/

4076 text files.
3883 unique files.
1521 files ignored.

http://cloc.sourceforge.net v 1.50  T=12.0 s (209.2 files/s, 70472.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                              135          18718          22862         140483
C/C++ Header                   147           7650          12093          44042
Bourne Shell                   116           3402           5789          36882
```

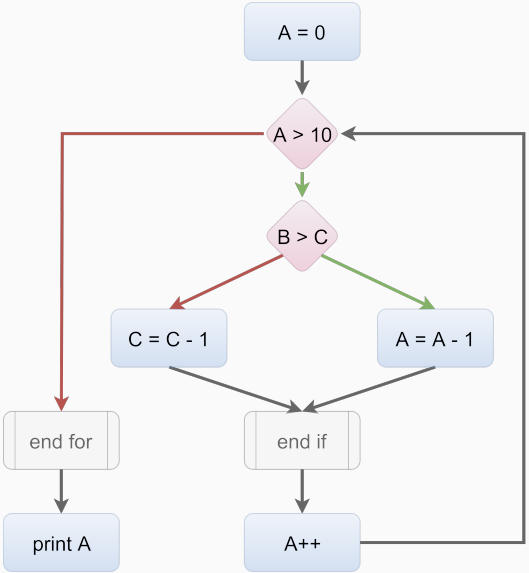
Features: filter by-file/language, SQL database, archive support, line count diff, etc.

Lizard is an extensible Cyclomatic Complexity Analyzer for many programming languages including C/C++

Cyclomatic Complexity: is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program source code

```
$lizard my_project/
================================================================
NLOC    CCN  token  param    function@line@file
----------------------------------------------------------------
10      2     29      2    start_new_player@26@./html_game.c
6       1      3      0    set_shutdown_flag@449@./httpd.c
24      3     61      1    server_main@454@./httpd.c
----------------------------------------------------------------
```

- `CCN`: cyclomatic complexity (should not exceed a threshold)
- `NLOC`: lines of code without comments
- `token`: Number of conditional statements



CCN = 3

CC	Risk Evaluation
1-10	a simple program, *without much risk*
11-20	more complex, *moderate risk*
21-50	complex, *high risk*
> 50	untestable program, *very high risk*

CC	Guidelines
1-5	The routine is probably fine
6-10	Start to think about ways to simplify the routine
> 10	Break part of the routine

Risk: `Lizard`: 15, `OCLint`: 10

- `www.microsoftpressstore.com/store/code-complete-9780735619678`
- `blog.feabhas.com/2018/07/code-quality-cyclomatic-complexity`

Other Tools

Code Formatting - clang-format

clang-format is a tool to automatically format C/C++ code (and other languages)

```
$ clang-format <file/directory>
```

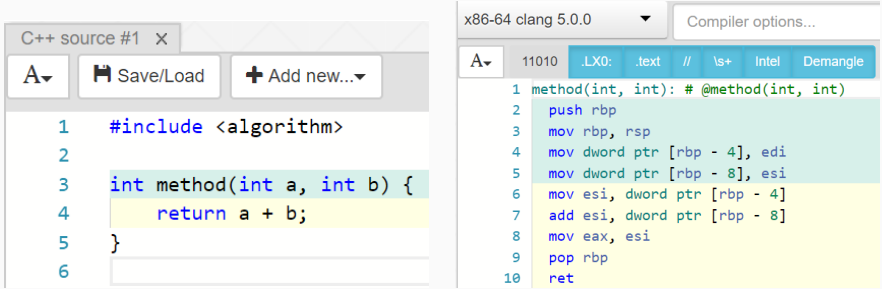
clang-format searches the configuration file .clang-format file located in the closest parent directory of the input file

clang-format example:

```
IndentWidth: 4
UseTab: Never
BreakBeforeBraces: Linux
ColumnLimit: 80
SortIncludes: true
```

Compiler Explorer (assembly and execution)

`Compiler Explorer` is an interactive tool that lets you type source code and see assembly output, control flow graph, optimization hint, etc.



Key features: support multiple architectures and compilers

Code Transformation - CppInsights

CppInsights See what your compiler does behind the scenes

AI-Powered Code Completion tools help writing code faster by drawing context from comments and code to suggest individual lines and whole functions

Common features:

- Semantic completion
- Recognize common language patterns
- Use the documentation to infer this function name, return type, and arguments
- Suggest bug fixes
- Generate comments, documentation, and even Pull Request text

They are commonly provided as plug-in for the most popular editors and IDE

- `CoPilot`
- `TabNine`
- `Codeium`
- `Replit Ghostwriter`
- `CodeWhisperer`

Local Code Search - `ugrep`, `ripgrep`, `hypergrep`

`ugrep`, `Ripgrep`, `Hypergrep` are code-searching-oriented tools for regex pattern

Features:

- Default recursively searches
- Skip `.gitignore` patterns, binary and hidden files/directories
- Windows, Linux, Mac OS support
- Up to 100x faster than `GNU grep`

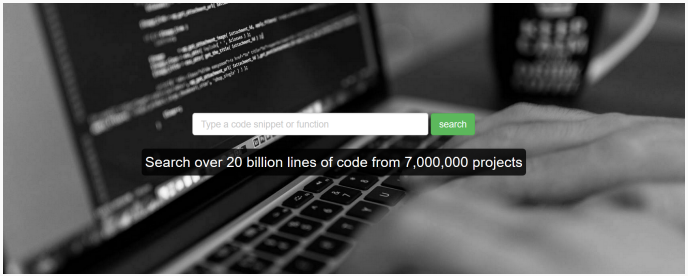
```
[andrew@Cheetah rust] rg -i rustacean
src/doc/book/nightly-rust.md
92:[Mibbit][mibbit]. Click that link, and you'll be chatting with other Rustaceans

src/doc/book/glossary.md
3:Not every Rustacean has a background in systems programming, nor in computer

src/doc/book/getting-started.md
176:Rustaceans (a silly nickname we call ourselves) who can help us out. Other great
376:Cargo is Rust's build system and package manager, and Rustaceans use Cargo to

src/doc/book/guessing-game.md
444:it really easy to re-use libraries, and so Rustaceans tend to write smaller

CONTRIBUTING.md
322:* [rustaceans.org][ro] is helpful, but mostly dedicated to IRC
333:[ro]: http://www.rustaceans.org/
[andrew@Cheetah rust]
```

Code Search Engine - `searchcode`

`Searchcode` is a free source code search engine

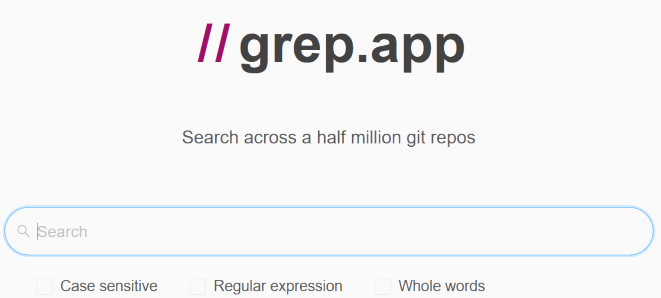
Features:

- Search over 20 billion lines of code from 7,000,000 projects
- Search sources: `github`, `bitbucket`, `gitlab`, `google code`, `sourceforge`, etc.



Code Search Engine - `grep.app`

`grep.app` searches across a half million GitHub repos



Code Benchmarking - Quick-Bench

Quick-benchmark is a micro benchmarking tool intended to quickly and simply compare the performances of two or more code snippets. The benchmark runs on a pool of AWS machines

Font for Coding

Many editors allow adding optimized fonts for programming which improve legibility and provide extra symbols (ligatures)

Scope	→ ⇒ :: __	-> => :: __
Equality	= ≡ ≠ ≠ == === ≠ ≠=	== === != =/= == === != !==
Comparisons	⩽ ⩾ ≤ ≥ ⟺	<= >= <= >= <=>

Some examples:

- JetBrain Mono
- Fira Code
- Microsoft Cascadia
- Consolas Ligaturized