CMake Presets

Sven Johannsen
Bretgeld Engineering
s.johannsen@bretgeld-engineering.de

TL;DR

- CMake Presets are machine-readable files (`.json`)
- stored next to `CMakeLists.txt`
- help IDEs work with CMake
- may also be (limited) helpful for command-line, Vim, Emacs users
- This presentation shows command-line usage to avoid IDE-specific details

(IDE-specific problems with CMake and CMake Presets are out of scope.)

Content

- calling CMake from the command line (simple use)
- CMake customization
  - generators
  - compiler / toolsets
  - options
- combinations of options
  - build and tests
- the CMake & IDE problem (before presets)
- (CMake-) Presets
- calling CMake from the command line
- (CMake-) Presets details
- IDE Demo
- Pitfalls

Outdated CMake Usage

How to build a C++ project with CMake?

README.md of a random older open-source project

```
# outdated!
mkdir build
cd build
cmake ..
make
ctest
```

A More Modern Approach

Calling CMake in the root folder:

```
cmake -S . -B ./build/
cmake --build ./build/
ctest --test-dir ./build/
```

See projects like SDL2 and others.

No functional changes, just different syntax.

- **Outdated:** Go to the build folder and call `make`.
- **Modern:** Pass the build folder as an argument; don't change folders.

Benefits:

- Clear separation of source, build, and configuration.
- Abstract the build command; independent of the generator.

However, the example shows a simplified, "hello world" view of CMake. Real projects need more parameters.

Generator

CMake is a meta build tool. It generates configurations for different versions of Make, Ninja, Visual Studio, and others.

CMake supports "single config" and "multi config" generators (Release and Debug are part of the configuration, or part of the build step).

```
cmake --help          # list all supported generators
cmake -G "Ninja" ...
```

Compilers

Select the compilers for the build:

```
export CC=gcc-14
export CXX=g++-14
cmake -S . -B build-gcc14
```

```
cmake -S. -B build-gcc14 -DCMAKE_C_COMPILER=gcc-14 -DCMAKE_CXX_COMPILER=g++-14
```

One C/C++/Fortran/Assembler compiler per CMake project.

Visual Studio: toolsets (`-T`) and Platforms (`-A`)

The IDE Problem

Most IDEs start the (CMake) configuration directly after selecting the folder / `CMakeLists.txt` file:

- some settings must be right in the first CMake run (generator, compiler).

Using a different generator, compiler (or build folder) requires reconfiguration of the project and usually command-line actions.

But there are more settings.

CMake Options

Settings (options) from the Google Benchmark project:

```
option(BENCHMARK_ENABLE_TESTING "Enable testing of the benchmark library." ON)
option(BENCHMARK_ENABLE_EXCEPTIONS "Enable the use of exceptions in the benchmark library." ON)
option(BENCHMARK_ENABLE_LTO "Enable link time optimisation of the benchmark library." OFF)
option(BENCHMARK_USE_LIBCXX "Build and test using libc++ as the standard library." OFF)
```

in a build folder:

```
cmake .. -DBENCHMARK_ENABLE_LTO=ON
```

After one month: What options are enabled in this build folder?

Hint: `CMakeCache.txt`, but it's hard to interpret. (Or use `cmake-gui`.)

CMake Variables

Example: `CMAKE_BUILD_TYPE` (for Makefiles or Ninja)

- Debug
- Release
- RelWithDebInfo
- MinSizeRel

```
cmake -DCMAKE_BUILD_TYPE=Release ...
```

See also: https://cmake.org/cmake/help/latest/manual/cmake-variables.7.html

Combination of options

Usually some combinations are related:

- LTO only for release builds
- libc++ only for Clang compiler

Even if an IDE supports options and CMake variables (unlikely), there is no help to organize combinations of options.

CMake Command-Line Parameters

- Configuration
- Build
- CTest

Customization (Configuration)

```
cmake -S . -B ./build-debug/ ...
```

- CMake-specific
  - `-G "Ninja"`: Generator
  - `-D...=...` settings:
    - `-DCMAKE_BUILD_TYPE=Release` (or Debug, Release, etc.)
    - `-DBENCHMARK_ENABLE_LTO=ON` (enable LTO for Google Benchmark)
- Platforms (`-A`)
  - Visual Studio: Win32, x64, ARM, ARM64
- Toolsets (`-T`)
  - Visual Studio: clang
  - XCode, ...
- Toolchains
  - cross-platform build
    - `--toolchain path/to/file`
    - `-DCMAKE_TOOLCHAIN_FILE=path/to/file`

Customization (Build)

`cmake --build ./build-debug/ ...`

- **Build:**
  - `--config <cfg>` (for multi-configuration generators like Visual Studio)
  - `-t <tgt>`, `--target <tgt>` (build a specific target)
  - `-j <N>` (number of parallel jobs; for make, this enables parallel builds; for Ninja, `-j 1` results in a serial build)

Platform-independent builds

```
cmake --build ./build/
```

Works only for very simple projects.

- Make: specify the number of jobs
- Ninja: specifying the number of jobs is not recommended
- Visual Studio needs a configuration (Debug/Release/...)

Customization (Test)

- Test:
  - `--output-on-failure` (show output for failed tests)
  - `--output-junit <file>` (specify output file in JUnit XML format)

Customize CMake from an IDE

How to open a CMake project / folder and modify the CMake parameters?

- **Visual Studio**: Open a folder with `CMakeLists.txt` and Visual Studio will run CMake with default parameters.
- **VS Code**: Open a folder with `CMakeLists.txt` and VS Code will ask you about using different compilers.

How to choose your generator? How to add all the `-D` parameters.

(Most IDEs support `CMAKE_BUILD_TYPE` with Release / Debug, but have limited support for options.)

There are workarounds, but they are still workarounds.

CMake Presets

CMake Presets help IDEs call `cmake`.

- `CMakeLists.txt`: "How to call the compiler?"
- `CMakePresets.json`: "How to call CMake?"

What are CMake Presets:

- JSON: machine-readable
- Supported by major IDEs (e.g., CLion, VS Code, Visual Studio, QtCreator, many others)
- Project-specific presets: `CMakePresets.json` (checked into version control)
- User presets: `CMakeUserPresets.json` (not checked into version control)
- Like a set of CMake command-line parameters.
- Usage is optional (an IDE might automatically pick the presets).

Structure

Files:

- `CMakePresets.json` (checked into version control)
- `CMakeUserPresets.json` (not checked into version control)
- (and JSON includes - not covered in this presentation)
- next to the top-level `CMakeLists.txt`

JSON

- cmake-presets(7)
  - Complete documentation.
  - Only one example provided, but it's comprehensive.
  - Example provides limited guidance for structuring presets.
- Version information
- Configuration: "configurePresets"
- Build: "buildPresets"
- Test: "testPresets"
- (Package: "packagePresets" - not covered)
- (Workflow: "workflowPresets" - not covered)

IDEs might support a superset. (e.g., CLion and QtCreator support configuration and build presets.)

Example file

```
{
  "version": 3,
  "configurePresets": [
    {
        "name": "debug-make",
        "generator": "Unix Makefiles",
        "binaryDir": "${sourceDir}/build-presets/${presetName}",
        "cacheVariables":
        {
            "CMAKE_BUILD_TYPE": "Debug"
        }
    }
  ]
}
```

New Names for Existing Parameters

JSON Property	CMake Parameter
`"name"`	-
`"displayName"`	-
`"description"`	-
`"generator"`	-G
`"binaryDir"`	-B
`"installDir"`	--install-prefix
`"architecture"`	-A
`"toolset"`	-T
`"toolchainFile"`	--toolchain
`"cacheVariables"`	-Dvar=value
`"environment"`	set environment
`"source"`	--trace-source

Preset JSON properties are more uniform than CMake parameters.

Versioning

```
{
  "version": 10,
  "cmakeMinimumRequired": {
    "major": 3, "minor": 23, "patch": 0
  },
  ...
}
```

"cmakeMinimumRequired" is redundant information.

Preset files are schema-checked. If a property is not part of the schema, the preset file is invalid!

Preset Schema version	CMake Version
1	3.19
4	3.23
...	...
8	3.28
10	3.31 (add comments)

Command-Line Usage

(CMake Presets improve CMake usage in IDEs, but using them on the command line provides a better understanding.)

Go to the root of your project (the same folder as the top-level `CMakeLists.txt`):

```
# configuration
cmake --list-presets
cmake --preset myConfiguration

# build
cmake --build --list-presets
cmake --build --preset myBuildSpecification

# test
ctest --list-presets
ctest --preset mySpecialTestSpec
```

Demo

Part IV Pitfalls

- Syntax errors
- Build folder
- Grows very fast
- `"inherits"` is very helpful for simplification, but not well documented
- Platform-dependent conditions work only on leaf nodes

Syntax errors

IDEs

- No presets found
- CMakePresets.json is invalid

CMake command-line

```
> cmake --list-presets
CMake Error: Could not read presets from /home/sven/src/a83:
CMakePresets.json:24: Missing ',' or ']' in array declaration
    {
    ^
CMakePresets.json:55: Extra non-whitespace after JSON value.
    },
     ^
```

Build folder

IDEs have safe fallbacks for the build folder.

Using presets on the command line without a build folder will mess up the source folder!

Easy to add a build folder: `"binaryDir"`

```
{
    "name": "base",
    "generator": "Ninja",
    "hidden": true,
    "binaryDir": "${sourceDir}/build-presets/${presetName}"
}
```

Macros

- `${sourceDir}`,
- `${presetName}`,

Other

- `${sourceParentDir}`: sourceDir without sourceDirName
- `${sourceDirName}`,
- `${generator}`,
- `${hostSystemName}`,
- `${fileDir}`,
- `${dollar}`,
- `${pathListSep}`,
- `$env{<variable-name>}`,
- `$penv{<variable-name>}`

Preset files grow fast

Keep presets simple!

- Every configuration needs a build folder,
- Every configuration needs a generator,
- ...

Use simple basic blocks and a few final blocks, and the `inherits` property.

"inherits" Example

Simple blocks, hidden (not visible in the IDE)

```
    "configurePresets": [
    {
      "name": "ninja",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build-presets/${presetName}"
    },
    {
      "name": "debug",
      "hidden": true,
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug"
      }
    },
    {
      "name": "ASAN",
      "hidden": true,
      "cacheVariables": {
        "MYCOMPANY_ASAN": "ON"
      }
```

"inherits" Example

Use `"inherits"`, not hidden (selectable in the IDE)

```
...
{
  "name": "gcc-14-debug",
  "inherits": [
    "ninja",
    "debug",
    "gcc-14"
  ]
},
{
  "name": "gcc-14-debug-ASAN",
  "inherits": [
    "gcc-14-debug",
    "ASAN"
  ]
}
```

Even with longer files, it's more maintainable.

"inherits" Is Very Helpful for Simplification, but Not Well Documented

- Pure JSON property replacement
- Order of replacement is not documented
- Do not inherit from blocks with the same properties!

Demo 2

Platform-Dependent Conditions

Limit the valid preset current system

```
{ ...
    {
        "name": "clang-18",
        "hidden": true,
        "cacheVariables": { "CMAKE_C_COMPILER": "clang-18", "CMAKE_CXX_COMPILER": "clang++-18" },
        "condition": { "type": "equals", "lhs": "${hostSystemName}", "rhs": "Linux"}
    },
{
    "name": "clang-18-release",
    "displayName": "clang-18 (Release)",
    "inherits": [
        "release",
        "clang-18",
        "ninja"
    ]
}
```

(My specific clang configuration works only on Linux and is not compatible with clang in MSVC.)

From https://github.com/microsoft/vscode-cmake-tools/blob/HEAD/docs/cmake-presets.md#enable-vcpkg-integration

Enable Vcpkg integration

Vcpkg helps you manage C and C++ libraries on Windows, Linux, and macOS. A vcpkg toolchain file (vcpkg.cmake) must be passed to CMake to enable vcpkg integration. For more information, see the vcpkg documentation.

Set the path to vcpkg.cmake with the VCPKG_ROOT environment variable in CMakePresets.json:

```
"cacheVariables": {
   "CMAKE_TOOLCHAIN_FILE": {
      "value": "$env{VCPKG_ROOT}/scripts/buildsystems/vcpkg.cmake",
       "type": "FILEPATH"
    }
 },
```

VCPKG_ROOT should be set to the root of your vcpkg installation. For more information, see vcpkg environment variables.

...

Benefits of Presets for Your Projects

- Machine-readable build instructions: IDEs learn how to call CMake.
- Verify your build instructions in CI pipelines.
- Potentially more useful for IDE users than for Vim/Emacs or command-line users.
- Potentially more useful for application developers than for library developers.
- Simplify your `.gitignore` for build folders (e.g., CLion, QtCreator).

Pick Your Preferred CMake Workflow (Examples)

- Use configuration presets
- Ignore build presets and call Make/Ninja on the command line
- …

Not Convinced?

No problem—presets are optional.

I have just one favor:

- Please add `CMakeUserPresets.json` to your `.gitignore` file.

```
...
CMakeUserPresets.json
...
```

Links

- CMake: cmake-presets(7):
  https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html
- QtCreator:
  https://doc.qt.io/qtcreator/creator-build-settings-cmake-presets.html
- VS Code:
  https://github.com/microsoft/vscode-cmake-tools/blob/HEAD/docs/cmake-presets.md
- Visual Studio:
  https://learn.microsoft.com/en-us/cpp/build/cmake-presets-vs?view=msvc-170
- CLion:
  https://www.jetbrains.com/help/clion/cmake-presets.html

Any questions

a.k.a. end of the presentation