Introduction to **CMake** in 30 Minutes

Andrey Upadyshev

Version 2018-05-30

What Is CMake?

- You know build systems: Make, Xcode, Visual Studio/ MSBuild etc
- CMake is a **meta build system**:
  - Generates project files for chosen build system
  - Invokes the build system (`cmake --build`)
- Supports C, C++, Fortran, asm, CUDA and custom targets
- Written in C++, supported on all major platforms
- Supports cross-compiling

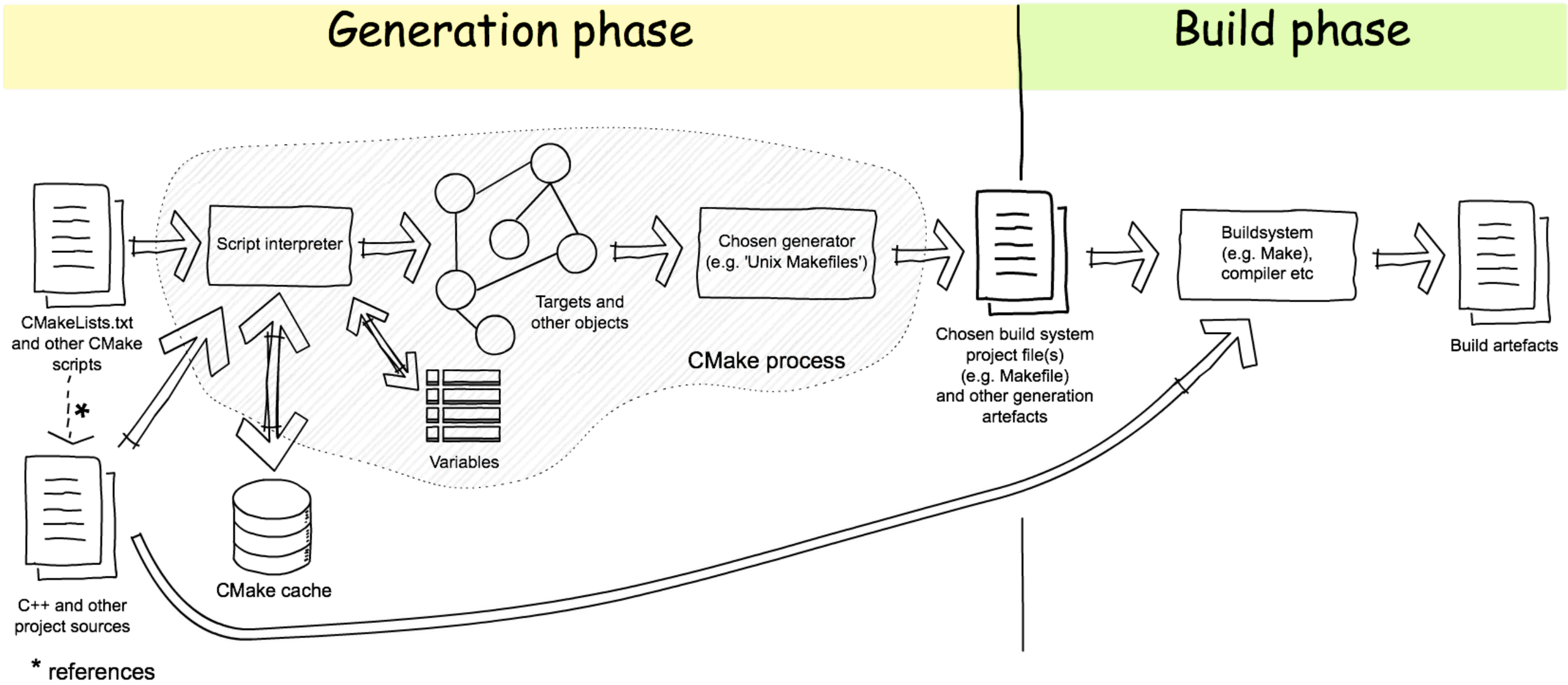
CMake is more than building

- Run and report tests (`ctest`)
- Create archives, DEBs, RPMs, MSIs etc from build artefacts (`cpack`)
- Download and build dependencies (`ExternalProject` module)
- Cross-platform scripting aka script mode (`cmake -P <script>`)
- Cross-platform shell commands aka command mode (`cmake -E <command> <arg>...`)

Data Model

- Variables, including environment ones
- Variables cache (the cache) - **per-project** cache to persist some variables between CMake invocations
- Objects:
  - Target; Directory; Source file; Test; Installed file; Cache entry (object that holds a cached variable); (the) Global scope
- Properties (of objects)
- Data types:
  - String
  - List (of strings). Called "*;-list*" because it's just a semicolon separated string.

Execution Model



Standard Workflow

- Generate project files with
  `cmake ...`
- Build project with
  `cmake --build ...` (or `make`)
- Run unit-tests with
  `ctest ...` (or `make test`)
- Install package with
  `cmake --build ... --target install`
  (or `make install`) ...
- ... or pack artefacts for uploading with `cpack ...`

Live Session 1: Hello CMake!

- Minimal CMake project which builds a single executable **uno**
  that depends on Boost library

```
# CMakeLists.txt:
cmake_minimum_required(VERSION 3.5)

project(Uno LANGUAGES CXX VERSION 1.0)

set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 14)

find_package(Boost 1.65 REQUIRED)

enable_testing()  # Enable test target

# Define executable target uno
add_executable(uno uno.cpp)
target_link_libraries(uno PRIVATE Boost::boost)
```

```
.
├── CMakeLists.txt
└── uno.cpp

$ mkdir build && cd build
$ cmake ..

build
├── Makefile
├── CMakeCache.txt
└── ...

$ cmake --build .
# or
$ make

build
├── uno
└── ...
```

CMake Language

- Procedural language to manipulate CMake Data Model
- Control structures (`if, foreach, while`)
- Functions
  - Built-in (implemented inside CMake, called *commands*)
  - User-defined
- Macros (a function that's executed in parent scope)
- Modules

Commands

- Tons of commands to work with strings, lists, files, paths, do math and much more
- Commands to create objects, e.g.:
  - `add_executable(...)`
  - `add_library(...)`
  - `add_test(...)`
- Commands to manipulate object properties **explicitly** or **implicitly**, e.g.:
  - `get_property`
  - `target_link_libraries`

Variables

- Scope:
  - Directory (visible in all subdirectories as well)
    - Root directory variables are effectively global
  - Function
  - Cached variables are global

- Many variables are used by CMake itself. CMake usually set them on startup and later read them when fill object properties on object creation
  - Information about platform, compiler, tools and project itself.
    E.g. `APPLE`, `MSVC_VERSION`, `CMAKE_HOST_UNIX`, `CMAKE_COMMAND`, `CMAKE_SHARED_LIBRARY_PREFIX`, `CMAKE_MAJOR_VERSION`, `PROJECT_NAME`
  - Control the project generation and build.
    E.g. `CMAKE_CXX_STANDARD`, `CMAKE_INSTALL_PREFIX`, `CMAKE_CXX_FLAGS`, `CMAKE_BUILD_TYPE`, `LIBRARY_OUTPUT_PATH`

Objects & Properties

- That's where your project is defined: generator generates project files out of objects and properties that you set in your CMake scripts.
- When object is created CMake populates its properties with meaningful defaults (exact values depends on the platform and generator and controlled via variables)
- Usually you don't directly manipulate properties but rather use higher-level functions that manage properties for you. E.g. `target_include_directories()` command sets `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES` target properties.

Targets

- "Normal" (buildable):
  - Executable
  - Static/Shared/Header-only library
  - Alias (to another target)
    - Imported - one that's built and exported by another project
- Custom target
- Pseudo-targets `test` and `install`
- When makes sense, target names can be passed to commands instead of library or executable name/path, e.g. `add_test(), target_link_libraries()`

Source & Binary Directories

- **Source dirs** are where the projects's `CMakeLists.txt` resides. Starts with the root source dir where the main `CMakeLists.txt` resides.
- **Binary dir** (often called *build dir*) is where the project files are generated (that's the dir where cmake was run) and `CMakeCache.txt` resides. Usually all the build artefacts are placed somewhere under this directory as well.
- **The best practice is to do out-of-source build** so the source and binary directories are different.
- Relative source ("input") paths given in `CMakeLists.txt` are relative to current source dir. Relative binary (output) paths are relative to current binary dir.
- Binary tree follows the source tree:

```
<src>
├── CMakeLists.txt
├── motu
│   ├── CMakeLists.txt
│   ├── include
│   │   └── ...
│   ├── src
│   │   ├── CMakeLists.txt
│   │   └── ...
│   └── test
│       ├── CMakeLists.txt
│       ├── logging
│       │   ├── CMakeLists.txt
│       │   └── ...
│       └ ...
└── src
    ├── CMakeLists.txt
    └── ...
```

```
<build>
├── CMakeCache.txt
├── SomeService.xcodeproj
├── Debug
│   └── libmodel.a
├── motu
│   ├── src
│   │   └── Debug
│   │       └── libmotu-runtime.dylib
│   └── test
│       ├── Debug
│       │   └── test-service-job
│       ├── logging
│       │   └── Debug
│       │       └── test-logging
│       └── ...
└── src
    └── Debug
        └── some-service
```

Target Dependencies

- `target_link_libraries(<target_name> <visibility> <dependency>...)`
  - Used for buildable targets
  - Dependency can be either a target name or a library name or a full library path
  - Command sets correct build order, link libraries, include directories, compiler flags, compiler definitions for you
- `add_dependencies(<target_name> <dependency>...)`
  - Used for custom targets
  - Only affects build order
- Target dependencies are transitive

Visibility Keywords

- `PRIVATE` - target's internal
- `INTERFACE` - target's interface only
- `PUBLIC` = `INTERFACE + PRIVATE`
- Used in many commands, e.g.:
  - `target_link_libraries()`, `target_include_directories()`, `target_compile_definitions()`, `target_compile_features()`, `target_compile_options()`, `target_sources()`
- Example:

```
add_library(mywget mywget.h mywget.cpp)
target_link_libraries(mywget
    PRIVATE curl fmt
    PUBLIC Boost::boost
)
```

- Do not use absolute paths in interface dependencies. This makes your package non-relocatable.

Build Configurations

- CMake supports multiple build configurations
- Default CMake configurations:
  - Debug
  - Release
  - RelWithDebInfo
  - MinSizeRel
- Can be changed

Single-Configuration Build Systems

- Some build systems (e.g. Make) are **single-configuration**: the generated project has a single configuration e.g. **either** Debug **or** Release.

- Configuration is chosen at generation time and then fixed:
  - ```
    cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
    cmake --build .
    ctest
    ```

- Configuration is stored in `${CMAKE_BUILD_TYPE}`
    ```

Multi-Configuration Build Systems

- Some build systems (e,g. Xcode and Visual Studio) are **multi-configuration**: the generated project contains multiple configurations e.g. **both** Debug **and** Release.

- Configuration is chosen at build time:
  - ```
    cmake ..
    cmake --build . --config RelWithDebInfo
    ctest -C RelWithDebInfo
    ```

- There are per-configuration variables and properties that preferred to their configuration-agnostic counterparts.
  - E.g. when present, `CMAKE_LIBRARY_OUTPUT_DIRECTORY_<CONFIG>` (where `<CONFIG>` is uppercase configuration name) is used instead of `CMAKE_LIBRARY_OUTPUT_DIRECTORY`
    ```

Generator Expressions

- What if object property need to have different value in different build configurations or be different when package is installed?
- This is fine. There is generator expressions for this.
- Generator expressions allows to specify property as a function so the final value is evaluated during the generation phase.
- Special syntax: `$<...>` and the whole mini-language with conditions etc
- Example:

```
target_include_directories(mylib PUBLIC
    "$<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/../include>"
    "$<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>")
```

- Example efines two values:
  - First is *only* set when target is consumed from build directory
  - Second is *only* set when target is consumed from install location
- Generators accept generator expression for many properties but not for all of them so always check documentation. It's going better with each CMake release.

Using Packages

- Command
  `find_package(<package_name> [version]`
  `[MODULE|CONFIG] [REQUIRED] ...)`
- Finds specified package installed on local system and loads exported targets from it.
- Crappy old packages offer variables with include path and link library location to be used instead of exported targets.
- Two modes to search packages: module or config. If not specified then it's module if there is a find module otherwise config.
- Module mode executes module `Find<package_name>.cmake` (called *finder*)
- Config mode searches for `<package_name>Config.cmake` or `<lower-case-package_name>-config.cmake` under particular locations
- Proper CMake-based packages installs package config to allow CMake to find them
- Module mode is usually used only for packages that aren't CMake-based

Mastering Packages

- Too big for this talk. Sorry.

Troubleshooting

- Check `<build_dir>/CMakeCache.txt`
- Check generated project and other files
- Add logging: `message(STATUS "var=${var}")`
- Add hard stops: `message(FATAL_ERROR "ouch")`
- Add `if()` around to make an assert.
- Know and use CMake's diagnostic command-line options `-Wdev`, `--warn-uninitialized`, `--warn-unused-vars`, `--debug-output`, `--trace` & co.

Further Reading

- https://cmake.org/cmake/help/latest/index.html
- https://cmake.org/cmake/help/latest/manual/cmake-packages.7.html
- Daniel Pfeifer. "Effective CMake" (C++Now 2017)
  https://www.youtube.com/watch?v=bsXLMQ6WgIk
- Mathieu Ropert "Modern CMake for modular design" (CppCon 2017)
  https://www.youtube.com/watch?v=eC9-iRN2b04
- Rico Huijbers "The Ultimate Guide to Modern CMake"
  https://rix0r.nl/blog/2015/08/13/cmake-guide/
- Jussi Pakkanen. "A list of common CMake antipatterns"
  http://voices.canonical.com/jussi.pakkanen/2013/03/26/a-list-of-common-cmake-antipatterns/
- `$ man cmake`
- CMake standard modules sources (`/usr/local/share/cmake/Modules` on mac)

Questions?

Thank you!