Introduction to modern CMake

Oxford Research Software Engineering

Fergus Cooper ~ Graham Lee ~ Thibault Lestang

Problem Statement

You want your C++ code to compile on other computers, not just your laptop.

- group workstation
- HPC compile node
- collaborator laptops

Everyone should end up with a program that behaves the same way, wherever they build.

Enter CMake

You describe *targets* (what to build), *inputs* (the sources files), and *configuration* (what libraries to use, what compiler settings, etc.).

CMake uses that with its own *rules* (how to turn sources into programs) to generate makefiles, IDE projects, or other outputs. CMake doesn't build your project itself!

CMake works on Linux, Windows, macOS and more.

Getting Started

Checkpoint 0 is a simple "hello, world" program written in C++. Let's use CMake to build it.

```
$ cd checkpoint_0
```

`CMakeLists.txt`

This is where you write the definitions of your targets and configuration.

Let's look at the sample `CMakeLists.txt` line by line.

CMake has changed a lot

```
cmake_minimum_required(VERSION 3.13)
```

Tells CMake which version we used, affecting the features available and the interpretation of `CMakeLists.txt`

Define a project

```
project(IntroCMakeCourse LANGUAGES CXX)
```

We have a project called `IntroCMakeCourse`, in the C++ language.

Configure the compiler

```
set(CMAKE_CXX_STANDARD 17)
```

We're using the C++17 language dialect.

Tell it what to build

```
add_executable(main_executable main.cpp)
```

There is a program, called `main_executable`, which depends on the source code in `main.cpp`

Using CMake

It's typical to build "out of tree", by running CMake in a separate place. Keeps generated files out of your source folder.

```
checkpoint0$ mkdir build
checkpoint0$ cd build
build$ cmake ..
[...]
-- Build files have been written to: <...>/checkpoint_0/build
```

Build your project

CMake only generated the build script, it didn't actually compile anything.

```
build$ make
[...]
[100%] Built target main_executable
build$ ./main_executable
Checkpoint 0
Hello, World!
```

Breakout time

Verify that we can all configure, compile and run the executable in Checkpoint 0.

Choosing a generator

CMake can create more than Makefiles. It can generate IDE projects, or build descriptions for the fast Ninja tool.

```
build$ cmake -G Ninja ..
[...]

build$ ninja
[2/2] Linking CXX executable main_executable
```

Choosing a generator

You can build uniformly, regardless of the generator:

```
build$ cmake -G Ninja ..
build$ cmake --build . --target main_executable
```

This can be particularly useful in automated scripts that may be run on different systems using different generators.

Setting configuration

You (and users) can override choices made by CMake using the `-D` argument.

```
build$ cmake -DCMAKE_CXX_COMPILER=/usr/local/bin/g++-10 ..
-- Configuring done
You have changed variables that require your cache to be deleted.
Configure will be re-run and you may have to reset some variables.
The following variables have changed:
CMAKE_CXX_COMPILER= /usr/local/bin/g++-10

-- The CXX compiler identification is GNU 10.2.0
[...]
```

Setting configuration

You can switch between Debug, Release, RelWithDebInfo and MinSizeRel, by default:

```
build$ cmake -DCMAKE_BUILD_TYPE=Release ..
[...]
```

The default flags with g++ are:

```
CMAKE_CXX_FLAGS_DEBUG            -g
CMAKE_CXX_FLAGS_MINSIZEREL       -Os -DNDEBUG
CMAKE_CXX_FLAGS_RELEASE          -O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO   -O2 -g -DNDEBUG
```

Breakout time

Try using the Ninja generator, compiling in Release mode, and using another compiler if you have one installed.

Remember that you might have to clean your build directory when, e.g., changing generator.

Adding subdirectories

```
CMakeLists.txt/
src/
    CMakeLists.txt
    functionality.cpp
    functionality.hpp
    main.cpp
```

In the top-level `CMakeLists.txt`:

```
add_subdirectory(src)
```

CMake processes the `CMakeLists.txt` file in the directory `src`.

Compartmentalising build logic

```
# src/CMakeLists.txt
set(src_source_files file1 file2 file3)
add_executable(executable ${src_source_files})
```

Variables defined in the upper scope are available in the lower scope, but not the other way around.

Using subdirectories enables clear structure and modularity, and keeps the top-level `CMakeLists.txt` clean and tidy.

Programming CMake

Variables can hold lists:

```
set( src_files main.cpp functionality.cpp functionality.hpp )
```

Variables can be dereferenced

```
set(another_list ${src_files})
```

The value of `another_list` is set to the value of `src_files`.

Nested example:

```
set(var files) # var = "files"
set(yet_another_list ${src_${var}})
```

Checkpoint 1

Our project has grown! In addition to the code in `main.cpp`, some new functionality was added to new source files `functionality.cpp` and `functionality.hpp`.

This code is now contained in a specific directory `src/`, inside the project directory.

Breakout time

Look through the files in Checkpoint 1.

Add a new pair of hpp/cpp files that defines a new function.

- Call it from the main executable
- Add the files to `src/CMakeLists.txt`
- Configure, compile and run: check that your new function has been executed

Target properties

CMake allows for a very fine-grained control of target builds, through *properties*.

For example, the property `INCLUDE_DIRECTORIES` specifies the list of directories to be specified with the compiler switch `-I` (or `/I`).

Properties can be set manually like variables, but in general CMake provides commands for it:

```
target_include_directories(main_executable
                           PUBLIC
                           ${CMAKE_CURRENT_SOURCE_DIR}
)
```

Properties are different from variables!

Creating a library

Similar to `add_executable()`:

```
add_library(my_lib STATIC ${source_files})
```

Use `SHARED` instead of `STATIC` to build a shared library: or, if omitted, CMake will pick a default based on the value of the variable `BUILD_SHARED_LIBS`.

Linking libraries (`PRIVATE`)

Library dependencies can be declared using the `target_link_libraries()` command:

```
target_link_libraries(another_target PRIVATE my_lib)
```

The `PRIVATE` keyword states that `another_target` uses `my_lib` only in its internal implementation. Programs using `another_target` don't need to know about `my_lib`.

Linking libraries (`PUBLIC`)

Picture another dependency scenario:

- `another_target` uses `my_lib` in its internal implementation.
- **and** `another_target` defines some function that take parameters of a type defined in `my_lib`.

Programs using `another_target` also must link against `my_lib`:

```
target_link_libraries(another_target PUBLIC my_lib)
```

Link libraries (INTERFACE)

Picture another dependency scenario:

- another_target only uses my_lib in its interface.
- **but not** in its internal implementation.

```
target_link_libraries(another_target INTERFACE my_lib)
```

Behaviour of target properties across dependencies

Target properties are paired with another property `INTERFACE_<PROPERTY>`. For instance

`INTERFACE_INCLUDE_DIRECTORIES`

These properties are inherited by depending targets (such as executables and other libraries).

Example:

```
target_include_directories(my_lib INTERFACE ${CMAKE_CURRENT_SOURCE_DIR})
```

- `PRIVATE`: sets `INCLUDE_DIRECTORIES`.
- `INTERFACE`: sets `INTERFACE_INCLUDE_DIRECTORIES`.
- `PUBLIC`: sets both.

Breakout time

Let's separate the functionality from the executable itself:

```
CMakeLists.txt
src/
    <library>
exe/
    <executable>
```

Tasks:

1. Modify `src/CMakeLists.txt` so that a static library is created out of `functionality.cpp` and `functionality.hpp`.
2. Move `main.cpp` into a new directory `exe`, and add a `CMakeLists.txt` defining a new target that links against the library.
3. Modify the top-level `CMakeLists.txt` so that it processes both directories.

Printing information with `message()`

```
set(name "Jane Doe")
message(STATUS "Hello ${name}")
```

```
-- The C compiler identification is GNU 8.3.0
...
-- Hello Jane Doe
-- Configuring done
-- Generating done
```

Options for `message()`

```
message(STATUS "A simple message")
```

`STATUS` can be replaced by *e.g.* `WARNING`, `SEND_ERROR`, `FATAL_ERROR` depending on the situation.

```
message(SEND_ERROR "An error occurred but configure step continues")
```

```
CMake Error at CMakeLists.txt:2 (message):
    An error occurred but configure step continues


-- Configuring incomplete, errors occurred!
```

Finding dependencies

Libraries can be installed in various locations on your system.

CMake makes it easy to link against libraries **without having to know where they are installed**:

```
find_package(library_name CONFIG REQUIRED)
```

The above defines a new target (usually named `library_name`) that can now be linked against other targets using `target_link_libraries`.

“config” mode for `find_package`

```
find_package(library_name CONFIG REQUIRED)
```

In “config mode”, `find_package` will search for a `<PackageName>Config.cmake` file.

This file specifies all the information CMake needs (particularly where the library is installed).

This is usually given by the library vendor.

Breakout time

Look at Checkpoint 3. A new file `src/functionality_eigen.cpp` depends on the Eigen library for linear algebra.

Task: Using **`find_package`**, modify the `CMakeLists.txt` in directory `src/` to link target `cmake_course_lib` against Eigen.

Hint: Useful instructions can be found at Using Eigen in CMake Projects.

Note that keyword `NO_MODULE` is equivalent to `CONFIG`.

“module” mode for `find_package`

Libraries don’t always come with a CMake config file `<PackageName>Config.cmake`.

CMake can also find the library based on a file `Find<PackageName>.cmake`. This behaviour corresponds to using `find_package` with the keyword `MODULE`:

```
find_package(library_name MODULE REQUIRED)
```

Such *module files* are typically provided by CMake itself.

They can also be written for a particular use case if required.

Package components

Often libraries are split into different components.

E.g. Boost: filesystem, thread, date-time, program-options, numpy...

Most programs only rely on a subset of components

```
set(boost_components filesystem chrono)
find_package(Boost MODULE REQUIRED COMPONENTS ${boost_components})
```

The CMake target for a component is `<PackageName>::<ComponentName>` (*e.g.* `Boost::filesystem`).

Breakout time

Look at Checkpoint 4. The executable `exe/main.cpp` depends on the Boost Program Options library for handling command line arguments.

Task: Using **`find_package`** in `MODULE` mode, modify the `CMakeLists.txt` in directory `exe/` to find and link target `main_executable` against `Boost::program_options`.

Adding CMake functionality using `include`

Any file containing valid CMake syntax can be “included” in the current `CMakeLists.txt`:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.13)
project(IntroCMakeCourse LANGUAGES CXX)
include(file_to_include.cmake)

set(name "Foo Bar")
message(STATUS "Hello ${name}")

# cmake/file_to_include.cmake
set(name "Jane Doe")
message(STATUS "Hello ${name}")
```

Adding CMake functionality using `include`

```
-- Hello Jane Doe
-- Hello Foo Bar
-- Configuring done
...
```

Programming CMake: conditionals and loops

Conditionals. . .

```
if(expression)
    # Do something
else()
    # Do something else
endif()
```

and loops:

```
set(mylist A B C D)
foreach(var IN LISTS mylist)
    message(${var})
endforeach()
```

Programming CMake: functions

CMake allows the declaration of functions:

```
function(add a b)
    math(EXPR result "{a}+{b}")
    message("The sum is ${result}")
endfunction()
```

Functions cannot return a value.

Functions introduce a new scope.

A similar notion is CMake *macros*, which does **not** introduce a new scope.

Setting options with `option()`

Boolean variables can be declared using `option()`:

```
option(WARNINGS_AS_ERRORS "Treat compiler warnings as errors" TRUE)
```

The value of options can be specified at the command line using the `-D` syntax:

```
cmake -DWARNINGS_AS_ERRORS=FALSE ..
```

Options are a special case of “cache” variable, whose value persist between CMake runs.

Built-in CMake variables

CMake provides *a lot* of pre-defined variables which values describe the system.

For instance, the value of `CMAKE_CXX_COMPILER_ID` can be queried to determine which C++ compiler is used.

```
if(MSVC)
    set(PROJECT_WARNINGS ${MSVC_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID MATCHES ".*Clang")
    set(PROJECT_WARNINGS ${CLANG_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
    set(PROJECT_WARNINGS ${GCC_WARNINGS})
else()
    # ...
```

Using an interface “library” to apply options across targets

A useful technique for adding options to targets, for instance adding compiler flags to use with a library, is to create an empty “library”, and link that against your other targets.

Let’s see how that works, in Checkpoint 5. . .

Breakout time

Look at Checkpoint 5. The compiler should now warn us about bad C++. This is encouraged!

Add some bad C++ to `main.cpp`, for instance:

```
int unused_variable = 0;
```

Do you get a compiler warning? An error? Try configuring `WARNINGS_AS_ERRORS`:

```
cmake -DWARNINGS_AS_ERRORS=ON ..
```

```
cmake -DWARNINGS_AS_ERRORS=OFF ..
```

That's all, folks

This was only the tiniest tip of the modern CMake iceberg. There are so many great resources available, and here are just a few of them:

- The CMake documentation (link)
- Professional CMake: A Practical Guide (link)

Thank you for coming!