Getting started with CMake

1 What is CMake?

CMake is a powerful cross-platform tool for building and testing software, using simple *platform* and *compiler independent* configuration files. It can generate *makefiles* for many platforms, including Unix, OS X, Windows and Cygwin, or even Microsoft Visual Studio project and solution files. CMake has good support for hierarchically structured software projects, and applications that depend on many libraries.

A typical CMake feature is support for *out-of-place builds*. Typically, `make` produces object files and executables in the same location as the source code: an *in-place build*. The "CMake way" is to produce these files in a separate directory: for example, `my-project/src/` contains the source code and `my-project/target/` contains the build output. That way, multiple builds can live alongside the source code.

2 A simple example

This section gives a very simple example of a CMake project and its configuration files. To see it in action, try:

```
cd gobelijn/doc/tex/cmake/Pong
mkdir build && cd build
cmake .. && make && make test
```

Pretend I'm making a simple Pong game in the directory `Pong/`. Its source code is in `Pong/src/`, and a library I wrote (and my game relies on) is in `Pong/src/PhysicsLib/`.

I've put a file called `CMakeLists.txt` in that last directory — this is CMake's version of a `Makefile`. For this library, it consists of one line:

```
add_library(Physics gravity.cpp)
```

This is our first example of a CMake command. It adds a library to our project, called `Physics`, and tells CMake to build it from the specified source files (`gravity.cpp`).

One level up, in `Pong/src/`, I've put another `CMakeLists.txt` file. Do you notice the hierarchy here? Every directory in a project can have its own build instructions, usually tying the subdirectories' build steps together. This time, the instructions are a bit more complicated.

```
# Add an executable called "Pong".
add_executable(Pong game.cpp logic.cpp)

# Link to our Physics library.
add_subdirectory(PhysicsLib)
include_directories(PhysicsLib)
target_link_libraries(Pong Physics)
```

What's new here? First of all, lines starting with `#` are comments. Next, we have `add_executable`: this is just like `add_library`, but defines an *executable* called `Pong`, instead.

Then, my game is linked to my library using these three commands:

- `add_subdirectory()` adds the `Physics/` directory to the build. This means the `CMakeLists.txt` file I've put in there will be processed.
- `include_directories()` instructs the compiler to search the given path(s) for include files (so that `game.cpp` can, for example, `#include "gravity.hpp"`.)
- `target_link_libraries()` links a *target* (an executable or library defined in `CMakeLists.txt`; here our game, `Pong`) to one or more libraries (here, `Physics`).

That's it for `src/`.

On the top level, in `Pong/`, we write our project-wide `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.2)

# Use the C++11 standard.
set(CMAKE_CXX_STANDARD 11) # (a)
set(CMAKE_CXX_STANDARD_REQUIRED ON) # (b)
set(CMAKE_CXX_EXTENSIONS OFF) # (c)

project(Pong)
add_subdirectory(src)

# ...
```

The `cmake_minimum_required()` command will warn users of CMake versions older than the one specified that their version is not supported. At the time of writing (September 2016), version 3.2 is readily available everywhere, and has plenty of useful new features. If you set it to an older version, make sure to test if your project actually builds on it.

The following three lines demonstrate one of those new features: they **(a)** tell CMake we're using the C++11 standard, **(b)** tell it falling back to C++98 on old compilers is not an option, and **(c)** make sure `g++` uses `-std=c++11` and not `-std=gnu++11` (no compiler-specific voodoo).

The `project()` command declares this directory as the root of our `Pong` project, and finally, we point CMake to our source code folder.

3 Tests using CMake

There's a final bit to our project's `CMakeLists.txt` file I haven't explained yet: it defines a single test. CMake ships with a testing tool called CTest, and operating it from within a CMake configuration file is pretty easy.

```
# ...

# Make sure our Pong game is set on planet Earth.
enable_testing()
add_test(NAME on_earth COMMAND Pong)
set_tests_properties(on_earth PROPERTIES
PASS_REGULAR_EXPRESSION "g = 9\.80665")
```

First, the `enable_testing()` command is used to enable tests for this directory and below. The `add_test()` command adds a new test, given a name for this test and a command to execute (with optional arguments).

By default, a test passes if and only if executing its command yields an exit status of 0. The `set_tests_properties()` command overrides this behavior: here, the `on_earth` test expects the program output to match a certain regular expression.

4 Miscellaneous commands

Here are some assorted useful commands that might help you out when writing your own CMake build scripts.

- Don't repeat yourself: the `set` command allows you to define your own variables, which you can interpolate into later commands. Here's an example:

```
set(SOURCE_FILES flour.cpp milk.cpp eggs.cpp)
add_executable(Crepes ${SOURCE_FILES})
```

- You can add user-configurable flags to your build process using the `option()` command:

```
option(SUGAR "Coat the crepes in sugar." OFF)
```

  Users can then run, say, `cmake -DSUGAR=ON ..` to enable the option.

- To check the value of such an option (or any variable) in a CMake build script, use:

```
if(SUGAR)
  # Commands go here.
elseif(...) # Optional.
  # More commands.
else() # Also optional.
  # Final commands.
endif()
```

- When the build process gets complicated, it can be nice to keep track of where you are. The `message` command simply displays whatever message you pass it. You can specify a message type:

  ```
  message("This message is really important.")
  message(STATUS "Detected a 32-bit fridge.")
  message(WARNING "Out of milk; using soy milk instead.")
  message(FATAL_ERROR "Everything is on fire!")
  ```

  A `STATUS` message goes to `stdout`, the others go to `stderr`.

5 Further resources

The Gobelijn project uses CMake extensively, and might be a useful source of inspiration when structuring your own CMake-reliant software. The official CMake documentation page at `https://cmake.org/documentation/` contains reference documentation, training materials, and a list of frequently asked questions (FAQ).