SPECTRAL:
TECHNOLOGIES

WEEK 5

Make and Cmake

Make

- Build system native for posix-systems
- Allows to build dependency graphs and automatically deduces, which parts of the project should be recompiled
- Rarely used directly nowadays for C++-projects
- However cmake project is usually compiled to makefile, so its important to know make as well
- Can be convinient for smaller projects

makefile

- Consists of set of rules:

```
target: dependencies
        command
        command
        Command
```

- `target` is something to build, usually its a filename
- `dependencies` are other targets, which are this target depends on. `target` should be rebuilt if some of the dependencies are updated
- `command` are shell-commands to execute for building `target` . Should be indented with **tabs**
- `make target` to build specific target

makefile

- `target` in makefile is both a build target (a node in a dependency graph) and a filename. `make target` executes only when there is no `target` filename or some dependency has been updated
- Runs every time

```
hello:
	echo "Hello, World"
```

- Runs only once

```
hello:
	echo "Hello, World" >hello
```

Typical example

```
main: main.cpp mylib.o
        g++ -std=c++20 -O3 main.cpp mylib.o -o main

mylib.o: mylib.cpp mylib.h
        g++ -std=c++20 -c mylib.cpp
```

Special targets

- `make` without arguments builds the first target in `Makefile` , so its common to have special `all` as a first target:

```
all: <all dependencies of your project>
```

- `clean` target is also commonly used for removing outputs of all other targets
- `install` target usually installs the project (executables, libraries and headers) globally into system
- Adding `.PHONY` to a target will prevent Make from confusing the phony target with a file name. So even if file `clean` is created, this always runs:

```
.PHONY: clean
clean:
        rm -f clean
```

Some features

- `make` by default prints every command it runs, use `-s` flag or `@` before a command to suppress it

```
all:
        @echo "This make line will not be printed"
        echo "But this will"
```

- Every command runs in a new shell (so cd by itself won't do anything, use `&&`)
- `make` by default stops on errors, you can skip them with `-k` flag
- You can use string variables (`'` or `"` have no meaning and are assigned directly to a variable)

```
files := file1 file2
some_file: $(files)
        ./some_command $$some_var # some_var is a shell variable
```

- `-j3` to build in parallel

cmake

- More high-level cross-platform build system, which generates builds for other build-systems
  - can generate `Makefile` on Linux/Mac
  - can generate MVCC projects on Windows
- It also has concepts like targets and their dependencies, but its also a scripting language
- Usually the code is written in `CMakeLists.txt` or `.cmake` files

cmake

- cmake file usually consists of multiple commands, functions and some other control structures. Typical commands:
  - `message`: prints given message
  - `cmake_minimum_required`: sets minimum version of cmake to be used
  - `add_executable`: adds executable target with given name
  - `add_library`: adds a library target to be build from listed source files
  - `add_subdirectory`: adds a subdirectory to build

cmake

- There are also variables, which can be set via `set` command.
  Some of the variables are predefined or have special meaning

```
set(CMAKE_CXX_STANDARD 14)
message("CXX Standard: ${CMAKE_CXX_STANDARD}")
```

- Variables have a scope, which we'll study later
- There are also lists, and list is basically a string with elements concatenated with `;`

```
set(files a.txt b.txt c.txt)
# sets files to "a.txt;b.txt;c.txt"
foreach(file ${files})
    message("Filename: ${file}")
endforeach()
```

Simplest cmake project

```
cmake_minimum_required(VERSION 3.16)
project(example)

add_executable(main main.cpp)
add_library(mylib STATIC a.cpp b.cpp)
target_link_library(main mylib)
```

- Your version should be newer, than the compiler's version
- Don't blindly put the latest version you have locally as it makes more difficult to build your project on other systems. You don't use the newest cmake-features most likely

Running cmake

- From the root directory with CMakeLists.txt

```
mkdir build && cd build
cmake ..
make # or cmake --build .
```

Linking targets

- The main command is

```
target_link_libraries(target list_of_libraries)
```

- However its better to use a full form with a scope

```
target_link_libraries(target
        PUBLIC list_of_libraries1
        PRIVATE list_of_libraries2
)
```

- `PUBLIC` libraries are the libraries, that are needed for both `target` and the users of it
- `PRIVATE` libraries are the libraries, that are only needed by `target`
  - Private libraries are implementation-specific, which we want to hide

Linking targets

- Assume some library `a` consists of `a.h` and `a.cpp`
- Assume another library `b` with `b.h` and b.cpp uses some functionality from `a`
  - If `a.h` isn't included in `b.h` , then users of `b` don't need `a` and thus we link it as `PRIVATE`
  - otherwise link it as `PUBLIC`
- There is also `INTERFACE` scope, but its rarely used
- The same public/private logic applies to `target_include_directories`

Adding packages

- Use `find_package` command to find some package, e.g.

```
find_package(Boost 1.73.0 COMPONENTS date_time filesystem)
```

- This command fills a lot of variables like `<package>_FOUND` and etc.
  - `Boost_FOUND, BOOST_INCLUDE_DIRS, Boost_LIBRARIES, ...`
  - `target_link_libraries(target ${Boost_LIBRARIES})`
- Add `REQUIRED` inside the brackets to fail on missing package.

Variables and cache

- cmake maintains a special cache with variable values, so you don't need to specify them everytime you run cmake for your project.
  - `cmake -DCMAKE_BUILD_TYPE=Release ..` for the first time
  - `cmake ..` after that
- You can view the content of this cache with `cmake -L ..`
- `set(MY_VAR "VALUE")` doesn't allow you to specify a variable via command-line
- `set(MY_VAR "VALUE" CACHE STRING "Description")` sets the value if it wasn't specified via command line
- `set(MY_VAR "VALUE" CACHE STRING "Description" FORCE)` does the same, but also overwrites the value even if its already in cache
- Its very common to set bool variables, so there is a shortcut

```
option(MY_OPTION "This is settable from the command line" OFF)
```

Variables and scope

- Every variable has a scope, if you leave a function or a file in a sub directory, the variable will no longer be defined
- There are 2 ways to include other cmake files:
  - `include(some_file.cmake)` inserts the text into the current scope. Useful for configs with variables and options
  - `add_subdirectory(some_dir)` executes `CMakeLists.txt` from `some_dir` in a different (child) scope. Should be used most of the time

Conditions

- Mostly used for options

```
option(USE_BOOST "Build with boost" OFF)
if (USE_BOOST)
  find_package(Boost ... REQUIRED)
  set(LIBS "${LIBS} ${Boost_LIBRARIES}")
else()
  set(LIBS "${LIBS} something else")
endif()

target_link_libraries(target PRIVATE ${LIBS})
```

Typical layout

```
- project
 - CMakeLists.txt
 - cmake
  - FindSomeLib.cmake
  - config.cmake
 - include
  - project
   - lib.h
 - src
  - CMakeLists.txt
  - lib.cpp
 - apps
  - CMakeLists.txt
  - main.cpp
 - tests
  - CMakeLists.txt
  - test.cpp
 - extern
  - googletest
```

Configuration

- Sometimes we want to be able to specify some macro-variables via cmake. In other words to translate `-DVAR` into `#define VAR`
- `configure_file(<input> <output>)` replaces `#cmakedefine VAR ...` in `<input>` file to `#define VAR ...` in `<output>` file if `VAR` is set in this cmake invokation
- These input files with `#cmakedefine` usually have `.in` extension.

Additional materials

- Make https://makefiletutorial.com/
- Cmake https://cliutils.gitlab.io/modern-cmake/