

Introduction to CMake

CMake is a language-agnostic, cross-platform build tool and is nowadays the *de facto* standard, with large projects using it to reliably build, test, and deploy their codebases.

CMake is not a build system itself, but it generates another system's build files.

In this workshop, you will learn

- Write a CMake build system for C/C++ and Fortran projects producing libraries and/or executables.
- Ensure your build system will work on different platforms.
- Detect and use external dependencies in your project.
- (optional) Run tests for your code with *CTest*.
- (optional) Safely and effectively build mixed-language projects (Python+C/C++, Python+Fortran, Fortran+C/C++)

Prerequisites

Before attending this workshop, please make sure that you have access to a computer with a compiler for your favorite programming language and a recent version of CMake.

If you have access to a supercomputer (e.g. a [NAISS system](#)) with a compute allocation you can use that during the workshop. Any practical questions on how to use a particular HPC resource should be directed to the appropriate support desk.

You can also use your own computer for this workshop, provided that it has the necessary tools installed.

- If you do not already have these installed, we recommend that you set up an isolated software environment using `conda`.
- For Windows computers we recommend to use the **Windows Subsystem for Linux (WSL)**. Detailed instructions can be found on the [Setting up your system](#) page.

Setting up your system

In order to follow this workshop, you will need access to compilers, Python and CMake. You can use an HPC cluster if you have access to one, but the instructions here cover how to install the prerequisites on your own computer.

These instructions are based on installing compilers and CMake via the [Conda package and environment manager](#), as it provides a convenient way to install binary packages in an isolated software environment.

For Windows users


We strongly recommend to use (and install if necessary) the **Windows Subsystem for Linux (WSL)** as it is a powerful tool which will likely be useful also after the workshop. Inside WSL you will need Python 3 and the conda environment manager. A useful guide to do this is found at [HERE](#). The installation of the required dependencies in a **WSL** terminal is documented below.

For MacOS and Linux users

MacOS and Linux users can simply open a terminal and install [Miniconda](#):

- For MacOS see [HERE](#).
- For Linux see [HERE](#).

Creating an environment and installing packages

Once you have `conda` installed (and `WSL` if you're using Windows OS) you can use the  `environment.yml` file to install dependencies.

First save it to your hard drive by clicking the link, and then in a terminal navigate to where you saved the file and type:

```
conda env create -f environment.yml
```

You then need to activate the new environment by:

```
conda activate intro-to-cmake
```

Now you should have CMake, compilers, Python and a few other packages installed!

The CMake modules on the Dardel cluster

If you can access to the Dardel cluster, you can follow the instructions below to load the CMake module from before running hand-on code examples.

- [Login to the Dardel cluster](#).

- In your home directory, you can create a folder to keep all your data (replace XXXXX with a special string for you).

```
mkdir cmake_XXXXX  
cd cmake_XXXXX/
```

- Loading the CMake module using the commands below.

```
ml PDC/23.12  
ml cmake/3.27.7  
cmake --version
```

- Cloning the github repository using Git, you can access to the code examples in the *content/code* subdirectory.

```
git clone https://github.com/ENCCS/intro-cmake.git  
cd intro-cmake/content/code/
```

- Following the steps in hand-on exercises to run the code examples.

From sources to executables

? Questions

- How do we use CMake to compile source files to executables?

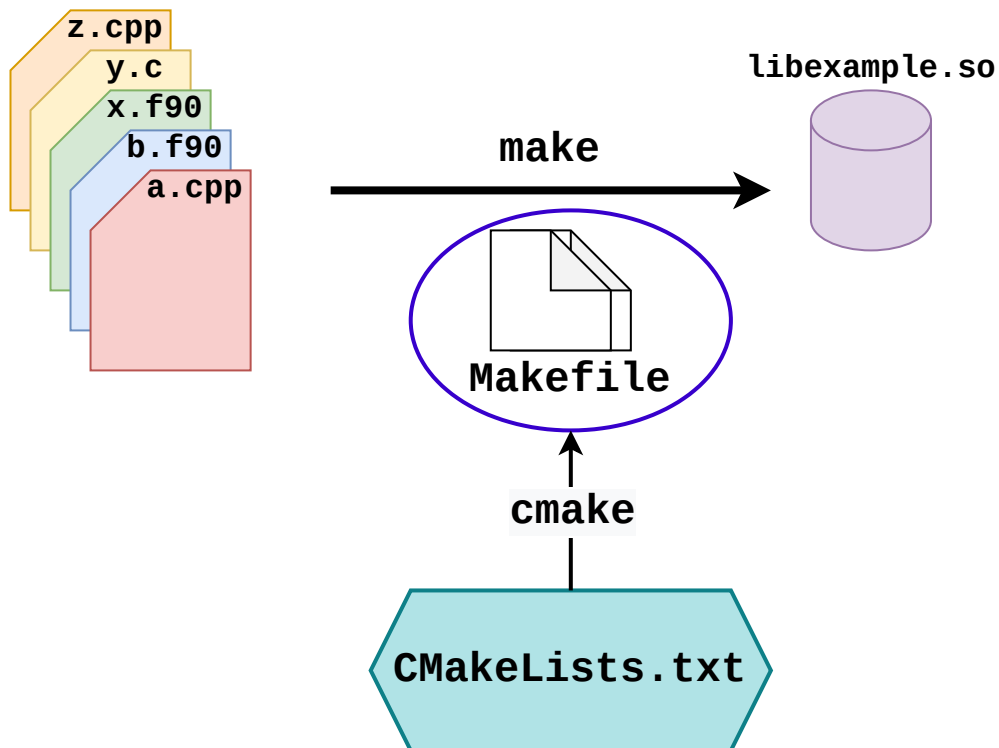
! Objectives

- Learn what tools are available in the CMake suite.
- Learn how to write a simple `CMakeLists.txt`.
- Learn the difference between **build systems**, **build tools**, and **build system generator**.
- Learn to distinguish between *configuration*, *generation*, and *build* time.

What is CMake and why should you care?

Software is everywhere and so are build systems. Whenever you run a piece of software, anything from calendar apps to computationally-intensive programs, there was a build system involved in transforming the plain-text source code into binary files that could run on the device you are using.

CMake is a **build-system generator**: it provides a family of tools and a *domain-specific language* (DSL) to **describe** what the build system should achieve when the appropriate build tools are invoked. The DSL is platform- *and* compiler-agnostic: you can reuse the same CMake scripts to obtain **native** build systems on any platform.



On GNU/Linux, the native build system will be a collection of `Makefile`-s. The `make` build tool uses these `Makefile`-s to transform sources to executables and libraries.

CMake abstracts the process of generating the `Makefile`-s away into a generic DSL.

A CMake-based build system:

- can bring your software closer to being platform- *and* compiler-agnostic.
- has good support within many integrated development environments (IDEs).
- automatically tracks and propagates internal dependencies in your project.
- is built on top of well-maintained functionality for automated dependency detection.

Hello, CMake!

❗ Compiling “Hello, world” with CMake

We will now proceed to compile a single source file to an executable. Choose your favorite language and start typing along!

C++

Fortran

You can find the file with the complete source code in the `content/code/00_hello-world/cxx` folder.

```
#include <cstdlib>
#include <iostream>

int main() {
    std::cout << "Hello world from C" << std::endl;

    return EXIT_SUCCESS;
}
```

A working solution is in the `solution` subfolder.

1. The folder contains only the source code. We need to add a file called `CMakeLists.txt` to it. CMake reads the contents of these special files when generating the build system.
2. The first thing we will do is declare the requirement on minimum version of CMake:

```
cmake_minimum_required(VERSION 3.18)
```

3. Next, we declare our project and its programming language:

```
project(Hello LANGUAGES CXX)
```

4. We create an *executable target*. CMake will generate rules in the build system to compile and link our source file into an executable:

```
add_executable(hello hello.cpp)
```

5. We are ready to call CMake and get our build system:

```
cmake -S. -Bbuild
```

6. And finally build our executable:

```
cmake --build build
```

Important issues for `CMakeLists.txt` file

1. Any CMake build system will invoke the following commands in its **root** `CMakeLists.txt` :

! `cmake_minimum_required`

```
cmake_minimum_required(VERSION <min>[...<max>] [FATAL_ERROR])
```

❗ Parameters

VERSION : Minimum and, optionally, maximum version of CMake to use.

FATAL_ERROR : Raise a fatal error if the version constraint is not satisfied. This option is ignored by CMake >=2.6

❗ project

```
project(<PROJECT-NAME>
  [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
  [DESCRIPTION <project-description-string>]
  [HOMEPAGE_URL <url-string>]
  [LANGUAGES <language-name>...])
```

❗ Parameters

<PROJECT-NAME> : The name of the project.

LANGUAGES : Languages in the project.

- The case of CMake commands does not matter: the DSL is case-insensitive. However, the plain-text files that CMake parses **must be called** `CMakeLists.txt` and the case matters! The variable names are also case sensitive!
- The command to add executables to the build system is `add_executable` :

❗ add_executable

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
  [EXCLUDE_FROM_ALL]
  [source1] [source2 ...])
```

- Using CMake you can abstract the generation of the build system and also the invocation of the build tools.

❗ Put your `CMakeLists.txt` under version control

All CMake-related files will evolve together with your codebase. It's a good idea to put them under [version control](#). On the contrary, any of the *generated* native build-system files, e.g. `Makefile`-s, should not be version-controlled.

❗ The command-line interface to CMake

Let us get acquainted with the CMake and especially its command-line interface.

We can get help at any time with the command

```
cmake --help
```

This will output quite a number of options to your screen. We can analyze the last few lines first:

Generators

The following generators are available on this platform (* marks default):

* Unix Makefiles	= Generates standard UNIX makefiles.
Green Hills MULTI	= Generates Green Hills MULTI files.
Ninja	= Generates build.ninja files.
Ninja Multi-Config	= Generates build-<Config>.ninja files.
Watcom WMake	= Generates Watcom WMake makefiles.
CodeBlocks - Ninja	= Generates CodeBlocks project files.
CodeBlocks - Unix Makefiles	= Generates CodeBlocks project files.
CodeLite - Ninja	= Generates CodeLite project files.
CodeLite - Unix Makefiles	= Generates CodeLite project files.
Sublime Text 2 - Ninja	= Generates Sublime Text 2 project files.
Sublime Text 2 - Unix Makefiles	= Generates Sublime Text 2 project files.
Kate - Ninja	= Generates Kate project files.
Kate - Unix Makefiles	= Generates Kate project files.
Eclipse CDT4 - Ninja	= Generates Eclipse CDT 4.0 project files.
Eclipse CDT4 - Unix Makefiles	= Generates Eclipse CDT 4.0 project files.

In CMake terminology, the native build scripts and build tools are called **generators**. On any particular platform, the list will show which native build tools can be used through CMake. They can either be “plain”, such as `Makefile` -s or Ninja, or IDE-like projects.

The `-S` switch specifies which source directory CMake should scan: this is the folder containing the *root* `CMakeLists.txt`, i.e., the one containing the `|project|` command. By default, CMake will allow *in-source* builds, i.e. storing build artifacts alongside source files. This is **not** good practice: you should always keep build artifacts from sources separate. Fortunately, the `-B` switch helps with that, as it is used to give where to store build artifacts, including the generated build system. This is the minimal invocation of `cmake`:

```
cmake -S. -Bbuild
```

To switch to another generator, we will use the `-G` switch (make sure that the Ninja is correctly built before running the command below):

```
cmake -S. -Bbuild -GNinja
```

Options to be used at build-system generation are passed with the `-D` switch. For example, to change compilers:

```
cmake -S. -Bbuild -GNinja -DCMAKE_CXX_COMPILER=clang++
```

Finally, you can access to the full CMake manual with:

```
cmake --help-full
```

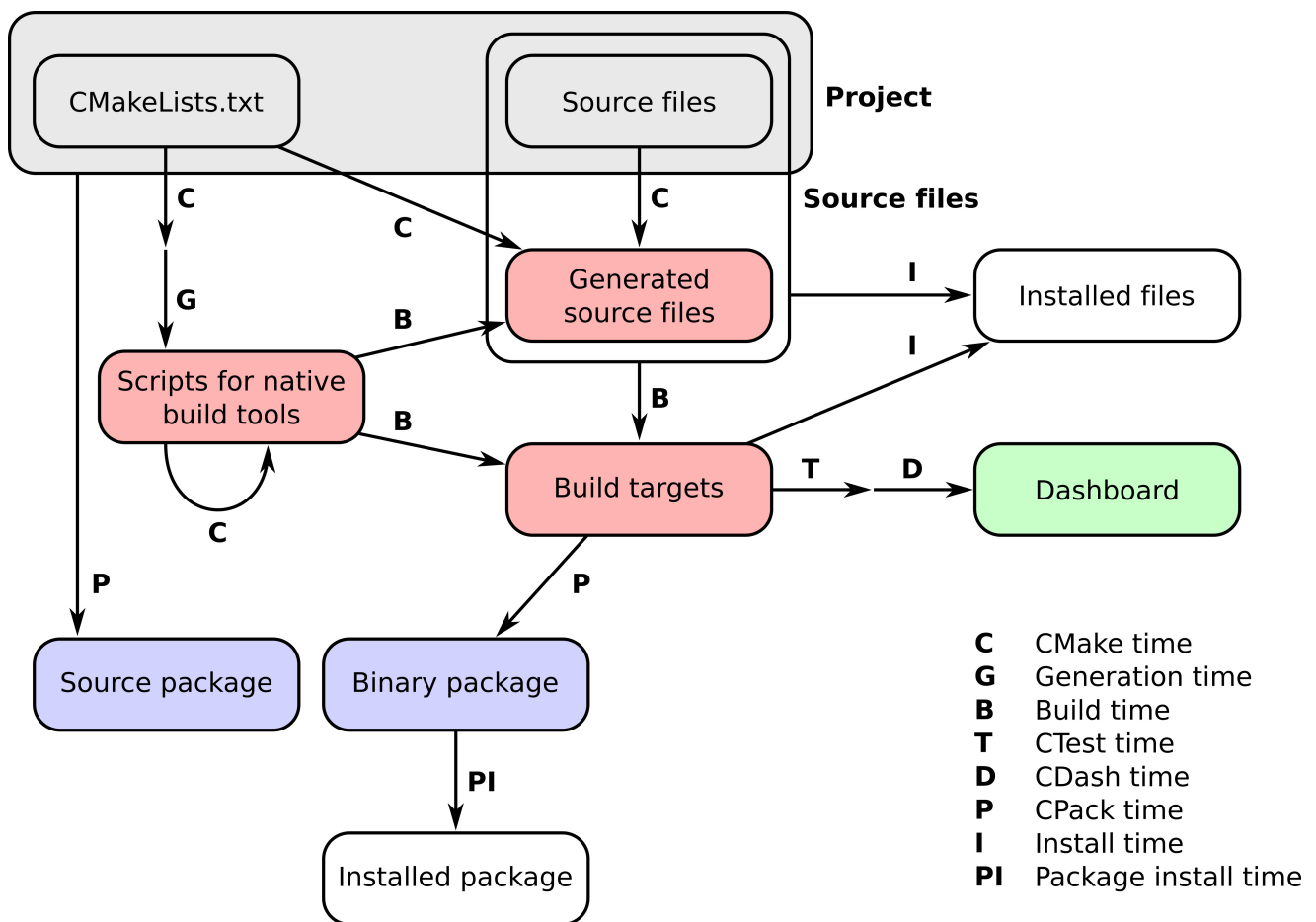
You can also inquire about a specific module, command or variable:

```
cmake --help-variable CMAKE_GENERATOR
```

A complete toolchain

The family of tools provided with CMake offers a complete toolchain to manage the development cycle: from sources to build artifacts, testing, and deployment. We refer to these stages as *CMake times* and each tool is appropriate at a specific time. In this workshop, we will discuss:

- **CMake time** or **configure time**. This is the stage when `cmake` is invoked to parse the `CMakeLists.txt` in your project, configure and generate the build system.
- **Build time**. This is handled by the native build tools, but, as we have seen, these can be effectively wrapped by `cmake` itself.
- **CTest time** or **test time**. At this stage, you will test your build artifacts.



You can manage all these stages of a software project's lifetime with tools provided by CMake.

This figure shows all these stages (times) and which tool is appropriate for each. This figure is reproduced from [CMake Cookbook](#) and is licensed under the terms of the [CC-BY-SA](#).

Producing libraries

CMake can of course be used to produce libraries as well as executables. The relevant command is `add_library`:

! `add_library`

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [<source>...])
```

You can link libraries into executables with `target_link_libraries`:

! `target_link_libraries`

```
target_link_libraries(<target>
                     <PRIVATE|PUBLIC|INTERFACE> <item>...
                     [<PRIVATE|PUBLIC|INTERFACE> <item>...])
```

📌 Executables and libraries are targets

We will encounter the term **target** repeatedly. In CMake, a target is any object given as first argument to `add_executable` or `add_library`. Targets are the basic atom in CMake. Whenever you will need to organize complex projects, think in terms of its targets and their mutual dependencies.

The whole family of CMake commands `target_*` can be used to express chains of dependencies and is much more effective than keeping track of state with variables. We will clarify these concepts in [Target-based build systems with CMake](#).

🔧 Exercise 1: Producing libraries

C++

Fortran

You can find a scaffold project in the `content/code/01_producing-libraries/cxx` folder.

1. Write a `CMakeLists.txt` to compile the source files `Message.hpp` and `Message.cpp` into a library. **DO NOT** specify the type of library, shared or static, explicitly.
2. Add an executable from the `hello-world.cpp` source file.
3. Link the library into the executable.

A working solution is in the `solution` subfolder.

What kind of library did you get? Static or shared?

📌 Keypoints

- CMake is a **build system generator**, not a build system.
- You write `CMakeLists.txt` to describe how the build tools will create artifacts from sources.
- You can use the CMake suite of tools to manage the whole lifetime: from source files to tests to deployment.
- The structure of the project is mirrored in the build folder.

CMake syntax

? Questions

- How can we achieve more control over the build system generated by CMake?
- Is it possible to let the user decide what to generate?

❗ Objectives

- Learn how to define variables with `set` and use them with the `${}` operator for [variable references](#).
- Learn the syntax for conditionals in CMake: `if` - `elseif` - `else` - `endif`.
- Learn the syntax for loops in CMake: `foreach`.
- Learn how CMake structures build artifacts.
- Learn how to print helpful messages.
- Learn how to handle user-facing options: `option` and the role of CMake cache.

CMake offers a **domain-specific language** (DSL) to describe how to generate a build system native to the specific platform you might be running on. In this episode, we will get acquainted with its syntax.

The CMake DSL

Remember that the DSL is **case-insensitive**. We will now have a look at its main elements.

Variables

These are either CMake- or user-defined variables. You can obtain the list of CMake-defined variables with the command:

```
cmake --help-variable-list
```

You can create a new variable with the `set` command:

❗ set

```
set(<variable> <value>... [PARENT_SCOPE])
```

Variables in CMake are always of string type, but certain commands can interpret them as other types. If you want to declare a *list* variable, you will have to provide it as a ;-separated string. Lists can be manipulated with the `list` family of commands.

You can inspect the value of any variable by *dereferencing* it with the `${}` operator, as in bash shell. For example, the following snippet sets the content of `hello` variable and then prints it:

```
set(hello "world")
message("hello ${hello}")
```

Two notes about **variable references**:

- if the variable within the `${}` operator is not set, you will get an empty string.
- you can *nest* variable references: `${outer_${inner_variable}_variable}`. They will be evaluated from the inside out.

One of the most confusing aspects in CMake is the **scoping of variables**. There are three variable scopes in the DSL:

- **Function:** In effect when a variable is `set` within a function, the variable will be visible within the function, but not outside.
- **Directory:** In effect when processing a `CMakeLists.txt` in a directory, variables in the parent folder will be available, but any that is `set` in the current folder will not be propagated to the parent.
- **Cache:** These variables are **persistent** across calls to `cmake` and available to all scopes in the project. Modifying a cache variable requires using a special form of the `set` function:

! `set`

```
set(<variable> <value>... CACHE <type> <docstring> [FORCE])
```

Here is a list of few **CMake-defined variables**:

- `PROJECT_BINARY_DIR`. This is the build folder for the project.
- `PROJECT_SOURCE_DIR`. This is the location of the root `CMakeLists.txt` in the project.
- `CMAKE_CURRENT_LIST_DIR`. This is the folder for the `CMakeLists.txt` currently being processed.

Help on a specific built-in variable can be obtained with:

```
cmake --help-variable PROJECT_BINARY_DIR
```

Commands

These are provided by CMake and are essential building blocks of the DSL, as they allow you to manipulate variables. They include control flow constructs and the `target_*` family of commands.

You can find a complete list of available commands with:

```
cmake --help-command-list
```

Functions and **macros** are built on top of the basic built-in commands and are either CMake- or user-defined. These prove useful to avoid repetition in your CMake scripts.

The difference between a function and a macro is their *scope*:

- **Functions** have their own scope: variables defined inside a function are not propagated back to the caller.
- **Macros** do not have their own scope: variables from the parent scope can be modified and new variables in the parent scope can be set.

Help on a specific built-in command, function or macro can be obtained with:

```
cmake --help-command target_link_libraries
```

Modules

These are collections of functions and macros and are either CMake- or user-defined. CMake comes with a rich ecosystem of modules and you will probably write a few of your own to encapsulate frequently used functions or macros in your CMake scripts.

You will have to include the module to use its contents, for example:

```
include(CMakePrintHelpers)
```

The full list of built-in modules is available with:

```
cmake --help-module-list
```

Help on a specific built-in module can be obtained with:

```
cmake --help-module CMakePrintHelpers
```

Flow control

The `if` and `foreach` commands are available as flow control constructs in the CMake DSL and you are surely familiar with their use in other programming languages.

Since *all* variables in CMake are strings, the syntax for `if` and `foreach` appears in a few different variants.

! if

```
if(<condition>)
  # <commands>
elseif(<condition>) # optional block, can be repeated
  # <commands>
else()              # optional block
  # <commands>
endif()
```

The truth value of the conditions in the `if` and `elseif` blocks is determined by boolean operators. In the CMake DSL:

- True is any expression evaluating to: `1`, `ON`, `TRUE`, `YES`, and `Y`.
- False is any expression evaluating to: `0`, `OFF`, `FALSE`, `NO`, `N`, `IGNORE`, and `NOTFOUND`.

CMake offers boolean operator for string comparisons, such as `STREQUAL` for string equality, and for version comparisons, such as `VERSION_EQUAL`.

! Variable expansions in conditionals

The `if` command expands the contents of variables before evaluating their truth value. See [official documentation](#) for further details.

🔧 Exercise 2: Conditionals in CMake

Modify the `CMakeLists.txt` from the previous exercise to build either a *static* or a *shared* library depending on the value of the boolean `MAKE_SHARED_LIBRARY`:

1. Define the `MAKE_SHARED_LIBRARY` variable.
2. Write a conditional checking the variable. In each branch call `add_library` appropriately.

C++

Fortran

You can find a scaffold project in the `content/code/02_conditionals/cxx` folder. A working solution is in the `solution` subfolder.

You can perform the same operation on a collection of items with `foreach`:

! foreach

```
foreach(<loop_var> <items>)  
  # <commands>  
endforeach()
```

The list of items is either space- or ;-separated. `break()` and `continue()` are also available.

! Loops in CMake

In this typealong, we will show how to use `foreach` and lists in CMake. We will work from a scaffold project in the `content/code/03_loops-cxx` folder.

The goal is to compile a library from a bunch of source files: some of them are to be compiled with `-O3` optimization level, while some others with `-O2`. We will set the compilation flags as properties on the library target. Targets and properties will be discussed at greater length in [Target-based build systems with CMake](#).

A working solution is in the `solution` subfolder.

It is instructive to browse the build folder for the project using the `tree` command:

```
ml tree/2.1.1  
tree -L 2 build
```

Then you can get the code structure like this:

```
build
├─ CMakeCache.txt
├─ CMakeFiles
│   └─ 3.27.7
│       ├── cmake.check_cache
│       ├── CMakeConfigureLog.yaml
│       ├── CMakeDirectoryInformation.cmake
│       ├── CMakeScratch
│       ├── compute-areas.dir
│       ├── geometry.dir
│       ├── Makefile2
│       ├── Makefile.cmake
│       ├── pkgRedirects
│       ├── progress.marks
│       └─ TargetDirectories.txt
├─ cmake_install.cmake
├─ compute-areas
├─ libgeometry.a
└─ Makefile
```

We note that:

- The project was configured with `Makefile` generator.
- The cache is a plain-text file `CMakeCache.txt`.
- For every target in the project, CMake will create a subfolder `<target>.dir` under `CMakeFiles`. The intermediate object files are stored in these folders, together with compiler flags and link line.
- The build artifacts, `compute-areas` and `libgeometry.a`, are stored at the root of the build tree.

Printing messages

You will most likely have to engage in debugging your CMake scripts at some point. Print-based debugging is the most effective way and the main workhorse for this will be the `message` command:

! message

```
message([<mode>] "message to display")
```

! Parameters

`<mode>`

What type of message to display, for example:

- `STATUS`, for incidental information.
- `FATAL_ERROR`, to report an error that prevents further processing and generation.

It should be noted that `message` can be a bit awkward to work with, especially when you want to print the name *and* value of a variable. Including the built-in module `CMakePrintHelpers` will make your life easier when debugging, since it provides the `cmake_print_variables` function:

! `cmake_print_variables`

```
cmake_print_variables(var1 var2 ... varN)
```

This command accepts an arbitrary number of variables and prints their name *and* value to standard output. For example:

```
include(CMakePrintHelpers)
cmake_print_variables(CMAKE_C_COMPILER CMAKE_MAJOR_VERSION DOES_NOT_EXIST)
```

gives:

```
-- CMAKE_C_COMPILER="/usr/bin/gcc" ; CMAKE_MAJOR_VERSION="2" ; DOES_NOT_EXIST=""
```

! Keypoints

- CMake offers a full-fledged DSL which empowers you to write complex `CMakeLists.txt`.
- Variables have scoping rules.
- The structure of the project is mirrored in the build folder.

Target-based build systems with CMake

? Questions

- How can we handle complex projects with CMake?
- What exactly are **targets** in the CMake domain-specific language (DSL)?

! Objectives

- Learn that the **basic elements** in CMake are not variables, but targets.
- Learn about properties of targets and how to use them.
- Learn how to use **visibility levels** to express dependencies between targets.
- Learn how to handle multiple targets in one project.
- Learn how to work with projects spanning multiple folders.

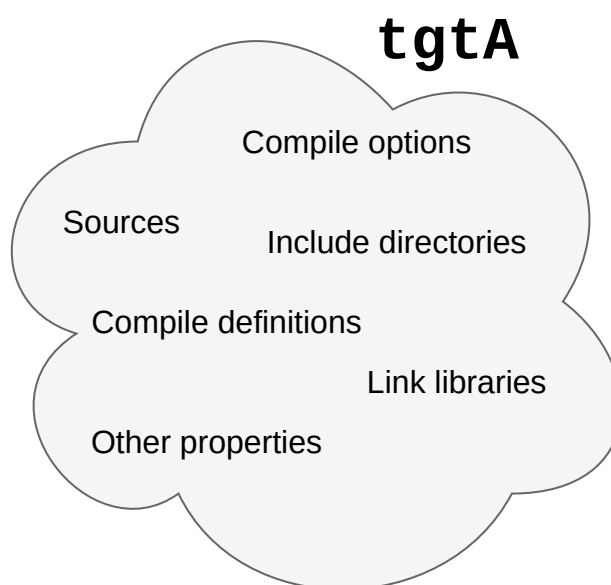
Real-world projects require more than compiling a few source files into executables and/or libraries. In most cases, you will come to projects comprising hundreds of source files sprawling in a complex source tree.

With the advent of CMake 3.0, also known as **Modern CMake**, there has been a significant shift that the CMake domain-specific language (DSL) is structured, which can help you keep the complexity of the build system in check. Rather than relying on **variables** to convey information in a project, all what you need in modern CMake is **targets** and **properties**.

Targets

A target is the basic element in the CMake DSL, which can be declared by either `add_executable` or `add_library`. Any target has a collection of **properties**, which define:

- *how* the build artifact should be produced,
- *how* it should be used by other targets in the project that depend on it.



In CMake, the five most used commands used to handle targets are:

- `target_sources`, specifying which source files to use when compiling a target.
- `target_compile_options`, specifying which compiler flags to use.
- `target_compile_definitions`, specifying which compiler definitions to use.
- `target_include_directories`, specifying which directories will contain header (for C/C++) and module (for Fortran) files.

- `target_link_libraries`, specifying which libraries to link into the current target.

There are additional commands in the `target_*` family:

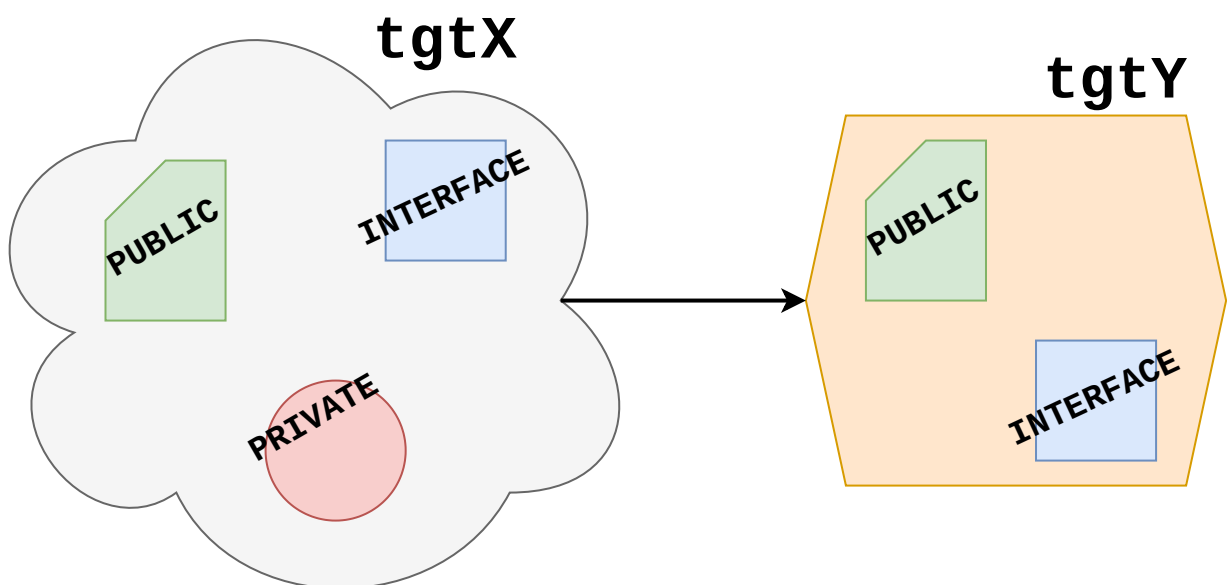
```
$ cmake --help-command-list | grep "^target_"

target_compile_definitions
target_compile_features
target_compile_options
target_include_directories
target_link_directories
target_link_libraries
target_link_options
target_precompile_headers
target_sources
```

Visibility levels

Why it is robust to use targets and properties than using variables? Given a target `tgtX`, we can invoke one command in the `target_*` family as follows.

```
target_link_libraries(tgtX
PRIVATE tgt1
INTERFACE tgt2
PUBLIC tgt3
)
```



*Properties on targets have varied **visibility levels**, which determine how CMake should propagate them between interdependent targets.*

Visibility levels `PRIVATE`, `PUBLIC`, or `INTERFACE` are very powerful and herein we will briefly demonstrate their difference.

A complete source code is available in the `content/code/04_visibility-levels/` folder.

In this code example, we want to compile a C++ library and an executable:

- The library code is in the `account` subfolder. It consists of one source (`account.cpp`) and one header file (`account.hpp`).
- The header file and the shared library are needed for the `bank.cpp` to produce the `bank` executable.
- We also want to use the `-ffast-math` compiler flag and propagate it throughout the project.

Thus code structure is arranged in the following format:

1. The `account` target declares the `account.cpp` source file as `PRIVATE` since it is only needed to produce the shared library.

```
target_sources(account
PRIVATE
  account.cpp
)
```

2. The `-ffast-math` is instead `PUBLIC` as since it needs to be propagated to all targets consuming `account`.

```
target_compile_options(account
PUBLIC
  "-ffast-math"
)
```

3. The `account` folder is an include directory with `INTERFACE` visibility because only targets consuming `account` need to know where `account.hpp` is located.

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

When working out which visibility settings to use for the properties of your targets you can refer to the following table:

Who needs?	Others	
Target	YES	NO
YES	PUBLIC	PRIVATE
NO	INTERFACE	N/A

An additional code example to demonstrate the difference of the visibility levels `PRIVATE`, `PUBLIC`, or `INTERFACE` is available in the [CodeRefinery CMake Workshop](#) lesson materials.

Properties

CMake lets you set properties at many different levels of visibility across the project:

- **Global scope.** These are equivalent to variables set in the root `CMakeLists.txt`. Their use is, however, more powerful as they can be set from *any* leaf `CMakeLists.txt`.
- **Directory scope.** These are equivalent to variables set in a given leaf `CMakeLists.txt`.
- **Target.** These are the properties set on targets that we discussed above.
- **Test.**
- **Source files.** For example, compiler flags.
- **Cache entries.**
- **Installed files.**

For a complete list of properties known to CMake:

```
$ cmake --help-properties | less
```

You can get the current value of any property with `get_property` and set the value of any property with `set_property`.

Multiple folders

In the code example about the visibility levels, we have two `CMakeLists.txt` files, one in the `account` subfolder and one in the main folder. This enables the code maintenance being easier if we split the CMake configuration into multiple `CMakeLists.txt` with the help of `add_subdirectory`. Our goal is to have multiple `CMakeLists.txt` files as close as possible to the source files.

```

project/
├── CMakeLists.txt          <--- Root
├── external
│   └── CMakeLists.txt      <--- Leaf at level 1
└── src
    ├── CMakeLists.txt      <--- Leaf at level 1
    ├── evolution
    │   └── CMakeLists.txt  <--- Leaf at level 2
    ├── initial
    │   └── CMakeLists.txt  <--- Leaf at level 2
    ├── io
    │   └── CMakeLists.txt  <--- Leaf at level 2
    └── parser
        └── CMakeLists.txt  <--- Leaf at level 2

```

Each folder in a multi-folder project will contain a `CMakeLists.txt`: a source tree with one **root** and many **leaves**.

- The root `CMakeLists.txt` will contain the invocation of the `project` command: variables and targets declared in the root have effectively global scope.
- The `PROJECT_SOURCE_DIR` will point to the folder containing the root `CMakeLists.txt`.
- In order to move between the root and a leaf or between leaves, you will use the `add_subdirectory` command.

Typically, you only need to pass the first argument: the folder within the build tree will be automatically computed by CMake. We can declare targets at any level, not necessarily the root: a target is visible at the level at which it is declared and all higher levels.

Exercise 05: Cellular automata

Let's work with a project spanning multiple folders. We will implement a relatively simple code to compute and print to screen elementary **cellular automata**. We separate the sources into `src` and `external` to simulate a nested project which reuses an external project.

Your goal is to:

1. Build the main executable at `content/code/05_automata/cxx/` for C++ and `content/code/05_automata/fortran/` for Fortran.
2. Where are the obtained executables located in the build tree? Remember that CMake generates a build tree mirroring the source tree.
3. The executable will accept 3 arguments: the length, number of steps, and automaton rule. You can run it with:

```
$ automata 40 5 30
```

The output will be:

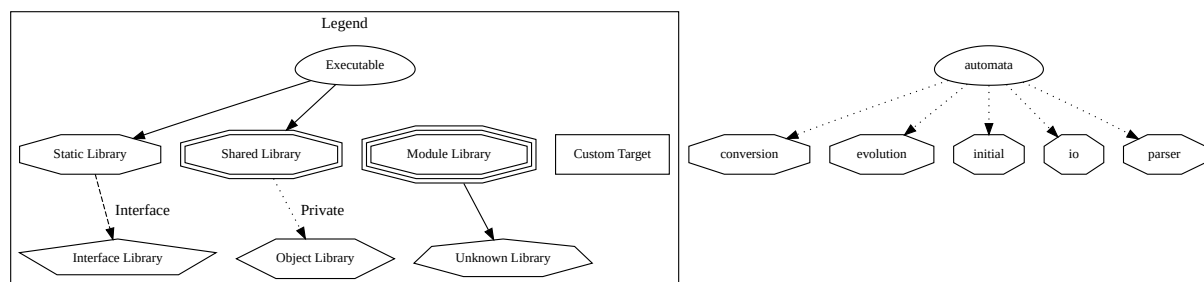
```
length: 40
number of steps: 5
rule: 30

      *
    ***
  **  *
**  ****
**  *  *
**  *  *
**  ****  **
```

❗ The internal dependency tree

You can visualize the dependencies between targets in the project with Graphviz (make sure that you have installed the Graphviz package):

```
$ cd build
$ cmake --graphviz=project.dot ..
$ dot -T svg project.dot -o project.svg
```



The dependencies between targets in the cellular automata project.

❗ Keypoints

- Using **targets**, you can achieve granular control over how artifacts are built and how their dependencies are handled.
- Compiler flags, definitions, source files, include folders, link libraries, and linker options are **properties** of a target.
- Avoid using variables to express dependencies between targets: use visibility levels **PRIVATE**, **INTERFACE**, **PUBLIC** and let CMake figure out the details.
- To keep the complexity of the build system at a minimum, each folder in a multi-folder project should have its own CMake script.

Probing compilation, linking, and execution

? Questions

- How can you add custom steps to your build system with CMake?

! Objectives

- Learn how and when to use `execute_process`
- Learn how to use `add_custom_command` with targets.
- Learn how to test compilation, linking, and execution.

CMake lets you run arbitrary commands at any stage in the project lifecycle. This is yet another mechanism for fine-grained customization and we will discuss some of the options in this episode.

Running custom commands at *configure-time*

The most straightforward method is to explicitly run one (or more) child process(es) when invoking the `cmake` command. This is achieved with the `execute_process` command.

! `execute_process`

```
execute_process(COMMAND <cmd1> [args1...]  
               [COMMAND <cmd2> [args2...] [...]]  
               [WORKING_DIRECTORY <directory>]  
               [TIMEOUT <seconds>]  
               [RESULT_VARIABLE <variable>]  
               [RESULTS_VARIABLE <variable>]  
               [OUTPUT_VARIABLE <variable>]  
               [ERROR_VARIABLE <variable>]  
               [INPUT_FILE <file>]  
               [OUTPUT_FILE <file>]  
               [ERROR_FILE <file>]  
               [OUTPUT_QUIET]  
               [ERROR_QUIET]  
               [OUTPUT_STRIP_TRAILING_WHITESPACE]  
               [ERROR_STRIP_TRAILING_WHITESPACE]  
               [ENCODING <name>])
```

Executes one or more child processes. The standard output and standard error streams are recorded into `OUTPUT_VARIABLE` and `ERROR_VARIABLE`, respectively. The result of the last child process is saved into `RESULT_VARIABLE`.

It is important to note that any command invoked through `execute_process` will only be run at **configure-time**, i.e. when running the `cmake` command. You **should not** rely on `execute_process` to update any artifacts at **build-time**.

Exercise 06: Find a Python module

In this exercise, we'll use `execute_process` to check whether the `cffi` Python module is installed in your environment. On the command line, you would do:

```
$ python -c "import cffi; print(cffi.__version__)"
```

Your goal is to replicate the same in CMake. The scaffold code is in `content/code/06_find_cffi`. You will have to modify the call to `execute_process` to run the command above.

A working example is in the `solution` subfolder.

Note the use of `find_package(Python REQUIRED)` to obtain the `python` executable. CMake comes with many modules dedicated to the detection of dependencies, such as Python. These are conventionally called `Find<dependency>.cmake` and you can inspect their documentation with:

```
$ cmake --help-module FindPython | more
```

We will revisit uses of `find_package` later on in [Finding and using dependencies](#).

Custom commands for your targets

As mentioned, the main problem of `execute_process` is that it will run a command at *configure-time*, when the `cmake` command is first invoked. It is thus *not* a viable alternative if we intend to perform some specific actions depending on targets or make the result of the custom commands a dependency for other targets.

Both cases have real-world examples, such as when using automatically generated code. The CMake command `add_custom_command` can be used in some of these instances.

 `add_custom_command`

```
add_custom_command(TARGET <target>
    PRE_BUILD | PRE_LINK | POST_BUILD
    COMMAND command1 [ARGS] [args1...]
    [COMMAND command2 [ARGS] [args2...] ...]
    [BYPRODUCTS [files...]]
    [WORKING_DIRECTORY dir]
    [COMMENT comment]
    [VERBATIM] [USES_TERMINAL])
```

Add one or more custom commands to a target, such as a library or an executable. The commands can be executed before linking (with `PRE_BUILD` and `PRE_LINK`) or after (with `POST_BUILD`)

Exercise 07: Before and after build

We want to perform some action before and after building a target, in this case a Fortran executable:

- Before building, we want to read the link line, as produced by CMake, and echo it to standard output. We use the `echo-file.py` Python script.
- After building, we want to check the size of the static allocations in the binary, by invoking the `size` command. We use the `static-size.py` Python script.

The scaffold code is in `content/code/07_pre_post-f`.

1. Add CMake commands to build the `example` executable from the Fortran sources. Find the text file with the link line under the build folder. Hint: have a look in `CMakeFiles` and keep in mind the name you gave to the target.
2. Call `add_custom_command` with `PRE_LINK` to invoke the `echo-file.py` Python script.
3. Call `add_custom_command` with `POST_BUILD` to invoke the `static-size.py` Python script.

A working example is in the `solution` subfolder.

Testing compilation, linking, and execution

We also want to be able to run checks on our compilers and linkers. Or check whether a certain library can be used correctly before attempting to build our own artifacts. CMake provides modules and commands for these purposes:

- `Check<LANG>CompilerFlag` providing the `check_<LANG>_compiler_flag` function, to check whether a compiler flag is valid for the compiler in use.
- `Check<LANG>SourceCompiles` providing the `check_<LANG>_source_compiles`. Which check whether a given source file compiles with the compiler in use.

- `Check<LANG>SourceRuns` providing the `check_<LANG>_source_runs`, to make sure that a given source snippet compiles, links, and runs.

In all cases, `<LANG>` can be one of `CXX`, `C` or `Fortran`.

Exercise 08: Check that a compiler accepts a compiler flag

Compilers evolve: they add and/or remove flags and sometimes you will face the need to test whether some flags are available before using them in your build.

The scaffold code is in `content/code/08_check_compiler_flag`.

1. Implement a `CMakeLists.txt` to build an executable from the `asan-example.cpp` source file.
2. Check that the address sanitizer flags are available with `check_cxx_compiler_flag`. The flags to check are `-fsanitize=address -fno-omit-frame-pointer`. Find the command signature with:

```
$ cmake --help-module CheckCXXCompilerFlag
```

3. If the flags do work, add them to the those used to compile the executable target with `target_compile_options`.

A working example is in the `solution` subfolder.

Exercise 09: Testing runtime capabilities

Testing that some features will work properly for your code requires not only compiling an object files, but also linking an executable and running it successfully.

The scaffold code is in `content/code/09_check_source_runs`.

1. Create an executable target from the source file `use-uuid.cpp`.
2. Add a check that linking against the library produces working executables. Use the following C code as test:

```
#include <uuid/uuid.h>

int main(int argc, char * argv[]) {
    uuid_t uuid;
    uuid_generate(uuid);
    return 0;
}
```

`check_c_source_runs` requires the test source code to be passed in as a *string*. Find the command signature with:

```
$ cmake --help-module CheckCSourceRuns
```

3. If the test is successful, link executable target against the UUID library: use the `PkgConfig::UUID` target as argument to `target_link_libraries`.

A working example is in the `solution` subfolder.

Keypoints

- You can customize the build system by executing custom commands.
- CMake offers commands to probe compilation, linking, and execution.

Finding and using dependencies

Questions

- How can I use CMake to detect and use the dependencies of my project?

Objectives

- Learn how to use `find_package`.
- Learn what other detection alternatives exist.

The vast majority of software projects do not happen in a vacuum: they will have dependencies on existing frameworks and libraries. Good documentation will instruct your users to ensure that these are satisfied in their programming environment. The build system is the appropriate place to check that these preconditions are met and that your project can be built correctly. In this episode, we will show you few examples of how to detect and use dependencies in your CMake build system.

Finding dependencies

CMake offers a family of commands to find artifacts installed on your system:

- `find_file` to retrieve the full path to a file.
- `find_library` to find a library, shared or static.
- `find_package` to find and load settings from an external project.
- `find_path` to find the directory containing a file.
- `find_program` to find an executable.

The workhorse of dependency discovery is `find_package`, which will cover your needs in almost all use cases.

! `find_package`

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
            [REQUIRED] [[COMPONENTS] [components...]]
            [OPTIONAL_COMPONENTS components...]
            [NO_POLICY_SCOPE])
```

This command will attempt finding the package with name `<PackageName>` by searching in a number of [predefined folders](#). It is possible to ask for a minimum or exact version. If `REQUIRED` is given, a failed search will trigger a fatal error. The rules for the search are obtained from modules named `Find<PackageName>.cmake`. Packages can also have *components* and you can ask to detect just a handful of them.

We cannot stress this enough: you should **only** use the other commands in the `find_` family in very special, very narrow circumstances. Why so?

1. For a large selection of common dependencies, the `Find<PackageName>.cmake` modules shipped with CMake work flawlessly and are maintained by the CMake developers. This lifts the burden of programming your own dependency detection tricks.
2. `find_package` will set up **imported targets**: targets defined *outside* your project that you can use with your own targets. The properties on imported targets defines *usage requirements* for the dependencies. A command such as:

```
target_link_libraries(your-target
    PUBLIC
    imported-target
)
```

will set compiler flags, definitions, include directories, and link libraries from `imported-target` to `your-target` and to all other targets in your project that will use `your-target`.

These two points simplify **enormously** the burden of dependency detection and consistent usage within a multi-folder project.

Using `find_package`

When attempting dependency detection with `find_package`, you should make sure that:

- A `Find<PackageName>.cmake` module exists,
- Which components, if any, it provides, and

- What imported targets it will set up.

A complete list of `Find<PackageName>.cmake` can be found from the command-line interface:

```
$ cmake --help-module-list | grep "Find"
```

Exercise 10: Using OpenMP

We want to compile the following OpenMP sample code: ¹

```
// example adapted from
// http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf page 85

#include <cstdlib>
#include <iostream>

void long_running_task(){
    // do something
    std::cout << "long_running_task" << std::endl;
};

void loop_body(int i, int j){
    // do something
    std::cout << "i = " << i << " j = " << j << std::endl;
};

void parallel_work() {
    int i, j;
    #pragma omp taskgroup
    {
        #pragma omp task
        long_running_task(); // can execute concurrently

        #pragma omp taskloop private(j) grainsize(500) nogroup
        for (i = 0; i < 100; i++) { // can execute concurrently
            for (j = 0; j < i; j++) {
                loop_body(i, j);
            }
        }
    }
}

int main() {
    parallel_work();
    return EXIT_SUCCESS;
}
```

Note the usage of the `taskloop` construct, which was introduced in OpenMP 4.5: we need to make sure our C++ compiler is suitably compatible with *at least* that version of the standard.

From the documentation of the `FindOpenMP.cmake` module:

```
$ cmake --help-module FindOpenMP | less
```

we find that the module provides the components `C`, `CXX`, and `Fortran` and that `OpenMP::OpenMP_CXX` target will be provided, if detection is successful. The scaffold project is in `content/code/10_taskloop`. You will need to find the suitable OpenMP library and link against the imported target.

We can configure and build verbosely.² Notice that compiler flags, include directories, and link libraries are properly resolved by CMake.

You can find the complete working example in the `solution` subfolder.

Exercise 11: Using MPI

In this exercise, you will attempt compiling a “Hello, world” program that uses the message passing interface (MPI).

1. Check whether a `FindMPI.cmake` module exists in the built-in module library.
2. Get acquainted with its components and the variables and imported targets it defines.

C++

Fortran

The scaffold project is in `content/code/11_mpi-cxx`.

1. Compile the source file to an executable.
2. Link against the MPI imported target.
3. Invoke a verbose build and observe how CMake compiles and links.

A working example is in the `solution` subfolder.

Alternatives: `Config` scripts and `pkg-config`

What to do when there is no built-in `Find<PackageName>.cmake` module for a package you depend on? The package developers might be already prepared to help you out:

- They ship the CMake-specific file `<PackageName>Config.cmake` which describes how the imported target should be made for their package. In this case, you need to point CMake to the folder containing the `Config` file using the special `<PackageName>_DIR` variable:

```
$ cmake -S. -Bbuild -  
D<PackageName>_DIR=/folder/containing/<PackageName>Config.cmake
```

- They include a `.pc` file, which, on Unix-like platforms, can be detected with the `pkg-config` utility. You can then leverage `pkg-config` through CMake:

```
# find pkg-config
find_package(PkgConfig REQUIRED)
# ask pkg-config to find the UUID library and prepare an imported target
pkg_search_module(UUID REQUIRED uuid IMPORTED_TARGET)
# use the imported target
if(TARGET PkgConfig::UUID)
    message(STATUS "Found libuuid")
endif()
```

This was the strategy adopted in [Probing compilation, linking, and execution](#) when testing the use of the UUID library.

! Keypoints

- CMake has a rich ecosystem of modules for finding software dependencies. They are called `Find<package>.cmake`.
- The `Find<package>.cmake` modules are used through `find_package(<package>)`.
- You can also use the classic Unix tool `pkg-config` to find software dependencies, but this is not as robust as the CMake-native `Find<package>` modules.

Footnotes

- [1] Example adapted from page 85 in [OpenMP 4.5 examples](#).
- [2] The way in which to trigger a verbose build depends on the native build tool you are using. For Unix Makefiles: `$ cmake --build build -- VERBOSE=1`, and for Ninja: `$ cmake --build build -- -v`.

Tips and tricks using CMake

! Objectives

- Learn what tools exist to structure projects as they grow.
- Discuss the value of localizing scope and avoiding side effects.
- Recognize more maintainable and less maintainable patterns.

As projects grow, code structures get more complicated: more possibilities, more corner cases, more options to users, and more developers who are contributing and may not oversee the entire CMake structure. In this episode we will mention a couple of tools to bring some structure and flow-control into larger projects. ¹

Listing sources or globbing them

In all our examples we have listed all sources when defining targets.

In CMake, you can glob patterns (e.g. all files that end with `*.cpp`) without listing them explicitly. This is tempting, but we advise **against** doing this. The reason is that CMake cannot **guarantee** correct tracking dependency changes when you add files after you have configured. ²

Listing files explicitly also allows to `grep` for them in the CMake code to see where a modification is likely needed. This can help colleagues in our projects who are not familiar with CMake to find out where to change things.

Options and flow control

You may want to give users the possibility to decide whether they want to enable an option or not.

```
# by default this one will be ON
option(ENABLE_MPI "Configure for MPI parallelization" ON)

if(ENABLE_MPI)
    find_package(MPI REQUIRED COMPONENTS Fortran)
else()
    message(STATUS "no problem, building without MPI")
endif()
```

Now the users can decide:

```
$ cmake -S. -Bbuild -DENABLE_MPI=OFF
```

Organizing files into modules

Modules are collections of functions and macros and are either CMake- or user-defined. CMake comes with a rich ecosystem of modules and you will probably write a few of your own to encapsulate frequently used functions or macros in your CMake scripts.

You can collect related CMake-code into a file called `my_lengthy_code.cmake` and then include it in another CMake code as shown below. This can help organizing projects that are growing out of hand and separate concerns.

```
include(my_lengthy_code)
```

Variables vs. targets

In [Target-based build systems with CMake](#) we have motivated why targets are preferable over variables.

When you portion your project into modules, then variable declaration impose an order and the risk is high that somebody will not know about the implicit order and reorder modules one day and the behavior will change.

Try to minimize the use of user-defined variables. They can point to a sub-optimal solution and a better, more state-less, declarative, solution may exist.

Functions and macros

Functions and **macros** are built on top of the basic built-in commands and are either CMake- or user-defined. These prove useful to avoid repetition in your CMake scripts. The difference between a function and a macro is their **scope**:

1. Functions have their own scope: variables defined inside a function are not propagated back to the caller.
2. Macros do not have their own scope: variables from the parent scope can be modified and new variables in the parent scope can be set.

Prefer functions over macros to minimize side-effects.

Where to list sources and tests?

Some projects collect all sources in one file, all tests in another file, and carry them across in variables:

```
project/  
├─ CMakeLists.txt  
├─ cmake  
│   ├── sources.cmake  
│   ├── tests.cmake  
│   └─ definitions.cmake  
├─ external  
└─ src  
    ├── evolution  
    ├── initial  
    ├── io  
    └─ parser
```

It is recommended to organize the sources like the format below, where sources, definitions, and tests are defined in the “closest” `CMakeLists.txt` files.

```
project/
├── CMakeLists.txt
├── external
│   └── CMakeLists.txt
└── src
    ├── CMakeLists.txt
    ├── evolution
    │   └── CMakeLists.txt
    ├── initial
    │   └── CMakeLists.txt
    ├── io
    │   └── CMakeLists.txt
    └── parser
        └── CMakeLists.txt
```

The reason is that this will minimize side-effects, ordering effects, and simplify maintenance for those who want to add or rename source files: they can do it in one place, close to where they are coding.

Order and side effects

- When portioning your project into modules, design them in a way so that order does not matter (much).
- This is easier with functions than with macros, and easier with targets than with variables.
- Avoid variables with parent or global scope. Encapsulate and prefer separation of concerns.

Where to keep generated files

CMake allows us to generate files at configure- or build-time. When generating files, we recommend to **always** generate into the build folder, never outside.

The reason is that you always want to maintain the possibility to configure different builds with the same source without having to copy the entire project to a different place.

Footnotes

[1] This episode is adapted, with permission, from the [CodeRefinery CMake lesson](#).

[2] A glob would be done using the `file` command. We quote the explanation in the official documentation as to why it is generally not safe to use the `GLOB` subcommand: *If no ``CMakeLists.txt`` file changes when a source is added or removed then the generated build system cannot know when to ask CMake to regenerate. The ``CONFIGURE_DEPENDS`` flag may not work reliably on all generators, or if a new generator is added in the future that cannot support it, projects using it will be stuck. Even if ``CONFIGURE_DEPENDS`` works reliably, there is still a cost to perform the check on every rebuild.*

Additional Topics

Objectives

- Additional topics for CMake workshop.
- A short summary of what we have learned.

Additional topics

In addition to the contents covered in this [Intro to CMake](#), there are additional topics and advanced features of the CMake build system that may be of use on projects as they get larger. This is far from a comprehensive list, and information related to unlisted tasks may be found on official CMake documentation.

These topics include:

- [Creating and running tests with CTest.](#)
- [Automated dependency handling with FetchContent.](#)
- [Mixing C++ and Fortran.](#)
- [Mixing Python and compiled languages.](#)
- [Detecting your environment.](#)

Resources and books

There are many free resources online regarding CMake:

- The [CMake official documentation](#).
- The [CMake tutorial](#).
- The [HEP Software Foundation](#) training course.
- [An Introduction to Modern CMake](#).

You can also consult the following books:

- **Professional CMake: A Practical Guide** by Craig Scott.
- **CMake Cookbook** by Radovan Bast and Roberto Di Remigio. The accompanying repository is on [GitHub](#).

Summary of *Intro to CMake*

Keypoints

- CMake is a powerful cross-platform build systems generator.
- CMake provides a very good reference-style documentation.
- For larger projects you probably want to write a lightweight scaffold around CMake.

- Many projects use CMake in other words you are not alone if you look for a solution.
- [Stack Overflow](#) and [GitHub repositories](#) are a good resource for solutions.

30 min	From sources to executables
30 min	CMake syntax
40 min	Target-based build systems with CMake
30 min	Probing compilation, linking, and execution
20 min	Finding and using dependencies
10 min	Tips and tricks using CMake
10 min	Additional Topics

Quick Reference

Who is the course for?

This course is for students, researchers, engineers, and programmers that have heard of [CMake](#) and want to learn how to use it effectively with projects they are working on. This course assumes no previous experience with [CMake](#). You will have to be familiar with the tools commonly used to build software in your compiled language of choice (C/C++ or Fortran).

Specifically, this lesson assumes that participants have some prior experience with or knowledge of the following topics (but no expertise is required):

- Compiling and linking executables and libraries.
- Differences between shared and static libraries.
- Automated testing.

About this course

This lesson material is originally developed by the [EuroCC National Competence Center Sweden \(ENCCS\)](#) and taught in the [CMake Workshop](#). Each lesson episode has clearly defined learning objectives and includes multiple exercises along with solutions, and is therefore also useful for self-learning.

This material [Introduction to CMake](#) was adapted from the lesson materials for [CMake Workshop \(ENCCS\)](#) and [CMake Workshop \(CodeRefinery\)](#), and will be use for the [Build Systems Course and Hackathon](#).

This lesson material is licensed under [CC-BY-4.0](#) and can be reused in any form (with appropriate credit) in other courses and workshops. Instructors who wish to teach this lesson can refer to the [Instructor's guide](#) for practical advice.

Outreach

There are many free online resources regarding CMake:

- The [CMake official documentation](#).
- The [CMake tutorial](#).
- The [HEP Software Foundation](#) training course.
- The [Building portable code with CMake](#) from the [CodeRefinery](#).
- The [CMake Workshop](#) from the [CodeRefinery](#).

You can also consult the following books:

- **Professional CMake: A Practical Guide** by Craig Scott.
- **CMake Cookbook** by Radovan Bast and Roberto Di Remigio. The accompanying repository is on [GitHub](#)

Credits

The lesson file structure and browsing layout is inspired by and derived from the [work](#) by [CodeRefinery](#) licensed under the [MIT license](#). We have copied and adapted most of their license text.

Instructional Material

All ENCCS instructional material is made available under the [Creative Commons Attribution license \(CC-BY-4.0\)](#). The following is a human-readable summary of (and not a substitute for) the [full legal text of the CC-BY-4.0 license](#). You are free:

- to **share** - copy and redistribute the material in any medium or format;
- to **adapt** - remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow these license terms:

- **Attribution** - You must give appropriate credit (mentioning that your work is derived from work that is Copyright (c) ENCCS and, where practical, linking to <https://enccs.se>), provide a [link to the license](#), and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions** - You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. With the understanding that:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Software

The code samples and exercises in this lesson were adapted from the GitHub repository for the [CMake Cookbook](#).

Except where otherwise noted, the example programs and other software provided by ENCCS are made available under the [OSI-approved MIT license](#).