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.
- Run tests for your code with *CTest*.
- Ensure your build system will work on different platforms.
- (optional) Detect and use external dependencies in your project.
- (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 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.

We also show how you can use Binder to run in the cloud.

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 doing 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!

From sources to executables

❓ Questions

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

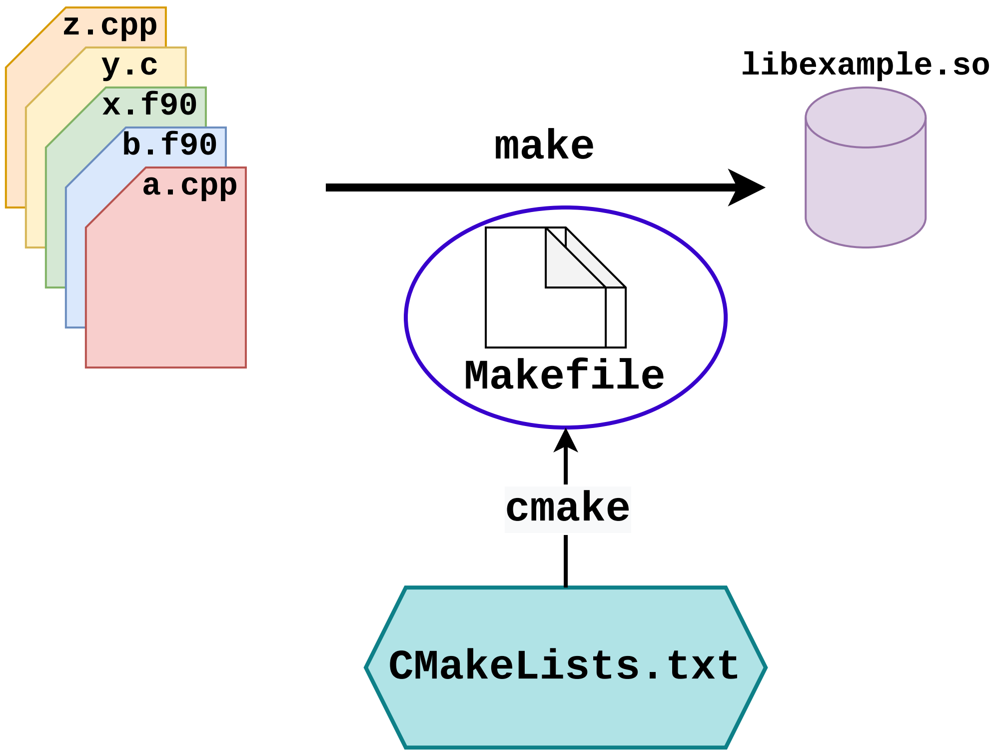
Objectives

- 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-cxx` folder.

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

int main() {
  std::cout << "Hello world" << 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 the `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.

2. 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!
3. 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 ...])
```

4. 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:

```
$ 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:

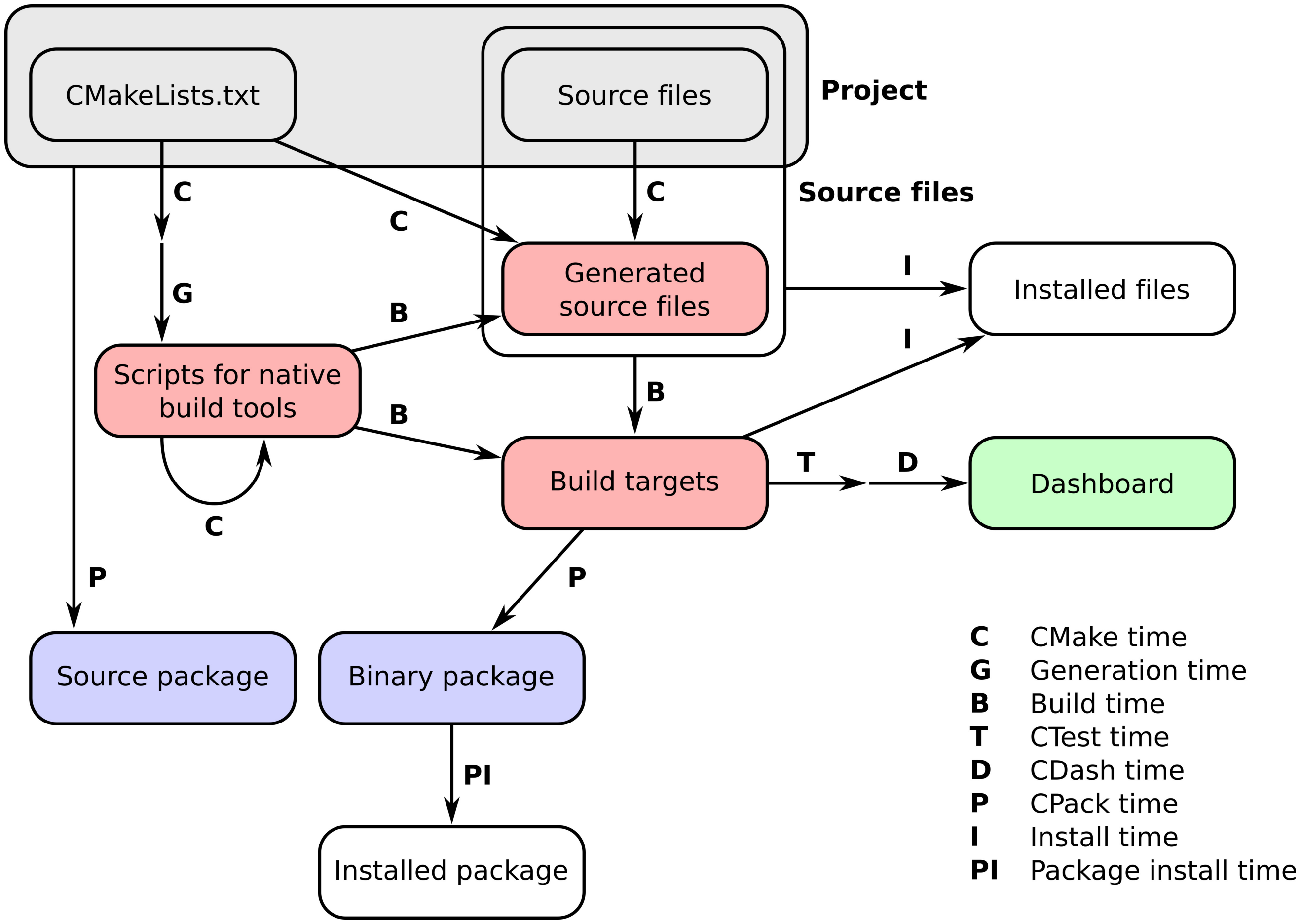
```
$ cmake --help-variable CMAKE_GENERATOR
```

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_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 encapulate 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:

```
$ tree -L 2 build

build
├── CMakeCache.txt
├── CMakeFiles
│   ├── 3.18.4
│   ├── cmake.check_cache
│   ├── CMakeDirectoryInformation.cmake
│   ├── CMakeOutput.log
│   ├── CMakeTmp
│   ├── compute-areas.dir
│   ├── geometry.dir
│   ├── Makefile2
│   ├── Makefile.cmake
│   ├── 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 more 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 work with projects spanning multiple folders.
- Learn how to handle multiple targets in one project.

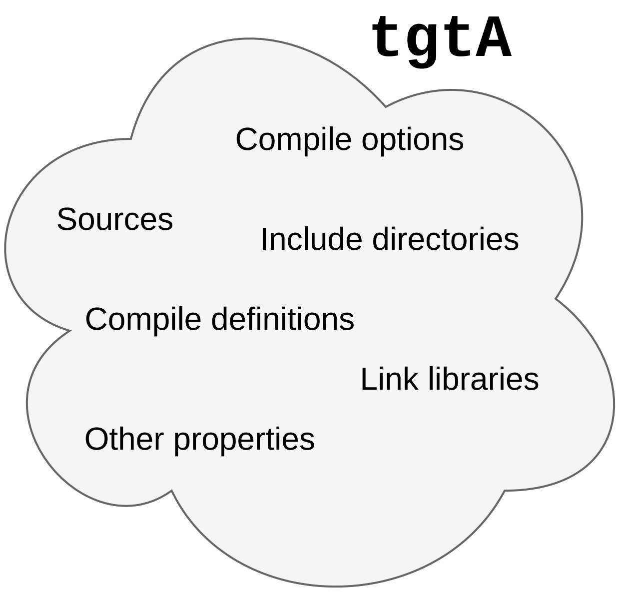
Real-world projects require more than compiling a few source files into executables and/or libraries. In most cases, you will come to projects comprising hundreds of source files sprawling in a complex source tree.

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:

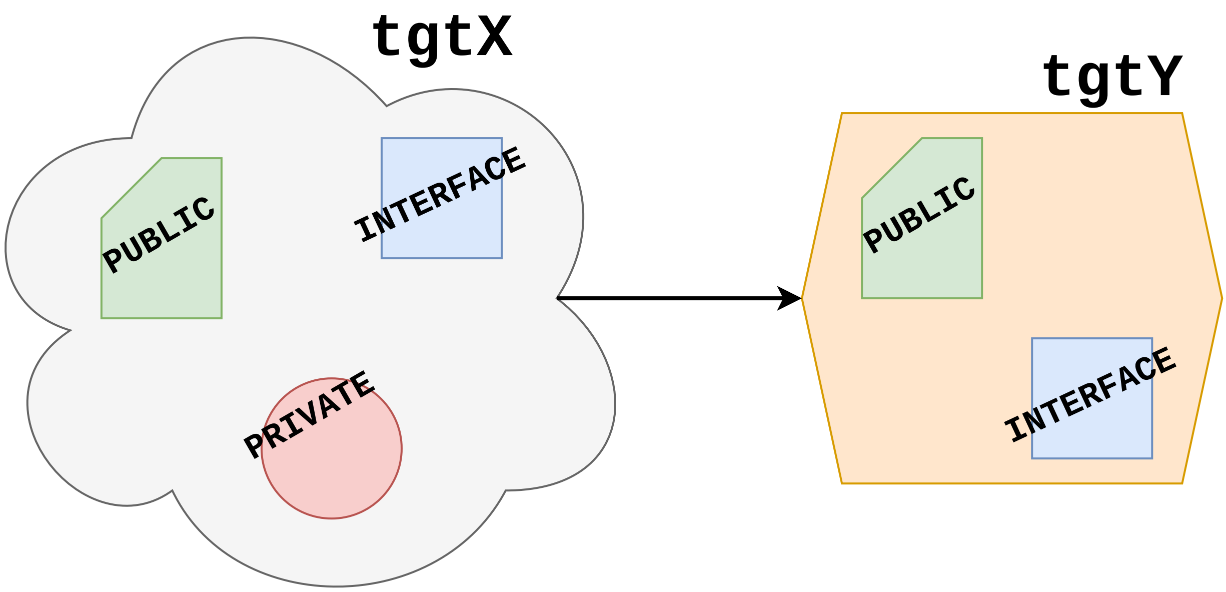
```
$ cmake --help-command-list | grep "^target_"

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.*

Understanding visibility levels

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

In this demo, we split the source code into 3 libraries and all files are available in the `content/code/xx_visibility-levels/` folder.

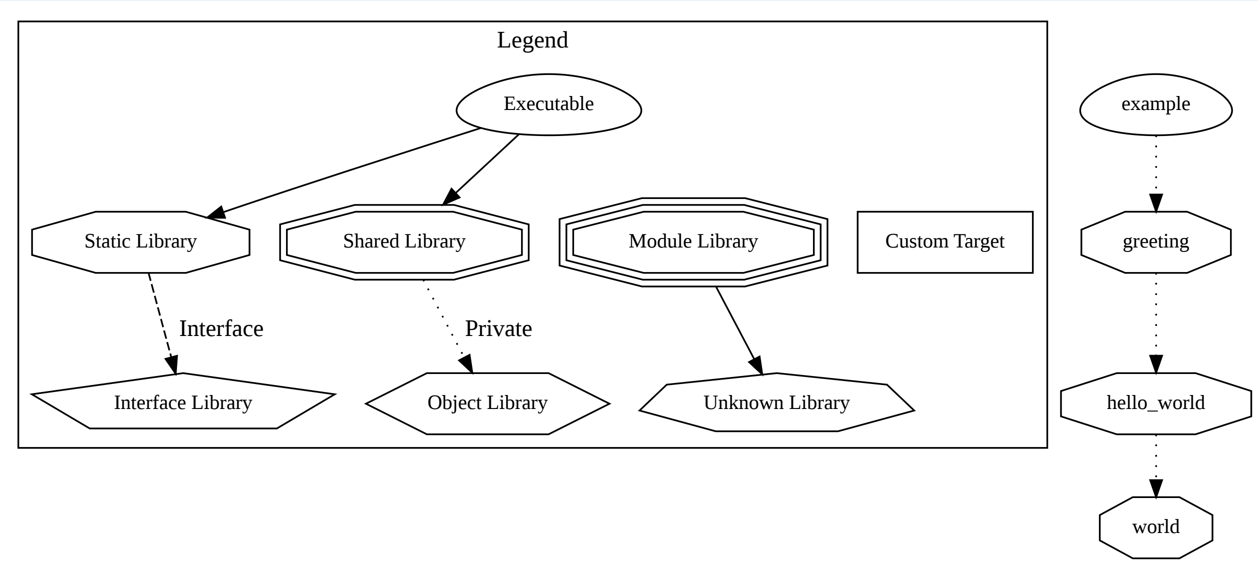
```
.
├── CMakeLists.txt
├── greeting
│   ├── greeting.cpp
│   └── greeting.hpp
├── hello_world
│   ├── hello_world.cpp
│   └── hello_world.hpp
├── main.cpp
└── world
        ├── world.cpp
        └── world.hpp
```

In this source code, the main function links to greeting which links to hello_world which links to world.

The internal dependency tree

If you have installed `Graphviz`, you can visualize the dependencies between these targets:

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



The dependencies between the four targets in the code example.

Take a look at the `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.14)

project(example LANGUAGES CXX)


add_library(world)
target_sources(world
  PUBLIC
    world/world.hpp
  PRIVATE
    world/world.cpp
  )
target_include_directories(world
  PUBLIC
    world
  )
# target_compile_definitions(world PRIVATE "MY_DEFINITION")


add_library(hello_world)
target_sources(hello_world
  PUBLIC
    hello_world/hello_world.hpp
  PRIVATE
    hello_world/hello_world.cpp
  )
target_include_directories(hello_world
  PUBLIC
    hello_world
  )

# hello_world depends on world
target_link_libraries(hello_world PRIVATE world)


add_library(greeting)
target_sources(greeting
  PUBLIC
    greeting/greeting.hpp
  PRIVATE
    greeting/greeting.cpp
  )
target_include_directories(greeting
  PUBLIC
    greeting
  )

# greeting depends on hello_world
target_link_libraries(greeting PRIVATE hello_world)


add_executable(example main.cpp)

# example depends on greeting
target_link_libraries(example PRIVATE greeting)
```

✍️ Testing the 3 different visibility levels

1. Browse, configure, build, and run the code.
2. Uncomment the highlighted line (line 17) with `target_compile_definitions`, configure into a fresh folder, and build using the commands below. You will see that the definition is used in `world.cpp` but nowhere else.

```
$ cmake -S. -Bbuild_private
$ cmake --build build_private
```

3. Change the definition to `PUBLIC`, configure into a fresh folder, and build. You will see that the definition is used both in `world.cpp` and `hello_world.cpp`.

```
$ cmake -S. -Bbuild_public
$ cmake --build build_public
```

4. Then change the definition to `INTERFACE`, configure into a fresh folder, and build. You will see that the definition is used only in `hello_world.cpp` but not in `world.cpp`.

```
$ cmake -S. -Bbuild_interface
$ cmake --build build_interface
```

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

20 min	From sources to executables
30 min	CMake syntax
40 min	hello-ctest
40 min	probing
40 min	Target-based build systems with CMake
35 min	python-bindings
20 min	tips-and-tricks

Quick Reference

Instructor's guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

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 material used for CMake Workshop and will be use for the Build Systems Course and Hackathon.

The 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.

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;

Software

The code samples and exercises in this lesson were adapted from the GitHub repository for the CMake Cookbook.