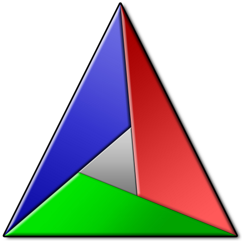
An introduction to CMake

Incontri Informatici MOX 2014



MOX - Politecnico di Milano 01/21/2014 Nicholas Tarabelloni

INTRODUCTION

What and Why?

What?

- **CMake** is a *cross-platform*, *open-source* **build system** designed to build, test and package software, and to control the software compilation process using simple platform and *compiler independent* configuration files.
- **CMake** can find dependencies and perform conditional actions, such as set up flags and compiler options or specify the type of build desired by the user.
- **CMake** uses *generators* to translate the platform-independent instructions into build structures specifically intended for the platform at hand.

What and Why?

What?

- **CMake** is a *cross-platform*, *open-source* **build system** designed to build, test and package software, and to control the software compilation process using simple platform and *compiler independent* configuration files.
- **CMake** can find dependencies and perform conditional actions, such as set up flags and compiler options or specify the type of build desired by the user.
- **CMake** uses *generators* to translate the platform-independent instructions into build structures specifically intended for the platform at hand.

Why?

- **CMake** was created by Kitware in response to the need for a powerful, cross-platform build environment for open-source projects such as ITK and VTK.
- **CMake** is useful to generalize almost effortlessly the build procedure of ongoing projects to other platforms.
- The use of **CMake** allows a rapid and easy diffusion of small-sized up to very complex software packages with, in principle, none or little adaptations to the specific machine in use.
- **CMake** is mature and stable, it easily supports the integration of external libraries into existing projects through ad-hoc configuration files.

Where and How?

Where & How?

Kitware maintains a public website (`www.cmake.org`) about **CMake**, which should be considered as a first reference for users. It contains:

- Details about the project (development, participants, history, etc.)
- Binaries or source code of recent releases, ready for download.
- A first (quite raw but whole) documentation of **CMake**'s instructions in form of a list.
- Pointers to several support resources, spanning from a Wiki to books, from courses to Kitware consulting (mind that some of these may not be free).
- Pointers to other open-source projects housed at Kitware (VTK, Paraview, CDash, BatchMake etc.).

Where and How?

Where & How?

Kitware maintains a public website (`www.cmake.org`) about **CMake**, which should be considered as a first reference for users. It contains:

- Details about the project (development, participants, history, etc.)
- Binaries or source code of recent releases, ready for download.
- A first (quite raw but whole) documentation of **CMake**'s instructions in form of a list.
- Pointers to several support resources, spanning from a Wiki to books, from courses to Kitware consulting (mind that some of these may not be free).
- Pointers to other open-source projects housed at Kitware (VTK, Paraview, CDash, BatchMake etc.).

Still need more help?
For practical issues try also Stack Overflow, a true trouble-shooting weapon. If any problem remains, Google it: there's a bunch of sparse documentation and tutorial all around the web.

Dramatis Personae

CMakeLists.txt:

To cut it short, **CMake**'s action is controlled through the instructions contained into files conventionally named `CMakeLists.txt`. The syntax of these files reminds of `shell` and `Makefile` languages.

`CMakeLists.txt` files are used to (for instance):

- define project's properties
- set up build options,
- set up variables,
- set up and configure dependencies (such as external libraries),
- declare subfolders (project's architecture),
- target executables, libraries, tests, ...

Dramatis Personae

CMakeLists.txt:

To cut it short, CMake's action is controlled through the instructions contained into files conventionally named `CMakeLists.txt`. The syntax of these files reminds of `shell` and `Makefile` languages.

`CMakeLists.txt` files are used to (for instance):

- define project's properties
- set up build options,
- set up variables,
- set up and configure dependencies (such as external libraries),
- declare subfolders (project's architecture),
- target executables, libraries, tests, . . .

Source directory:

It contains our hard-won source code and header files making up the project.

- Each directory contains a `CMakeLists.txt` file (and those who don't and/or are not added in the build tree with the command ADD_SUBDIRECTORY are not considered).
- The CMake's variable CMAKE_SOURCE_DIR specifies the source directory's path.

Dramatis Personae

Build Directory:

There are two kinds of builds available:

- **in-source**: The build is done inside the source tree.

  Disadvantages: the source directory is messed-up with the (many) files generated during the build procedure, moreover I may want different build versions (debug, release, ...).

- **out-of-source**: the build directory is a separate environment in which **CMake** writes the output of the build procedure.

The build directory is accessible through the variable **CMAKE_BINARY_DIR**.
At the top level it contains a `CMakeCache.txt` file, in which the variables defined by **CMake** during the build procedure are listed (and can be modified - we will see this later on).

Dramatis Personae

Build Directory:

There are two kinds of builds available:

- **in-source**: The build is done inside the source tree.

  Disadvantages: the source directory is messed-up with the (many) files generated during the build procedure, moreover I may want different build versions (debug, release, . . .).

- **out-of-source**: the build directory is a separate environment in which **CMake** writes the output of the build procedure.

The build directory is accessible through the variable **CMAKE_BINARY_DIR**.
At the top level it contains a `CMakeCache.txt` file, in which the variables defined by **CMake** during the build procedure are listed (and can be modified - we will see this later on).

Install Directory:

This is the directory where the executables will be installed by typing “`make install`”.
It is set with the **CMake** variable **CMAKE_INSTALL_PREFIX**.

One day in the life of CMake

A typical and very basic way to build a project with **CMake** on a Linux system (the essentials):

1. Write your own supercool code.
2. Add a **CMake** support structure.
3. "`mkdir build-dir`" inside some directory (out-of-source build is supposed).
4. "`cd /path/to/build-dir && cmake /path/to/source-dir`".
5. Wait for the build to end.
6. Possible modification of build variables, e.g. with `ccmake` (see later on).
7. Ready to compile: "`make -j4`".
8. @#*&! (return to **point 1**)

BASIC TOPICS

How to start-up a new project

To start-up a new project a first CMakeLists.txt file must be created in the top folder of the source tree.

Mind that each subdirectory making up the project must be added properly through the command ADD_SUBDIRECTORY() and has to contain a CMakeLists.txt file.

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.6 ) # Minimum version supporting
  # the following instructions, this setup is up to you!

PROJECT(Pippo) #Declaring the project's name

SET(PROJECT_MAJOR_VERSION 1)  # Setup the major version number
SET(PROJECT_MINOR_VERSION 0) # Setup the minor version number
SET(PROJECT_VERSION
  ${PROJECT_MAJOR_VERSION}.${PROJECT_MINOR_VERSION})

# Adding a subdirectory to the source tree
ADD_SUBDIRECTORY(Pluto)

MESSAGE( "this is my first CMakeLists.txt file" )
```

How to setup variables

It is often the case that a variable must be created in order to store build information. To this aim **CMake** provides an ad-hoc instruction: **SET**:

```
SET( <variable> <value> [[ CACHE <type> <docstring> [FORCE]] | PARENT SCOPE] )
```

```
SET( MY_VARIABLE 123 )

SET( MY_OTHER_VARIABLE ${MY_VARIABLE})

SET( MY_STRING ‘‘Hello World!’’ )
MESSAGE( ${MY_STRING} )

#If you don’t specify a value CMake won’t
# define/will un-define the variable
SET( MY_FLAG)
#This is much less obfuscated-CMake-style
UNSET( X )

#More complete usage of SET
SET( MY_FLAG TRUE)
SET( MY_FLAG TRUE CACHE BOOL ‘‘This is a flag of mine’’ )
SET( MY_FLAG TRUE CACHE BOOL ‘‘This is a flag of mine’’ FORCE )
```

SET in details

Let's understand the non-standard features of SET step-by-step.

1.
```
SET( MY_FLAG TRUE)
```

- If the variable already existed in the `CMakeCache.txt`, that value is shadowed.
- This instruction never writes to `CMakeCache.txt`, thus a possible pre-existing value is kept.

2.
```
SET( MY_FLAG TRUE CACHE BOOL ‘‘This is a flag of mine’’ )
```

- If cache contains a value for "MY_FLAG", this is preferred to the value we are specifying.
- If not previously contained inside Cache, the variable is registered with the specified documentation string.
- The `Type` information is used by the CMake GUI to show up a widget. It can be one among FILEPATH, PATH, STRING, BOOL, INTERNAL.
- When the type is INTERNAL any cache value is overwritten.

3.
```
SET( MY_FLAG TRUE CACHE BOOL ‘‘This is a flag of mine’’ FORCE )
```

- The FORCE option overwrites the Cache value removing any changes by the user.

Some useful variables

There is a set of built-in variables which are useful to query or modify while writing/executing a **CMake** build. Some of them are the following (refer to this **CMake** wiki address for a longer list):

- CMAKE_COMMAND: It contains the path to the current **CMake** executable being run.
- CMAKE_CURRENT_BINARY_DIR: this is the directory where the output files of the current `CMakeLists.txt` will go to. For in-source builds this is the same as CMAKE_CURRENT_SOURCE_DIR.
- CMAKE_CURRENT_SOURCE_DIR: this is the directory where the currently processed `CMakeLists.txt` is located in.
- ENV{name}: this is not an environment variable, but this is how environment variables can be accessed by **CMake**.
- CMAKE_INCLUDE_PATH: this is used when searching for include files e.g. using the FIND_PATH() command. If you need several directories, separate them by the platform specific separators (e.g. ":" on UNIX)
- CMAKE_LIBRARY_PATH: this is used when searching for libraries e.g. using the FIND_LIBRARY() command.
- CMAKE_BUILD_TYPE: A variable which controls the type of build.
- BUILD_SHARED_LIBS: if this is set to ON, then all libraries are built as shared libraries by default.

Implicitly created variables

Some common variables are often detected and set during the first **CMake** run, then are written to the cache without the need of specific setup in the `CMakeLists.txt` files. Among those, are quite important:

- CMAKE_C_COMPILER: the compiler used for C files. **IT CAN NOT** be changed after the first **CMake** run, though you might think you have done it.
- CMAKE_CXX_COMPILER: the compiler used for C++ files (as CMAKE_C_COMPILER, it cannot be modified after the first **CMake** run).
- CMAKE_C_FLAGS_*: the compiler flags for compiling C sources for the build type *.
- CMAKE_CXX_FLAGS_*: the compiler flags for compiling C++ sources for the build type *.
- CMAKE_AR: the tool for creating libraries.
- CMAKE_INSTALL_PREFIX: the directory for the installation.
- CMAKE_LINKER: the linker used for the compilation.
- CMAKE_VERBOSE_MAKEFILE: to setup the verbosity during compilation.
- CMAKE_COLOR_MAKEFILE: it may enable a colored makefile.

How to modify variables in cache

- You can use **ccmake**, both during and after a first build.
  Like **CMake**, ccmake is a console program with a minimal interface allowing the setup of variables defined during the build procedure. A typical procedure is:
  - At the first run, launch **ccmake**:
    ```
    ccmake /path/to/source/dir
    ```
  - Configure (use the apposite key).
  - Modify variables to the desired values (you may want to toggle the advanced mode for the full list).
  - Generate and exit.
- You can use **cmake-gui**, a GUI for the control of the entire build procedure. The typical use is the following:
  - Launch **cmake-gui**:
    ```
    cmake-gui /path/to/source/dir
    ```
  - Press the configure button. When prompted choose generator and other options (native compilers etc.).
  - After the configuration, modify properly the variables.
  - Generate and exit.

Conditional Blocks

CMake supports a simple syntax for conditional blocks. If combined with other specific instructions, they can lead to quite sophisticated behaviours.

The corresponding instructions are:

```
IF( EXPRESSION) ELSE(EXPRESSION) ENDIF(EXPRESSION)
```

An utterly meaningless usage example is:

```
IF(EXPRESSION1)
    # then section.
    MESSAGE( ‘‘PIPPO’’)
    MESSAGE( ‘‘PLUTO’’)
    #...
  ELSEIF(EXPRESSION2)
    # elseif section.
    SET(PAPERINO ‘‘PAPERINO’’)
  ELSE(EXPRESSION1)
    # else section.
    SET( PAPEROGA ‘‘PAPEROGA’’)
    #...
ENDIF(EXPRESSION1)
```

Tips & tricks on conditional blocks

```
if(DEFINED variable) #True if the variable is defined

if(EXISTS ${FILENAME})
if(NOTEXISTS ${FILENAME})

if(file1 IS_NEWER_THAN file2)
if(IS_DIRECTORY ${DIR_NAME}) #True if the given name is a directory.
if(IS_ABSOLUTE ${PATH}) #True if the given path is an absolute path.

if(NOT EXPRESSION) # Logical negation

if(EXPRESSION1 AND EXPRESSION2) #Logical conjunction
if(EXPRESSION1 OR EXPRESSION2) #Logical disjunction

# True if the given string or variable's value
# matches the given regular expression.
if(VARIABLE MATCHES REGEX)
if(STRING MATCHES REGEX)

#True if the given string or variable's value is a valid number
# and the inequality or equality is true.
if(VARIABLE LESS NUMBER)
if(STRING LESS  NUMBER)
#...
```

Loops

CMake provides also two set of instructions to perform loops of commands: `FOREACH` and `WHILE`.

```
FOREACH( VAR ARG1 ARG2 ...)
    COMMAND1
    COMMAND2
    ...
ENDFOREACH(VAR)
```

```
FOREACH( VAR RANGE START END
[STEP])
    COMMAND1
    COMMAND2
    ...
ENDFOREACH(VAR)
```

The commands enclosed inside the for loop are recorded without being invoked. Once the `ENDFOREACH` is evaluated, they are invoked on `VAR`, set to the current argument value listed in `FOREACH`.
The `WHILE` instruction is similar in its use: it evaluates a group of commands while a condition is true.

```
WHILE( CONDITION )
    COMMAND1
    COMMAND2
...
ENDWHILE( CONDITION )
```

The execution of commands follows the same steps as in `FOREACH`. Here `CONDITION` is evaluated using the same logic as the if command.

How to add an executable

A most important feature of build systems is to setup targets for the generation of executables. To this aim, **CMake** provides the command:

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

- `<name>` is the name of the generated executable and must be globally unique in the project. The actual name of the executable is built corresponding to the conventions of the system in use (e.g. `*.exe` under Windows).
- The executable file will be created in the build directory hierarchically corresponding to the source folder containing the (`CMakeLists.txt` that contains) the `ADD_EXECUTABLE` call.
- `EXCLUDE_FROM_ALL` is an option causing the target to be exclude from the `all` build target.
  1. This is useful, for example, when the target is related to a section of code not strictly necessary to the whole project, i.e. examples.
  2. In order to build it, the target must be explicitly addressed.
- Example of use:

```
ADD_EXECUTABLE( pippo.exe main.cpp source1.cpp source2.cpp )
```

How to create libraries

Targets associated to the construction of libraries can be defined with the command:

```
ADD_LIBRARY(<name> [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL]
                  source1 source2 ...  sourceN)
```

- `<name>` is the name of the target, and must be globally unique inside the project. The actual name of the library built is determined depending on the conventions of the platform at hand (such as, under Linux systems, `lib<name>.a`).
- `STATIC`, `SHARED`, or `MODULE` options specify the kind of library desired.
  1. `STATIC` defines a static library, i.e. an archive of object files to be statically linked to other targets.
  2. `SHARED` defines shared libraries, dynamically linked and loaded at runtime.
  3. `MODULE` defines libraries which are plugins, not linked to targets but dynamically loaded at runtime.

  If no type is given explicitly, the type is `STATIC` or `SHARED` based on whether the current value of the variable `BUILD_SHARED_LIBS` is `TRUE`.
- If not specified otherwise, the library output will be created in the build tree directory corresponding to the source tree directory in which the command was invoked.
- Example:

```
ADD_LIBRARY( disney STATIC ${DISNEY_SRCS} )
```

Linking an executable against a library

When developing large projects it is better to collect homogeneous compiled source files into libraries, and then to link executable against them. Clearly, this implies a dependency between the two targets (automatically set up by **CMake**).

Once a library has been created, **CMake** allows to link executable against it through the command:

```
TARGET_LINK_LIBRARIES( <target> [item1 [item2 [...]]]
      [[debug|optimized|general] <item>] ...  )
```

- `<target>` must be a valid name for an executable created in the current folder (the target of an `ADD_EXECUTABLE`).
- Keywords `debug`, `optimized`, or `general` indicate for which build configuration the following library must be used for the linking.
- Example:

```
ADD_EXECUTABLE( pippo.exe main.cpp )
ADD_LIBRARY( disney STATIC ${DISNEY_SRCS} )
TARGET_LINK_LIBRARIES( pippo.exe disney )
```

Remark on executables handling:
In order to add a desired directory to the include path, use the command `INCLUDE_DIRECTORIES()`.

Installing package's targets

The instruction appointed to setup the installation of the targets, once they have been configured and built, is INSTALL. This instruction is very rich, and offers different signatures to accomplish different behaviours.

Let's see a quite basic usage of INSTALL:

```
SET( CMAKE_INSTALL_PREFIX ${CMAKE_SOURCE_DIR}/../install
CACHE PATH "" FORCE)

SET(INSTALL_BIN bin CACHE PATH "")
SET(INSTALL_STATIC ${CMAKE_INSTALL_PREFIX}/lib/static CACHE PATH "")
SET(INSTALL_SHARED ${CMAKE_INSTALL_PREFIX}/lib/shared CACHE PATH "")
SET(INCLUDE ${CMAKE_INSTALL_PREFIX}/include CACHE PATH "")

INSTALL( TARGETS paperino.exe disneyShared disneyStatic
  RUNTIME DESTINATION ${INSTALL_BIN}
  ARCHIVE DESTINATION ${INSTALL_STATIC}
  LIBRARY DESTINATION ${INSTALL_SHARED} )
INSTALL( FILES disney.h DESTINATION ${INCLUDE} )
```

- A PERMISSION option allows to specify permissions for the installed files.
- CONFIGURATIONS argument specifies a list of build configurations for which the install rule applies (Debug, Release, etc.).
- Many other options are available to shape INSTALL's behaviour, see the reference documentation for details.

ADVANCED TOPICS

Working with external libraries

No matter how good our code is, it is simply inconceivable to develop a project without using external code.

Thus, a burning issue is to learn how to search, find and configure third-party libraries on the current system that are necessary to the project.

To this aim, **CMake** offers the command `FIND_PACKAGE()` which, in principle, renders this task almost automatic:

```
FIND_PACKAGE( <package> [version] [EXACT] [QUIET] [MODULE]
              [[REQUIRED|COMPONENTS] [components]]
              [NO_POLICY_SCOPE])
```

- It finds and loads settings for a external packages.
- `<package>_FOUND` will be set to indicate whether the package was found.
- If the package is found, ad-hoc variables are defined providing package-specific information.
- The `EXACT` option forces the search to the specified version.
- The `QUIET` option disables messages if the package cannot be found.
- The `REQUIRED` option stops processing with an error message if the package cannot be found.

Find_Package()

Warning: FIND_PACKAGE() can be used in two ways:

- MODULE mode, in which a file FindPackage.cmake is searched and executed
- CONFIG mode, in which a configuration file provided by the package is sought.

Two simple examples of FIND_PACKAGE()'s use are:

```
# Specifying a version and requiring components
FIND_PACKAGE( DISNEY 1.1.0 REQUIRED )

# Typically a variable is created to store
# the include directories
FIND_PACKAGE( DISNEY )

IF(DISNEY_FOUND)
MESSAGE( "I've found Disney at" ${DISNEY_DIR})
INCLUDE_DIRECTORIES( ${DISNEY_INCLUDE_DIRS} )
ELSE(DISNEY_FOUND)
 MESSAGE( "Disney not found!" )
ENDIF(DISNEY_FOUND)
```

Concerning `FIND_PACKAGE()`'s behaviour in `MODULE` mode, remind that:

- It searches, reads and executes a corresponding `FindPackage.cmake` file.
- `FindPackage.cmake` files report a list of commands that, typically,
  - search default locations for the installed files of the package,
  - check the version,
  - set up variables for the path to the include directory, flags . . .

Each `FindPackage.cmake` behaves in its own way, but usually contains a brief description of what it's going to do and which variables will be defined, so check their heads for specific information.

- By default the `FindPackage.cmake` files are searched into `<cmakeroot>/Modules`
- **CMake** comes along with a bunch of modules intended to support a variety of common used software, such as Lapack, BLAS, Eigen, OpenMP, MPI, CUDA, LaTeX, Doxygen . . .

Mind that support for software through proper `FindPackage.cmake` files is officially added from version to version, so check backward compatibility of your `CMakeLists.txt`.

- If you want to modify or write a new `FindPackage.cmake` file, modify the variable `CMAKE_MODULE_PATH` to its location.

Example: Boost and MPI

```
# Finding Boost
FIND_PACKAGE(Boost)
INCLUDE_DIRECTORIES(${Boost_INCLUDE_DIR})

# Finding MPI
# If the first run fails or detects an
# undesired implementation of MPI, try
SET( MPI_COMPILER "/opt/openmpi/bin/mpicxx" )
# If CMake still fails to configure properly,
# set these to circumvent autodetection entirely
SET( MPI_LIBRARY "MPI_LIBRARY-NOTFOUND")
SET( MPI_INCLUDE_PATH "/opt/openmpi/include" )

FIND_PACKAGE(MPI REQUIRED)

INCLUDE_DIRECTORIES(${MPI_INCLUDE_PATH})
```

If you want to run a parallel executable, after having linked the target with the libraries contained in `CMAKE_MPI_LIBRARIES`, use:

```
${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} PROCS ${MPIEXEC_PREFLAGS} EXECUTABLE
                         ${MPIEXEC_POSTFLAGS} ARG
```

Example: CUDA

In the last distributions of **CMake** CUDA projects are supported through a FindCUDA.cmake file.

The script tries to detect the CUDA-toolkit based on the location of nvcc. If this fails, the user has to specify a CUDA_TOOLKIT_ROOT_DIR.

Some of the variables created during the configuration are:

- CUDA_HOST_COMPILER, storing the host C compiler.
- CUDA_INCLUDE_DIRS, i.e. the include directory for cuda headers.
- CUDA_LIBRARIES, containing the runtime libraries.
- CUDA_CUBLAS_LIBRARIES, specifying the location of cublas libraries.

Among the commands defined, instead, there are:

- CUDA_ADD_EXECUTABLE, with signature analogous to the standard one, creates an executable which is made up of the files specified.
- CUDA_ADD_LIBRARY, which creates a library from a set of source files.
- CUDA_INCLUDE_DIRECTORIES, used to set up directories which will be passed to nvcc.
- CUDA_COMPILE, which directly calls the compiler on the given source files.
- CUDA_WRAP_SRCS, called by the others, acts separately on cuda and standard files to create PTX or linkable objects.

Importing CMake code

Similar, but more general than FIND_PACKAGE(), the command INCLUDE allows to *import* an external piece of CMake's code into the current CMakeLists.txt, and to execute it.

```
INCLUDE( <file|module> [OPTIONAL] [RESULT_VARIABLE <VAR>]
        [NO_POLICY_SCOPE])
```

- This commands loads and runs the code of the specified filename
- If OPTIONAL is present, then no error is raised if the file does not exist.
- If RESULT_VARIABLE is given the variable will be set to the full filename which has been included or NOTFOUND if it failed.
- If a module is specified instead of a file, the file with name <modulename>.cmake is searched first in CMAKE_MODULE_PATH, then in the CMake module directory.
- In principle, no use of this command should be done within a standard project but for very special purposes.

A simple test configuration

A great feature of **CMake** is the possibility to setup tests from project's executables. Under the skin, the tool used to actually perform tests is **CTest**.

- As a first step, the command `ENABLE_TESTING()` must be added to `CMakeLists.txt` before any source folder containing test targets.
- Next, in the source folders, use the command `ADD_TEST(...)`:

```
ADD_TEST(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
    [WORKING_DIRECTORY dir]
    COMMAND <command> [arg1 [arg2 ...]])
```

  For example:

```
ADD_TEST( disney paperino.exe qui quo qua )
```

- `ENABLE_TESTING` adds the target `test` to the generator's output (i.e. a `Makefile` on Unix), so to launch the test it is sufficient to give:

```
make test # or
ctest
```

- In principle, tests' results may be collected and elaborated (see documentation of **CDash**

How to typeset CMake?

Introducing a **CMake** code into a LaTeX document requires a proper syntax highlighting. I found it very helpful to install and use `minted`, a package that facilitates expressive syntax highlighting in LaTeXfor a great variety of languages, exploiting the additional software `Pygments`.

In order to use `minted`:

- Install both `Pygments` and `minted` (see the reference documentation on `CTAN`).
- Modify your LaTeX example similarly to the following example:

```
\usepackage{minted}
\newminted{cmake}{linenos=true,bgcolor=whitesmoke}

\begin{document}
\begin{cmakecode}
...
\end{cmakecode}
\end{document}
```

- To build the document, you must tell the compiler to enable calls to "outer world", giving:

```
pdflatex -shell-escape pippo.tex #Also latex, xelatex, ...
```

- In order to discover other languages supported by `Pygments`, type in your Linux system

```
pygmentize -L lexers
```