Building projects with CMake

Victor Eijkhout

Fall 2025

Justification

CMake is a portable build system that is becoming a *de facto* standard for C++ package management.
Also usable with C and Fortran.

Table of contents

Help! This software uses CMake!

What are we talking here?

- You have downloaded a library
- It contains a file `CMakeLists.txt`
- $\Rightarrow$ you need to install it with CMake.
- . . . and then figure out how to use it in your code.

Building with CMake

Use CMake for the configure stage, then make:

```
mkdir build
cd build
cmake /home/your/software/package ## source location
make
```

Often:

```
cd some_package
ls CMakeLists.txt # make sure it exists
mkdir build
cd build
cmake ..
make
```

Now you have binaries and such in the `build` directory.

Building and install

- Use CMake for the configure stage, then make:

```
cmake -D CMAKE_INSTALL_PREFIX=/home/yourname/packages \
    /home/your/software/package ## source location
make
make install
```

or

- do everything with CMake:

```
cmake ## arguments
cmake --build ## stuff
cmake --install ## stuff
```

We focus on the first option; the second one is portable to non-Unix environments.

What does this buy you?

1. The source directory is untouched
2. The build directory contains all temporaries
3. Your install directory (as specified to CMake) now contains executables, libraries, headers etc.
   You can add these to `$PATH`, compiler options, `$LD_LIBRARY_PATH`.
   But see later . . .

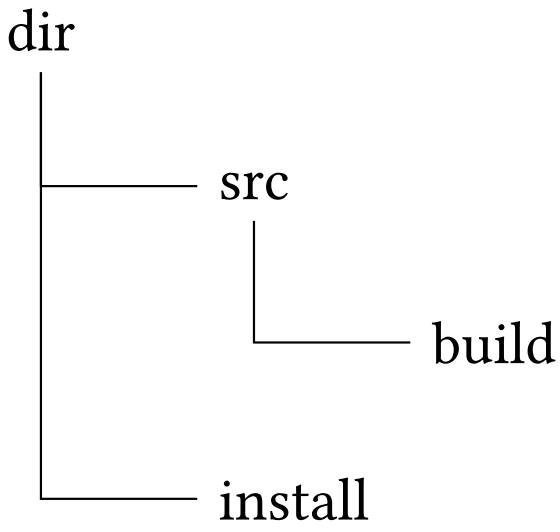
The build/make cycle

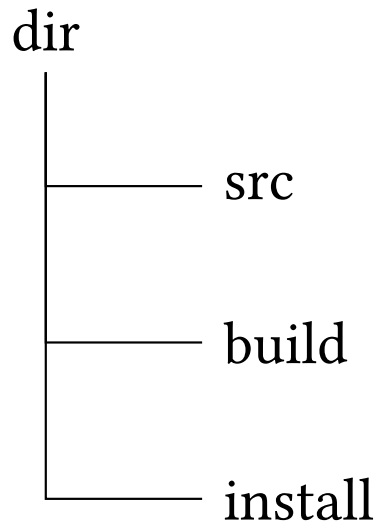
CMake creates makefiles;
makefiles ensure minimal required compilation

```
1    cmake            ## make the makefiles
2    make             ## compile your project
3    emacs onefile.c ## edit
4    make             ## minimal recompile
5
```

Only if you add (include) files do you rerun CMake.

Directory structure: two options





- In-source build: pretty common
- Out-of-source build: cleaner because never touches the source tree
- Some people skip the install step, and use everything from the build directory.

Out-of-source build: preferred

- Work from a build directory
- Specify prefix and location of `CMakeLists.txt`

```
ls some_package_1.0.0                 # we are outside the source
ls some_package_1.0.0/CMakeLists.txt # source contains cmake file
mkdir builddir                        # location for temporaries
cd builddir                           # goto build location
cmake -D CMAKE_INSTALL_PREFIX=../installdir \
      ../some_package_1.0.0           # cmake invocation
make                                  # make all tmp data in build loc
make install                          # move only final products
```

Example: eigen

Download from
`https://eigen.tuxfamily.org/index.php`
and install.
What compiler is it finding? If you are at TACC, is it the module you have loaded?

Help! I want to write CMake myself!

What are we talking here?

You have a code that you want to distribute in source form for easy installation.
You decide to use CMake for portability.
You think that using CMake might make life easier, at least for your users.
$\Rightarrow$ To do: write the `CMakeLists.txt` file.

The CMakeLists file

```
1 cmake_minimum_required( VERSION 3.20 )
2 project( myproject VERSION 1.0 )
```

- Which cmake version is needed for this file?
  (CMake has undergone quite some evolution!)
- Give a name to your project.
- Maybe pick a language.
  C and C++ available by default, or:

  ```
  1 enable_language(Fortran)
  ```

  (list: *C*, *CXX*, *CSharp*, *CUDA*, *OBJC*, *OBJCXX*, *Fortran*, *HIP*, *ISPC*, *Swift*, and a couple of variants of *ASM*)

Target philosophy

- Declare a target: something that needs to be built, and specify what is needed for it

```
1 add_executable( myprogram )
2 target_sources( myprogram PRIVATE program.cxx )
```

  Use of macros:

```
1 add_executable( ${PROJECT_NAME} )
```

- Do things with the target, for instance state where it is to be installed:

```
1 install( TARGETS myprogram DESTINATION . )
```

  relative to the prefix location.

Example: single source

Build an executable from a single source file:

```
1 cmake_minimum_required( VERSION 3.13 )
2 project( singleprogram VERSION 1.0 )
3
4 add_executable( program )
5 target_sources( program PRIVATE program.cxx )
6 install( TARGETS program DESTINATION . )
```

Deprecated usage

Possible usage, but deprecated:

```
1 add_executable( myprogram myprogram.c myprogram.h )
```

As much as possible use ‘target’ design:

```
1 add_executable( program )
2 target_sources( program PRIVATE program.cxx )
```

C++ example

```
1 cmake_minimum_required( VERSION 3.20 )
2 project( homework VERSION 1.0 )
3
4 message( "Using sources: homework.cpp" )
5 add_executable( homework )
6 target_sources( homework PRIVATE homework.cpp )
7
8 target_compile_features( homework PRIVATE cxx_std_17 )
9 install( TARGETS homework DESTINATION . )
```

Exercise

- Write a ‘hello world’ program;
- Make a CMake setup to compile and install it;
- Test it all.

Exercise: using the Eigen library

This is a short program using Eigen:

```
1 #include <iostream>
2 #include <Eigen/Dense>
3
4 int main() {
5     // Define a 3x3 matrix
6     Eigen::Matrix3d matrix;
7     matrix << 1, 2, 3,
8               4, 5, 6,
9               7, 8, 9;
10    std::cout << "Original matrix:\n" << matrix << std::endl;
11    return 0;
12 }
```

- Make a CMake setup to compile and install it;
- Test it.

Make your own library

First a library that goes into the executable:

```
1 add_library( auxlib )
2 target_sources( auxlib PRIVATE aux.cxx aux.h )
3 target_link_libraries( program PRIVATE auxlib )
```

Library during build, setup

Full configuration for an executable that uses a library:

```
1  cmake_minimum_required( VERSION 3.13 )
2  project( cmakeprogram VERSION 1.0 )
3
4  add_executable( program )
5  target_sources( program PRIVATE program.cxx )
6
7  add_library( auxlib )
8  target_sources( auxlib PRIVATE aux.cxx aux.h )
9
10 target_link_libraries( program PRIVATE auxlib )
11
12 install( TARGETS program DESTINATION . )
```

Library shared by default; see later.

Shared and static libraries

In the configuration file:

```
add_library( auxlib STATIC )
# or
add_library( auxlib SHARED )
```

(default shared if left out), or by adding a runtime flag

```
cmake -D BUILD_SHARED_LIBS=TRUE
```

Build both by having two lines, one for shared, one for static.
Related: the `-fPIC` compile option is set by `CMAKE_POSITION_INDEPENDENT_CODE`:

```
cmake -D CMAKE_POSITION_INDEPENDENT_CODE=ON
```

Release a library

To have the library released too, use `PUBLIC`.
Add the library target to the `install` command.

Example: released library

```
1  cmake_minimum_required( VERSION 3.13 )
2  project( cmakeprogram VERSION 1.0 )
3
4  add_executable( program )
5  target_sources( program PRIVATE program.cxx )
6
7  add_library( auxlib STATIC )
8  target_sources( auxlib PRIVATE lib/aux.cxx lib/aux.h )
9
10 target_link_libraries( program PUBLIC auxlib )
11 target_include_directories( program PRIVATE lib )
12
13 install( TARGETS program DESTINATION bin )
14 install( TARGETS auxlib DESTINATION lib )
15 install( FILES lib/aux.h DESTINATION include )
```

Note the separate destination directories.

We are getting realistic

The previous setup was messy
Better handle the library through a recursive cmake
and make the usual `lib` `include` `bin` setup

Recursive setup, main directory

Declare that there is a directory to do recursive make:

```
1 cmake_minimum_required( VERSION 3.13 )
2 # needs >3.12 to let the executable target find the .h file
3 project( cmakeprogram VERSION 1.0 )
4
5 add_executable( program )
6 target_sources( program PRIVATE program.cxx )
7 add_subdirectory( lib )
8 target_include_directories(
9         program PUBLIC lib )
10 target_link_libraries( program PUBLIC auxlib )
11 install( TARGETS program DESTINATION bin )
```

(Note that the name of the library comes from the subdirectory)

Recursive setup, subdirectory

Installs into `lib` and `include`

```
1 cmake_minimum_required( VERSION 3.13 )
2 # needs >3.12 to let the executable target find the .h file
3
4 add_library( auxlib STATIC )
5 target_sources( auxlib
6     PRIVATE aux.cxx
7     PUBLIC aux.h )
8 install( TARGETS auxlib DESTINATION lib )
9 install( FILES aux.h DESTINATION include )
```

External libraries

- Use *LD_LIBRARY_PATH*, or
- use *rpath*.

(Apple note: forced to use second option)

```
1 set_target_properties(
2     ${PROGRAM_NAME} PROPERTIES
3     BUILD_RPATH   "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
4     INSTALL_RPATH "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
5     )
```

Fetch content

Include libraries actuall **in** your project:

- Use the `FetchContent` module
- Declare library with `FetchContent_Declare`, build with `FetchContent_MakeAvailable`

```
1  cmake_minimum_required( VERSION 3.20 )
2  project( program VERSION 1.0 )
3
4  include( FetchContent )
5  FetchContent_Declare(
6      fmtlib
7      GIT_REPOSITORY https://github.com/fmtlib/fmt.git
8      )
9  FetchContent_MakeAvailable( fmtlib )
10
11 add_executable( program program.cxx )
12 target_link_libraries( program PRIVATE fmt::fmt )
13
14 install( TARGETS program DESTINATION . )
```

Flexibly fetching

Only fetch if needed:

- Try to find a package with *QUIET*
- Test *MYPACKAGE_FOUND*
- If not, fetch

```
1  cmake_minimum_required( VERSION 3.20 )
2  project( program VERSION 1.0 )
3
4  find_package( fmt QUIET )
5  if( fmt_FOUND )
6      message( STATUS "Found installation of fmtlib" )
7  else()
8      message( STATUS "Installing fmtlib for you" )
9      include( FetchContent )
10     FetchContent_Declare(
11         fmtlib
12         GIT_REPOSITORY https://github.com/fmtlib/fmt.git
13         )
14     FetchContent_MakeAvailable( fmtlib )
15 endif()
16
17 add_executable( program program.cxx )
18 target_link_libraries( program PRIVATE fmt::fmt )
19
20 install( TARGETS program DESTINATION . )
```

Basic customizations

Compiler settings:

```
cmake -D CMAKE_CXX_COMPILER=icpx
```

Alternatively:

```
export CXX=icpx
cmake .......
```

Many settings can be done on the commandline:

```
    -D BUILD_SHARED_LIBS=ON
```

Also check out the `ccmake` utility.

Tracing and logging

- CMake prints some sort of progress messages.
- To see commandlines:

```
cmake -D CMAKE_VERBOSE_MAKEFILE=ON ...
make V=1
```

- CMake leaves behind a log and error file, but these are insufficent:
- $\Rightarrow$ use the above verbose mode and capture all output.

Help! I need to find other software!

Using CMake packages through pkgconfig

What are we talking here?

You have just installed a CMake-based library.
Now you need it in your own code, or in another library.
How easy can we make that?

Problem

You want to install an application/package

. . . which needs 2 or 3 other packages.

```
gcc -o myprogram myprogram.c \
    -I/users/my/package1/include \
    -L/users/my/package1/lib \
    -I/users/my/package2/include/package \
    -L/users/my/package2/lib64
```

or:

```
cmake \
    -D PACKAGE1_INC=/users/my/package1/include \
    -D PACKAGE1_LIB=/users/my/package1/lib \
    -D PACKAGE2_INC=/users/my/package2/include/package \
    -D PACKAGE2_LIB=/users/my/package2/lib64 \
    ../newpackage
```

Can this be made simpler?

Finding packages with ‘pkg config’

- Many packages come with a `package.pc` file
- Add that location to `PKG_CONFIG_PATH`
- The package can now be found by other CMake-based packages.

Package config settings

Let's say you've installed a library with CMake.
Somewhere in the installation is a `.pc` file:

```
find $TACC_SMTHNG_DIR -name \*.pc
${TACC_SMTHNG_DIR}/share/pkgconfig/smthng3.pc
```

That location needs to be on the `PKG_CONFIG_PATH`:

```
export PKG_CONFIG_PATH=${TACC_SMTHNG_DIR}/share/pkgconfig:${
    PKG_CONFIG_PATH}
```

Example: eigen

Can you find the `.pc` file in the Eigen installation?

Scenario 1: finding without cmake

Packages with a `.pc` file can be found through the *pkg-config* command:

```
gcc -o myprogram myprogram.c \
    $( pkg-config --cflags package1 ) \
    $( pkg-config --libs package1 )
```

In a makefile:

```
CFLAGS = -g -O2 $$( pkg-config --cflags package1 )
```

Example: eigen

Make a C++ program (extension *cpp* or *cxx*):

```
1 #include "Eigen/Core"
2 int main(int argc,char **argv) {
3   return 0;
4 }
```

Can you compile this on the commandline, using *pkg-config*? Small problem: ‘eigen’ wants to be called ‘eigen3’.

Scenario 2: finding from CMake

You are installing a CMake-based library
and it needs Eigen, which is also CMake-based

1. you install Eigen with CMake, as above
2. you add the location of `eigen.pc` to `PKG_CONFIG_PATH`
3. you run the installation of the higher library:
   this works because it can now find Eigen.

Lifting the veil

So how does a CMake install find libraries such as Eigen?
Full `CMakeLists.txt` file:

```
1  cmake_minimum_required( VERSION 3.13 )
2  project( eigentest )
3
4  find_package( PkgConfig REQUIRED )
5  pkg_check_modules( EIGEN REQUIRED eigen3 )
6
7  add_executable( eigentest eigentest.cxx )
8  target_include_directories(
9          eigentest PUBLIC
10         ${EIGEN_INCLUDE_DIRS})
```

Note 1: header-only so no library, otherwise `PACKAGE_LIBRARY_DIRS` and `PACKAGE_LIBRARIES` defined.
Note 2: you will learn how to write these configurations in the second part.

Summary for now

- You can use CMake to install libraries;
- You can use these libraries from commandline / makefile;
- You can let other CMake-based libraries find them.

Other discovery mechanisms

Some packages come with `FindWhatever.cmake` or similar files.
Add package root to *CMAKE_MODULE_PATH*
Pity that there is not just one standard.
These define some macros, but you need to read the docs to see which.
Pity that there is not just one standard.
Some examples follow.