TACC

Building projects with CMake

Victor Eijkhout

Fall 2023

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

Using a cmake-based library

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 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:

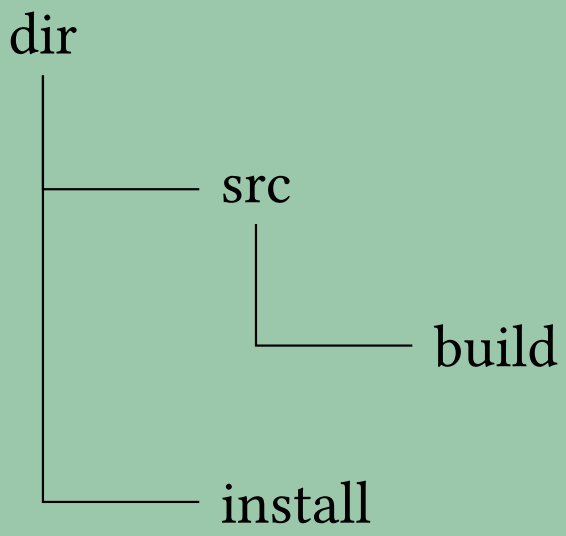
```
cmake ## arguments
cmake --build ## stuff
cmake --install ## stuff
```

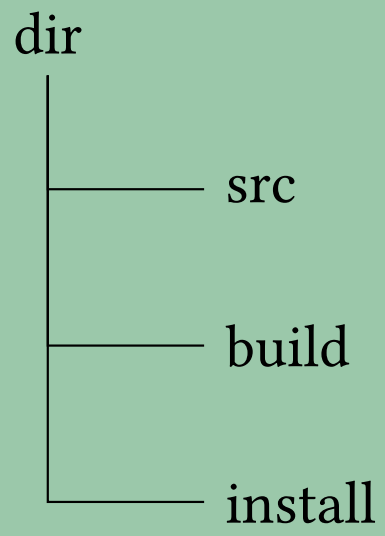
We focus on the first option; the second one is portable to non-Unix environments.

What does this buy you?

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 . . .

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 && cd builddir # goto build location
cmake -D CMAKE_INSTALL_PREFIX=../installdir \
      ../some_package_1.0.0
make # make all tmp data in build loc
make install # move final products to install loc
```

Using packages through pkgconfig

What are we talking here?

TACC

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 a 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/packaage \
    -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/packaage \
    -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.

Somewhere in the installation is a `.pc` file:

```
find $TACC_EIGEN_DIR -name \*.pc
${TACC_EIGEN_DIR}/share/pkgconfig/eigen3.pc
```

That location needs to be on the `PKG_CONFIG_PATH`:

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

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 $( shell pkg-config --cflags package1 )
```

Scenario 2: finding from CMake

TACC

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?

```
1  cmake_minimum_required( VERSION 3.12 )
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 configuration in the second part.

Make your CMake configuration

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.
To do: write the `CMakeLists.txt` file.

The CMakeLists file

```
cmake_minimum_required( VERSION 3.12 )
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.

Target philosophy

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

```
add_executable( myprogram program.cxx )
```

  Use of macros:

```
add_executable( ${PROJECT_NAME} program.cxx )
```

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

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

Example: single source

Build an executable from a single source file:

```
cmake_minimum_required( VERSION 3.12 )
project( singleprogram VERSION 1.0 )

add_executable( program program.cxx )
install( TARGETS program DESTINATION . )
```


Use of a library

First a library that goes into the executable:

```
add_library( auxlib aux.cxx aux.h )
target_link_libraries( program PRIVATE auxlib )
```

Example: library during build

Full configuration for an executable that uses a library:

```
1  cmake_minimum_required( VERSION 3.12 )
2  project( cmakeprogram VERSION 1.0 )
3
4  add_executable( program program.cxx )
5  add_library( auxlib
6               aux.cxx aux.h )
7  target_link_libraries( program PRIVATE auxlib )
8  install( TARGETS program DESTINATION . )
```

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 3.12 )
2  project( cmakeprogram VERSION 1.0 )
3
4  add_executable( program program.cxx )
5  add_library( auxlib
6               aux.cxx aux.h )
7  target_link_libraries( program PUBLIC auxlib )
8  install( TARGETS program auxlib DESTINATION . )
```


More about libraries

Static vs shared libraries. In the configuration file:

```
add_library( auxlib STATIC aux.cxx aux.h )
# or
add_library( auxlib SHARED aux.cxx aux.h )
```

or by adding a runtime flag

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

Related: the `-fPIC` compile option is set by `CMAKE_POSITION_INDEPENDENT_CODE`.

External libraries

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

(Apple note: forced to use second option)

```
set_target_properties(
        ${PROGRAM_NAME} PROPERTIES
        BUILD_RPATH   "${CATCH2_LIBRARY_DIRS};${
    FMTLIB_LIBRARY_DIRS}"
        INSTALL_RPATH "${CATCH2_LIBRARY_DIRS};${
    FMTLIB_LIBRARY_DIRS}"
    )
```

More stuff

Other discovery mechanisms

Some packages come with `FindWhatever.cmake` or similar files.
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.

MPI from C

```
cmake_minimum_required( VERSION 3.12 )
project( cxxprogram VERSION 1.0 )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
find_package( MPI )
target_include_directories(
       ${PROJECT_NAME} PUBLIC
       ${MPI_C_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_libraries(
       ${PROJECT_NAME} PUBLIC
       ${MPI_C_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

MPI from Fortran

```
1  cmake_minimum_required( VERSION 3.12 )
2  project( ${PROJECT_NAME} VERSION 1.0 )
3
4  enable_language(Fortran)
5
6  find_package( MPI )
7
8  if( MPI_Fortran_HAVE_F08_MODULE )
9  else()
10   message( FATAL_ERROR "No f08 module for this MPI" )
11 endif()
12
13 add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
14 target_include_directories(
15         ${PROJECT_NAME} PUBLIC
16         ${MPI_Fortran_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR}
       )
17 target_link_directories(
18         ${PROJECT_NAME} PUBLIC
19         ${MPI_LIBRARY_DIRS} )
20 target_link_libraries(
21         ${PROJECT_NAME} PUBLIC
```


OpenMP

```
cmake_minimum_required( VERSION 3.12 )
project( ompprogram VERSION 1.0 )

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()

add_executable( program program.cxx )
target_link_libraries( program
    PUBLIC OpenMP::OpenMP_CXX)

install( TARGETS program DESTINATION . )
```

The build/make cycle

CMake creates makefiles;
makefiles ensure minimal required compilation

```
cmake                ## make the makefiles
make                 ## compile your project
emacs onefile.c ## edit
make                 ## minimal recompile
```

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