TACC

Building projects with CMake

Victor Eijkhout

Fall 2022

Justification

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

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 cCMake.
- . . . and then figure out how to use it in your code.

Building software the old way

Using ‘GNU Autotools’:

```
./configure
make
make install
```

User vs system packages

The `make install` often tries to copy to a system directory. If you're not the admin, do:

```
./configure --prefix=/home/yourname/mypackages
```

with a location of your choice.

Building with CMake

- Either replace only the configure stage

```
cmake -D CMAKE_INSTALL_PREFIX=/home/yourname/packages
    ## maybe more arguments
make
make install
```

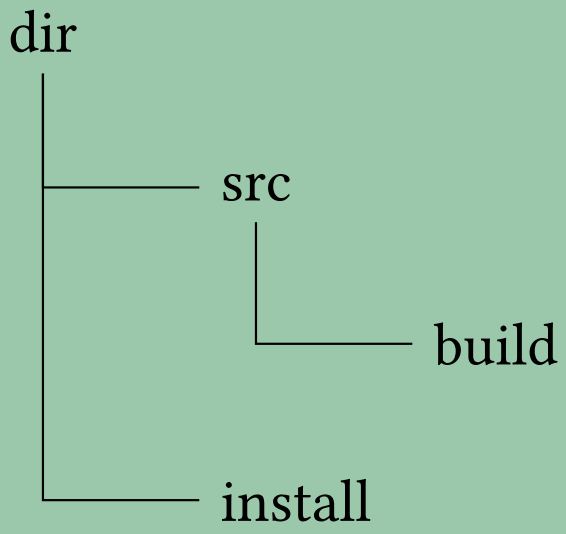
or

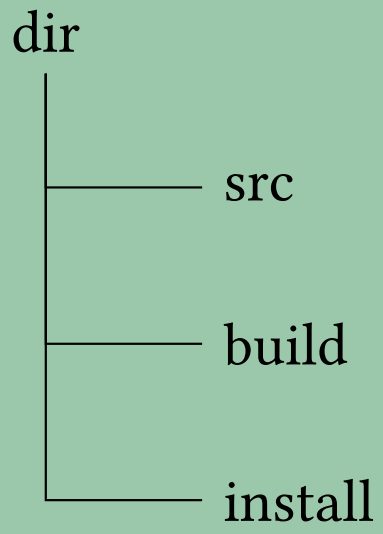
- do everything with CMake:

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

(The second one is portable to non-Unix environments.)

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, 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 stuff to install loc
```

Using packages through pkgconfig

Problem

You want to install a package/application
. . . which needs 2 or 3 other packages.

```
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 make simpler?

Finding packages with ‘pkg config’

- Many packages come with a `package.pc` file
- Add that location to `PKG_CONFIG_PATH`
- That defines variables in your own cmake file

Example: PETSc
add `$PETSC_DIR/$PETSC_ARCH/lib/pkgconfig` to config path, then

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( PETSC REQUIRED petsc )
target_include_directories(
       program PUBLIC
       ${PETSC_INCLUDE_DIRS} )
```

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})
```

Where do these settings come from?

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

Pkgconfig outside CMake

`.pc` files are also useful outside CMake:

```
$ pkg-config --cflags tbb
-I/opt/intel//oneapi/tbb/latest/lib/pkgconfig/../..//include
$ pkg-config --libs tbb
-L/opt/intel//oneapi/tbb/latest/lib/pkgconfig/../..//lib/intel64
    /gcc4.8 -ltbb
```

(useful for packages where there is no module)

Make your CMake configuration

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
- specify what is needed for it

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

Use of macros:

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


Example: single source

```
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

```
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

```
1  cmake_minimum_required( VERSION 3.12 )
2  project( ${PROJECT_NAME} VERSION 1.0 )
3
4  find_package( MPI )
5
6  add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.c )
7  target_include_directories(
8         ${PROJECT_NAME} PUBLIC
9         ${MPI_C_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
10 target_link_libraries(
11        ${PROJECT_NAME} PUBLIC
12        ${MPI_C_LIBRARIES} )
13
14 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.