Chapter 4

The Cmake build system

4.1 CMake as build system

CMake is a general build system that uses other systems such as *Make* as a back-end. The general workflow is:

1. The configuration stage. Here the *CMakeLists.txt* file is parsed, and a `build` directory populated. This typically looks like:
   ```
   mkdir build
   cd build
   cmake <source location>
   ```
   Some people create the build directory in the source tree, in which case the *CMake* command is
   ```
   cmake ..
   ```
   Others put the build directory next to the source, in which case:
   ```
   cmake ../src_directory
   ```
2. The build stage. Here the installation-specific compilation in the build directory is performed. With *Make* as the 'generator' this would be
   ```
   cd build
   make
   ```
   but more generally
   ```
   cmake --build <build directory>
   ```
   Alternatively, you could use generators such as *ninja*, *Visual Studio*, or *XCode*:
   ```
   cmake -G ninja
     ## the usual arguments
   ```
3. The install stage. This can move binary files to a permanent location, such as putting library files in `/usr/lib`:
   ```
   make install
   ```
   or

General directives	
`cmake_minimum_required`	specify minimum cmake version
`project`	name and version number of this project
`install`	specify directory where to install targets
Project building directives	
`add_executable`	specify executable name and source files for it
`add_library`	specify library name and files to go into it
`add_subdirectory`	specify subdirectory where cmake also needs to run
`target_link_libraries`	specify executable and libraries to link into it
`target_include_directories`	specify include directories, privately or publicly
`find_package`	other package to use in this build
Utility stuff	
`target_compile_options`	literal options to include
`target_compile_features`	things that will be translated by cmake into options
`target_compile_definitions`	macro definitions to be set private or publicly
`file`	define macro as file list
`message`	Diagnostic to print, subject to level specification
Control	
`if() else() endif()`	conditional

Table 4.1: Cmake commands.

```
cmake --install <build directory>
```

However, the install location already has to be set in the configuration stage. We will see later in detail how this is done.

Summarizing, the out-of-source workflow as advocated in this tutorial is

```
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 install
```

The resulting directory structure is illustrated in figure 4.1.

Figure 4.1: In-source (left) and out-of-source (right) build schemes.

4.1.1 Target philosophy

Modern *CMake* works through declaring targets and their requirements. For requirements during building:

```
target_some_requirement( the_target PRIVATE the require ments )
```

Usage requirements:

```
target_some_requirement( the_target PUBLIC the require ments )
```

4.1.2 Languages

CMake is largely aimed at C++, but it easily supports C as well. For *Fortran* support, first do

```
enable_language(Fortran)
```

Note that capitalization: this also holds for all variables such as `CMAKE_Fortran_COMPILER`.

4.1.3 Script structure

Commands learned in this section	
`cmake_minimum_required`	declare minimum required version for this script
`project`	declare a name for this project

CMake is driven by the *CMakeLists.txt* file. This needs to be in the root directory of your project. (You can additionally have files by that name in subdirectories.)

Since *CMake* has changed quite a bit over the years, and is still evolving, it is a good idea to start each script with a declaration of the (minimum) required version:

```
cmake_minimum_required( VERSION 3.12 )
```

You can query the version of your *CMake* executable:

```
$ cmake --version
cmake version 3.19.2
```

You also need to declare a project name and version, which need not correspond to any file names:

```
project( myproject VERSION 1.0 )
```

4.2 Examples cases

4.2.1 Executable from sources

(The files for this examples are in `tutorials/cmake/single`.)

Commands learned in this section	
`add_executable`	declare an executable and its sources
`install`	indicate location where to install this project
`PROJECT_NAME`	macro that expands to the project name

If you have a project that is supposed to deliver an executable, you declare in your `CMakeLists.txt`:

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

Often, the name of the executable is the name of the project, so you'd specify:

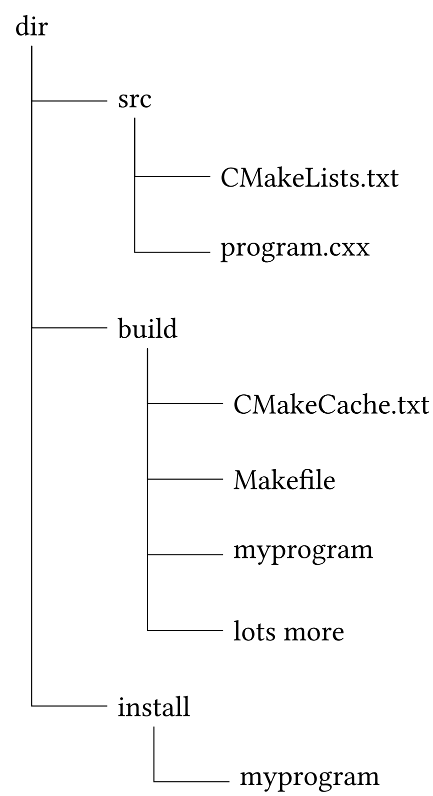
```
add_executable( ${PROJECT_NAME} program.cxx )
```

In order to move the executable to the install location, you need a clause

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

Without the `DESTINATION` clause, a default `bin` directory will be created; specifying `DESTINATION foo` will put the program in a `foo` sub-directory of the installation directory.

In the figure on the right we have also indicated the `build` directory, which from now on we will not show again. It contains automatically generated files that are hard to decyper, or debug. Yes, there is a `Makefile`, but even for simple projects this is too complicated to debug by hand if your *CMake* installation misbehaves.



Here is the full `CMakeLists.txt`:

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

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

4.2.2 Making libraries

(The files for this examples are in `tutorials/cmake/multiple`.)

Commands learned in this section	
`add_library`	declare a library and its sources
`target_link_libraries`	indicate that the library belong with an executable

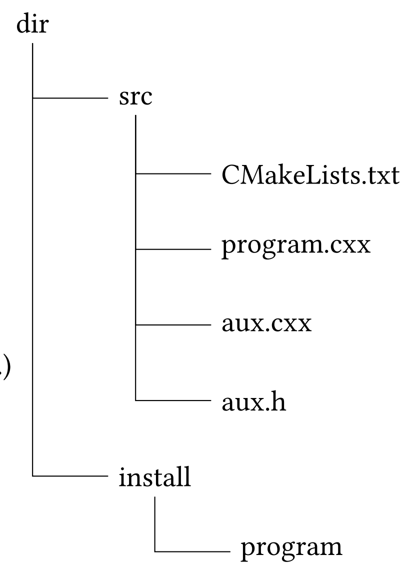
If there is only one source file, the previous section is all you need. However, often you will build libraries. You declare those with an `add_library` clause:

```
add_library( auxlib aux.cxx aux.h )
```

Next, you need to link that library into the program:

```
target_link_libraries( program PRIVATE auxlib )
```

The `PRIVATE` clause means that the library is only for purposes of building the executable. (Use `PUBLIC` to have the library be included in the installation; we will explore that in section 4.2.2.2.)



The full `CMakeLists.txt`:

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

add_executable( program program.cxx )
add_library( auxlib STATIC
             aux1.cxx aux2.cxx aux.h )
target_link_libraries( program PRIVATE auxlib )
install( TARGETS program DESTINATION . )
```

Note that private shared libraries make no sense, as they will give runtime unresolved references.

4.2.2.1 Testing the generated makefiles

In the Make tutorial 3 you learned how Make will only recompile the strictly necessary files when a limited edit has been made. The makefiles generated by *CMake* behave similarly. With the structure above, we first touch the `aux.cxx` file, which necessitates rebuilding the library:

```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
```

```
[100%] Built target program
```

On the other hand, if we edit a header file, the main program needs to be recompiled too:

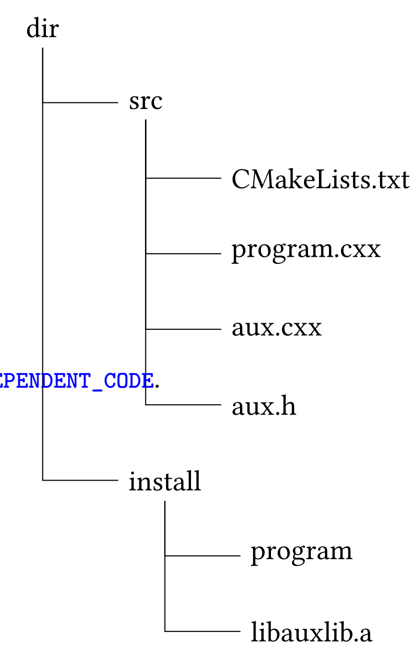
```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
[100%] Built target program
```

4.2.2.2 *Making a library for release*

(The files for this example are in `tutorials/cmake/withlib`.)

Commands learned in this section	
`SHARED`	indicated to make shared libraries

In order to create a library we use `add_library`, and we link it into the target program with `target_link_libraries`.



By default the library is build as a static `.a` file, but adding

```
add_library( auxlib SHARED aux.cxx aux.h )
```

or adding a runtime flag

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

changes that to a shared `.so` type.

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

The full *CMake* file:

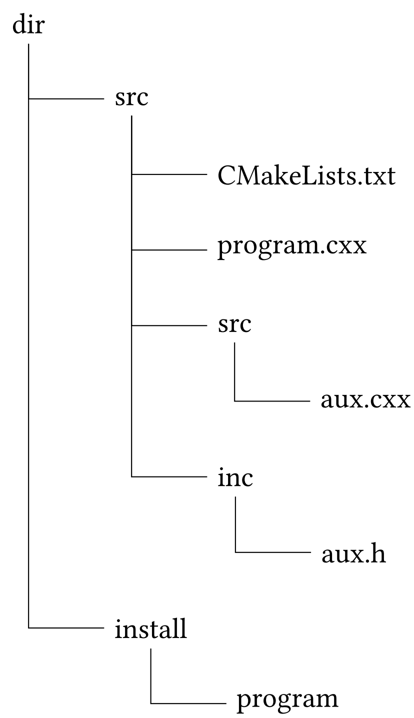
```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program program.cxx )
add_library( auxlib
             aux.cxx aux.h )
target_link_libraries( program PUBLIC auxlib )
install( TARGETS program auxlib DESTINATION . )
```

4.2.3 Using subdirectories during the build

(The files for this examples are in `tutorials/cmake/includedir`.)

Commands learned in this section	
`target_include_directories`	indicate include directories needed
`target_sources`	specify more sources for a target
`CMAKE_CURRENT_SOURCE_DIR`	variable that expands to the current directory
`file`	define single-name synonym for multiple files
`GLOB`	define single-name synonym for multiple files

Suppose you have a directory with header files, as in the diagram on the right. The main program would have




```
#include <iostream>
using namespace std;

#include "aux.h"

int main() {
  aux1();
  aux2();
  return 0;
}
```

and which is compiled as:

```
cc -c program.cxx -I./inc
```

To make sure the header file gets found during the build, you specify that include path with `target_include_directories`:

```
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )
```

It is best to make such paths relative to `CMAKE_CURRENT_SOURCE_DIR`
, or the source root `CMAKE_SOURCE_DIR`, or equivalently `PROJECT_SOURCE_DIR`

Usually, when you start making such directory structure, you will also have sources in subdirectories. If you only need to compile them into the main executable, you could list them into a variable

```
set( SOURCES program.cxx src/aux.cxx )
```

and use that variable. However, this is deprecated practice; it is recommended to use `target_sources`:

```
target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
```

Use of a wildcard is not trivial:

```
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
```

Complete *CMake* file:

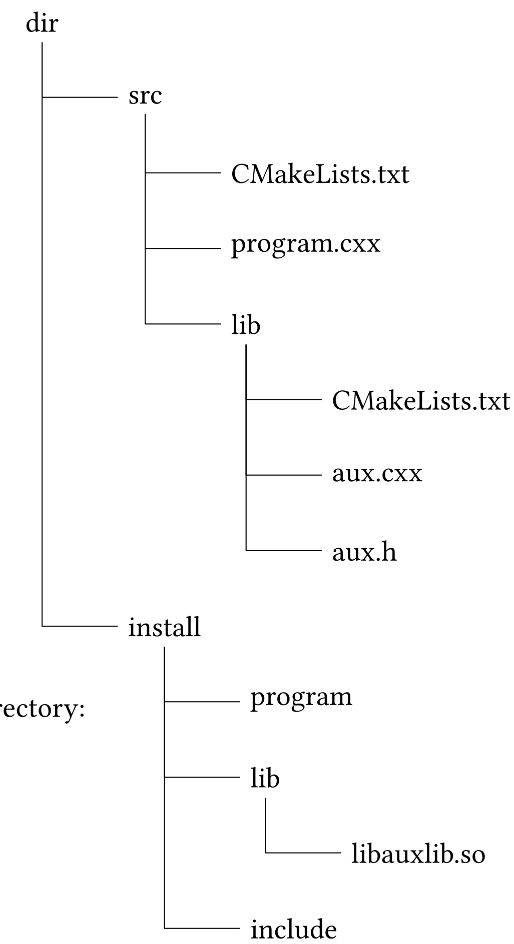
```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

add_executable( program program.cxx )
#target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )

install( TARGETS program DESTINATION . )
```

4.2.4 Libraries for release; rpath

(The files for this examples are in `tutorials/cmake/publiclib`.)

Commands learned in this section	
`add_subdirectory`	declare a subdirectory where cmake needs to be run
`CMAKE_CURRENT_SOURCE_DIR`	directory where this command is evaluated
`CMAKE_CURRENT_BINARY_DIR`	
`LIBRARY_OUTPUT_PATH`	
`FILES_MATCHING PATTERN`	wildcard indicator

If your sources are spread over multiple directories, there needs to be a `CMakeLists.txt` file in each, and you need to declare the existence of those directories. Let's start with the obvious choice of putting library files in a `lib` directory with `add_subdirectory`:




```
add_subdirectory( lib )
```

For instance, a library directory would have a `CMakeLists.txt` file:

```
cmake_minimum_required( VERSION 3.14 )
project( auxlib )

add_library( auxlib SHARED
             aux.cxx aux.h )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}" )
install( TARGETS auxlib DESTINATION lib )
install( FILES aux.h DESTINATION include )
```

to build the library file from the sources indicated, and to install it in a `lib` subdirectory.

We also add a clause to install the header files in an include directory:

```
install( FILES aux.h DESTINATION include )
```

For installing multiple files, use

```
install(DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
    DESTINATION ${LIBRARY_OUTPUT_PATH}
    FILES_MATCHING PATTERN "*.h")
```

One problem is to tell the executable where to find the library. For this we use the *rpath* mechanism. By default, *CMake* sets it so that the executable in the build location can find the library. If you use a non-trivial install prefix, the following lines work:

```
set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )
```

Note that these have to be specified before the target.

The whole file:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )

add_executable( program program.cxx )
add_subdirectory( lib )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRECT_SOURCE_DIR}" )
target_link_libraries(
        program PUBLIC auxlib )

install( TARGETS program DESTINATION . )
```

4.2.5 Programs that use other libraries

So far we have discussed executables and libraries that can be used by themselves. What to if your build result needs external libraries? We will discuss how to find those libraries in section 4.3; here we point out their use.

The issue is that these libraries need to be findable when someone uses your binary. There are two strategies:

- make sure they have been added to the *LD_LIBRARY_PATH*;
- have the *linker* add their location through the *rpath* mechanism to the binary itself. This second option is in fact the only one on recent versions of *Apple Mac OS* because of 'System Integrity Protection'.

As an example, assume your binary needs the *Catch2* and *fmtlib* libraries. You would then, in addition to the `target_link_directories` specification, have:

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

4.2.6 Header-only libraries

Use the `INTERFACE` keyword.

4.3 Finding and using external packages

If your program depends on other libraries, there is a variety of ways to let *CMake* find them.

4.3.1 CMake commandline options

(The files for this example are in `tutorials/cmake/usepubliclib`.)

You can indicate the location of your external library explicitly on the commandline.

```
cmake -D OTHERLIB_INC_DIR=/some/where/include
      -D OTHERLIB_LIB_DIR=/somewhere/lib
```

Example *CMake* file:

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

# with environment variables
# set( AUX_INCLUDE_DIR $ENV{TACC_AUX_INC} )
# set( AUX_LIBRARY_DIR $ENV{TACC_AUX_LIB} )

# with cmake -D options
```

```
option( AUX_INCLUDE_DIR "include dir for auxlib" )
option( AUX_LIBRARY_DIR   "lib dir for auxlib" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${AUX_INCLUDE_DIR} )
target_link_libraries( program PUBLIC auxlib )
target_link_directories(
        program PUBLIC
        ${AUX_LIBRARY_DIR} )
install( TARGETS program DESTINATION . )
```

4.3.2 Package finding through 'find library' and 'find package'

Commands learned in this section	
find_library	find a library with a FOOConfig.cmake file
CMAKE_PREFIX_PATH	location for FOOConfig.cmake files
find_package	find a library with a FindFOO module
CMAKE_MODULE_PATH	location for FindFOO modules

The find_package command looks for files with a name FindXXX.cmake, which are searched on the CMAKE_MODULE_PATH. Unfortunately, the working of find_package depend somewhat on the specific package. For instance, most packages set a variable *FooFound* that you can test

```
find_package( Foo )
if ( FooFound )
    # do something
else()
    # throw an error
endif()
```

Some libraries come with a FOOConfig.cmake file, which is searched on the CMAKE_PREFIX_PATH through find_library. If it is found, you can test the variable it is supposed to set:

```
find_library( FOOLIB foo )
if (FOOLIB)
  target_link_libraries( myapp PRIVATE ${FOOLIB} )
else()
  # throw an error
endif()
```

4.3.2.1 Example: MPI

(The files for this example are in tutorials/cmake/mpiprog.)

While many MPI implementations have a .pc file, it's better to use the FindMPI module. This package defines a number of variables that can be used to query the MPI found; for details see https://cmake.org/cmake/help/latest/module/FindMPI.html

C version:

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

find_package( MPI )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.c )
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 . )
```

Fortran version:

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

enable_language(Fortran)

find_package( MPI )

if( MPI_Fortran_HAVE_F08_MODULE )
else()
  message( FATAL_ERROR "No f08 module for this MPI" )
endif()

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_LIBRARIES} )

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

4.3.2.2 Example: OpenMP

```
find_package(OpenMP)
if(OpenMP_C_FOUND) # or CXX
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()
# for C:
add_executable( ${program} ${program}.c )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_C )
# for C++:
add_executable( ${program} ${program}.cxx )
```

```
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_CXX)
# for Fortran
enable_language(Fortran)
# test: if( OpenMP_Fortran_FOUND )
add_executable( ${program} ${program}.F90 )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_Fortran )
```

4.3.2.3 Example: MKL

(The files for this example are in `tutorials/cmake/mklcmake`.)

Intel compiler installations come with *CMake* support: there is a file `MKLConfig.cmake`.

Example program using *Cblas* from MKL:

```
#include <iostream>
#include <vector>
using namespace std;

#include "mkl_cblas.h"

int main() {
  vector<double> values{1,2,3,2,1};
  auto maxloc = cblas_idamax ( values.size(),values.data(),1);
  cout << "Max abs at: " << maxloc << " (s/b 2)" << '\n';

  return 0;
}
```

The following configuration file lists the various options and such:

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

## https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/
    top/getting-started/cmake-config-for-onemkl.html

find_package( MKL CONFIG REQUIRED )

add_executable( program program.cxx )
target_compile_options(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_COMPILE_OPTIONS> )
target_include_directories(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_INCLUDE_DIRECTORIES> )
target_link_libraries(
        program PUBLIC
        $<LINK_ONLY:MKL::MKL>)

install( TARGETS program DESTINATION . )
```

4.3.3 Use of other packages through 'pkg config'

These days, many package support the *pkgconfig* mechanism.

1. Suppose you have a library `mylib`, installed in `/opt/local/mylib`.
2. If `mylib` supports `pkgconfig`, there is most likely a path `/opt/local/mylib/lib/pkgconfig`, containing a file `mylib.pc`.
3. Add the path that contains the *.pc* file to the *PKG_CONFIG_PATH* environment variable.

Cmake is now able to find `mylib`:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( MYLIBRARY REQUIRED mylib )
```

This defines variables

```
MYLIBRARY_INCLUDE_DIRS
MYLIBRARY_LIBRARY_DIRS
MYLIBRARIES
```

which you can then use in the `target_include_directories` and `target_link_directories` `target_link_libraries` commands.

4.3.3.1 Example: PETSc

(The files for this example are in `tutorials/cmake/petscprog`.)

This *CMake* setup searches for `petsc.pc`, which is located in `$PETSC_DIR/$PETSC_ARCH/lib/pkgconfig`:

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

find_package( PkgConfig REQUIRED )
pkg_check_modules( PETSC REQUIRED petsc )
message( STATUS "PETSc includes: ${PETSC_INCLUDE_DIRS}" )
message( STATUS "PETSc libraries: ${PETSC_LIBRARY_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${PETSC_INCLUDE_DIRS} )
target_link_directories(
        program PUBLIC
        ${PETSC_LIBRARY_DIRS} )
target_link_libraries(
        program PUBLIC petsc )

install( TARGETS program DESTINATION . )
```

4.3.3.2 Example: Eigen

(The files for this example are in `tutorials/cmake/eigen`.)

The *eigen* package uses *pkgconfig*.

```
cmake_minimum_required( VERSION 3.12 )
project( eigentest )

find_package( PkgConfig REQUIRED )
pkg_check_modules( EIGEN REQUIRED eigen3 )

add_executable( eigentest eigentest.cxx )
target_include_directories(
        eigentest PUBLIC
        ${EIGEN_INCLUDE_DIRS})
```

4.3.3.3 Example: cxxopts

(The files for this example are in `tutorials/cmake/cxxopts`.)

The *cxxopts* package uses *pkgconfig*.

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

find_package( PkgConfig REQUIRED )
pkg_check_modules( OPTS REQUIRED cxxopts )
message( STATUS "cxxopts includes: ${OPTS_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${OPTS_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

4.3.3.4 Example: fmtlib

(The files for this example are in `tutorials/cmake/fmtlib`.)

In the following example, we use the *fmtlib*. The main *CMake* file:

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

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes: ${FMTLIB_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

4.3.3.5 *Example: fmtlib used in library*

(The files for this example are in `tutorials/cmake/fmtliblib`.)

We continue using the *fmtlib* library, but now the generated library also has references to this library, so we use `target_link_directories` and `target_link_library`.

Main file:

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

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes : ${FMTLIB_INCLUDE_DIRS}" )
message( STATUS "fmtlib lib dirs : ${FMTLIB_LIBRARY_DIRS}" )
message( STATUS "fmtlib libraries: ${FMTLIB_LIBRARIES}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

add_subdirectory( prolib )
target_link_libraries( program PUBLIC prolib )

install( TARGETS program DESTINATION . )
```

Library file:

```
project( prolib )

add_library( prolib SHARED aux.cxx aux.h )
target_include_directories(
        prolib PUBLIC
        ${FMTLIB_INCLUDE_DIRS})
target_link_directories(
        prolib PUBLIC
        ${FMTLIB_LIBRARY_DIRS})
target_link_libraries(
        prolib PUBLIC fmt )
```

4.3.4 Writing your own pkg config

We extend the configuration of section 4.2.4 to generate a `.pc` file.

First of all we need a template for the `.pc` file:

```
prefix="@CMAKE_INSTALL_PREFIX@"
exec_prefix="${prefix}"
libdir="${prefix}/lib"
includedir="${prefix}/include"

Name: @PROJECT_NAME@
```

```
Description: @CMAKE_PROJECT_DESCRIPTION@
Version: @PROJECT_VERSION@
Cflags: -I${includedir}
Libs: -L${libdir} -l@libtarget@
```

and we transform this by:

```
set( libtarget auxlib )
configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/${PROJECT_NAME}.pc.in
    ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    @ONLY
)
install(
    FILES ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    DESTINATION share/pkgconfig
)
```

4.4 Customizing the compilation process

Commands learned in this section	
add_compile_options	global compiler options
target_compile_features	compiler-independent specification of compile flags
target_compile_definitions	pre-processor flags

4.4.1 Customizing the compiler

It's probably a good idea to tell *CMake* explicitly what compiler you are using, otherwise it may find some default gcc version that came with your system. Use the variables CMAKE_CXX_COMPILER, CMAKE_C_COMPILER, CMAKE_Fortran_COMPILER, CMAKE_LINKER.

Alternatively, set environment variables *CC*, *CXX*, *FC* by the explicit paths of the compilers. For examples, for Intel compilers:

```
export CC=`which icc`
export CXX=`which icpc`
export FC=`which ifort`
```

4.4.2 Global and target flags

Most of the time, compile options should be associated with a target. For instance, some file could need a higher or lower optimization level, or a specific C++ standard. In that case, use target_compile_features.

Certain options may need to be global, in which case you use add_compile_options. Example:

```
## from https://youtu.be/eC9-iRN2b04?t=1548
if (MVSC)
    add_compile_options(/W3 /WX)
else()
    add_compile_options(-W -Wall -Werror)
endif()
```

4.4.2.1 *Universal flags*

Certain flags have a universal meaning, but compiler-dependent realization. For instance, to specify the C++ standard:

```
target_compile_features( mydemo PRIVATE cxx_std_17 )
```

Alteratively, you can set this one the commandline:

```
cmake -D CMAKE_CXX_STANDARD=20
```

The variable `CMAKE_CXX_COMPILE_FEATURES` contains the list of all features you can set.

Optimization flags can be set by specifying the `CMAKE_BUILD_TYPE`:

- *Debug* corresponds to the `-g` flag;
- *Release* corresponds to `-O3 -DNDEBUG`;
- *MinSizeRel* corresponds to `-Os -DNDEBUG`
- *RelWithDebInfo* corresponds to `-O2 -g -DNDEBUG`.

This variable will often be set from the commandline:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

Unfortunately, this seems to be the only way to influence optimization flags, other than explicitly setting compiler flags; see next point.

4.4.2.2 *Custom compiler flags*

Set the variable `CMAKE_CXX_FLAGS` or `CMAKE_C_FLAGS`; also `CMAKE_LINKER_FLAGS` (but see section 4.2.4 for the popular *rpath* options.)

4.4.3 Macro definitions

CMake can provide macro definitions:

```
target_compile_definitions
  ( programname PUBLIC
    HAVE_HELLO_LIB=1 )
```

and your source could test these:

```
#ifdef HAVE_HELLO_LIB
#include "hello.h"
#endif
```

4.5 CMake scripting

Commands learned in this section	
option	query a commandline option
message	trace message during cmake-ing
set	set the value of a variable
CMAKE_SYSTEM_NAME	variable containing the operating system name
STREQUALS	string comparison operator

The *CMakeLists.txt* file is a script, though it doesn't much look like it.

- Instructions consist of a command, followed by a parenthesized list of arguments.
- (All arguments are strings: there are no numbers.)
- Each command needs to start on a new line, but otherwise whitespace and line breaks are ignored.

Comments start with a hash character.

4.5.1 System dependencies

```
if (CMAKE_SYSTEM_NAME STREQUALS "Window")
  target_compile_options( myapp PRIVATE /W4 )
elseif (CMAKE_SYSTEM_NAME STREQUALS "Darwin" -Wall -Wextra -Wpedantic)
  target_compile_options( myapp PRIVATE /W4 )
endif()
```

4.5.2 Messages, errors, and tracing

The `message` command can be used to write output to the console. This command has two arguments:

```
message( STATUS "We are rolling!")
```

Instead of `STATUS` you can specify other logging levels (this parameter is actually called 'mode' in the documentation); running for instance

```
cmake --log-level=NOTICE
```

will display only messages of 'notice' status or higher.

The possibilities here are: `FATAL_ERROR`, `SEND_ERROR`, `WARNING`, `AUTHOR_WARNING`, `DEPRECATION`, `NOTICE`, `STATUS`, `VERBOSE`, `DEBUG`, `TRACE`.

The `NOTICE`, `VERBOSE`, `DEBUG`, `TRACE` options were added in CMake-3.15.

For a complete trace of everything *CMake* does, use the commandline option `--trace`.

You can get a verbose make file by using the option

```
-D CMAKE_VERBOSE_MAKEFILE=ON
```

on the *CMake* invocation. You still need `make V=1`.

4.5.3 Variables

Variables are set with `set`, or can be given on the commandline:

```
cmake -D MYVAR=myvalue
```

where the space after `-D` is optional.

Using the variable by itself gives the value, except in strings, where a shell-like notation is needed:

```
set(SOME_ERROR "An error has occurred")
message(STATUS "${SOME_ERROR}")
set(MY_VARIABLE "This is a variable")
message(STATUS "Variable MY_VARIABLE has value ${MY_VARIABLE}")
```

Variables can also be queried by the *CMake* script using the `option` command:

```
option( SOME_FLAG "A flag that has some function" defaultvalue )
```

Some variables are set by other commands. For instance the `project` command sets `PROJECT_NAME` and `PROJECT_VERSION`.

4.5.3.1 Environment variables

Environment variables can be queried with the `ENV` command:

```
set( MYDIR $ENV{MYDIR} )
```

4.5.3.2 Numerical variables

```
math( EXPR lhs_var "math expr" )
```

4.5.4 Control structures

4.5.4.1 Conditionals

```
if ( MYVAR MATCHES "value$" )
    message( NOTICE "Variable ended in 'value'" )
elseif( stuff )
    message( stuff )
else()
    message( NOTICE "Variable was otherwise" )
endif()
```

4.5.4.2 Looping

```
while( myvalue LESS 50 )
    message( stuff )
endwhile()
```

```
foreach ( var IN ITEMS item1 item2 item3 )
  ## something wityh ${var}
endforeach()
foreach ( var IN LISTS list1 list2 list3 )
  ## something wityh ${var}
endforeach()
```

Integer range, with inclusive bounds, upper bound zero by default:

```
foreach ( idx RANGE 10 )
foreach ( idx RANGE 5 10 )
foreach ( idx RANGE 5 10 2 )
endforeach()
```

4.5.4.3 Things not to do

Do not use macros that affect all targets: `include_directories`, `add_definitions`, `link_libraries`.

Do not use `target_include_directories` outside your project: that should be found through some of the above mechanisms.