## How to create a shared library with cmake?

Asked 11 years, 5 months ago Modified 1 year, 1 month ago Viewed 367k times

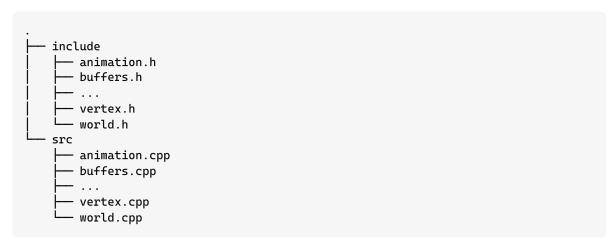


I have written a library that I used to compile using a self-written Makefile, but now I want to switch to cmake. The tree looks like this (I removed all the irrelevant files):

234







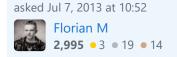
So what I am trying to do is just to compile the source into a shared library and then install it with the header files.

Most examples that I have found compile executables with some shared libraries but never just a plain shared library. It would also be helpful if someone could just tell me a very simple library that uses cmake, so I can use this as an example.



Share Improve this question Follow





related stackoverflow.com/questions/2152077/... - Ciro Santilli OurBigBook.com Apr 15, 2016 at 20:35

Same question, but is there a way to maintain my sources mixed (.h ans .cpp in the same sources directory) but to let Cmake produce an include directory, product of its work? – Sandburg Oct 22, 2019 at 7:27

## 4 Answers

Sorted by: Highest score (default)



cmake\_minimum\_required(VERSION 3.9)

406



You should declare a project. cmake says it is mandatory and it will define convenient variables PROJECT\_NAME, PROJECT\_VERSION and PROJECT\_DESCRIPTION (this latter variable necessitate cmake 3.9):



```
project(mylib VERSION 1.0.1 DESCRIPTION "mylib description")
```



Declare a new library target. Please avoid the use of file(GLOB ...). This feature does not provide attended mastery of the compilation process. If you are lazy, copy-paste output of ls -1 sources/\*.cpp :

```
add_library(mylib SHARED
    sources/animation.cpp
    sources/buffers.cpp
    [...]
)
```

Set **VERSION** property (optional but it is a good practice):

```
set_target_properties(mylib PROPERTIES VERSION ${PROJECT_VERSION})
```

You can also set SOVERSION to the major number of VERSION. So libmylib.so.1 will be a symlink to libmylib.so.1.0.0.

```
set_target_properties(mylib PROPERTIES SOVERSION ${PROJECT_VERSION_MAJOR})
```

Declare public API of your library. This API will be installed for the third-party application. It is a good practice to isolate it in your project tree (like placing it include/ directory). Notice that, private headers should not be installed and I strongly suggest to place them with the source files.

```
set_target_properties(mylib PROPERTIES PUBLIC_HEADER include/mylib.h)
```

If you work with subdirectories, it is not very convenient to include relative paths like "../include/mylib.h". So, pass a top directory in included directories:

```
target_include_directories(mylib PRIVATE .)
```

```
target_include_directories(mylib PRIVATE include)
target_include_directories(mylib PRIVATE src)
```

Create an install rule for your library. I suggest to use variables CMAKE\_INSTALL\_\*DIR defined in GNUInstallDirs:

```
include(GNUInstallDirs)
```

And declare files to install:

```
install(TARGETS mylib
   LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
   PUBLIC_HEADER DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
```

You may also export a pkg-config file. This file allows a third-party application to easily import your library:

- with Makefile, see <a href="pkg-config">pkg-config</a>
- with Autotools, see <a href="PKG\_CHECK\_MODULES">PKG\_CHECK\_MODULES</a>
- with cmake, see <a href="mailto:pkg\_check\_modules">pkg\_check\_modules</a>

Create a template file named mylib.pc.in (see <u>pc(5) manpage</u> for more information):

```
prefix=@CMAKE_INSTALL_PREFIX@
exec_prefix=@CMAKE_INSTALL_PREFIX@
libdir=${exec_prefix}/@CMAKE_INSTALL_LIBDIR@
includedir=${prefix}/@CMAKE_INSTALL_INCLUDEDIR@

Name: @PROJECT_NAME@
Description: @PROJECT_DESCRIPTION@
Version: @PROJECT_VERSION@

Requires:
Libs: -L${libdir} -lmylib
Cflags: -I${includedir}
```

In your CMakeLists.txt, add a rule to expand @ macros (@ONLY ask to cmake to not expand variables of the form \${VAR}):

```
configure_file(mylib.pc.in mylib.pc @ONLY)
```

And finally, install generated file:

```
install(FILES ${CMAKE_BINARY_DIR}/mylib.pc DESTINATION
${CMAKE_INSTALL_DATAROOTDIR}/pkgconfig)
```

You may also use <u>cmake EXPORT feature</u>. However, this feature is only compatible with <u>cmake</u> and I find it difficult to use.

Finally the entire CMakeLists.txt should looks like:

```
cmake_minimum_required(VERSION 3.9)
project(mylib VERSION 1.0.1 DESCRIPTION "mylib description")
include(GNUInstallDirs)
add_library(mylib SHARED src/mylib.c)
set_target_properties(mylib PROPERTIES
    VERSION ${PROJECT_VERSION}
    SOVERSION ${PROJECT_VERSION_MAJOR}
    PUBLIC_HEADER api/mylib.h)
configure_file(mylib.pc.in mylib.pc @ONLY)
target_include_directories(mylib PRIVATE .)
install(TARGETS mylib
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    PUBLIC_HEADER DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
install(FILES ${CMAKE_BINARY_DIR}/mylib.pc
    DESTINATION ${CMAKE_INSTALL_DATAROOTDIR}/pkgconfig)
```

## **EDIT**

As mentioned in comments, to comply with standards you should be able to generate a static library as well as a shared library. The process is bit more complex and does not match with the initial question. But it worths to mention that it is greatly explained <u>here</u>.

Share Improve this answer Follow



answered Aug 23, 2017 at 15:20

Jérôme Pouiller

10.1k • 6 • 43 • 48

- Just complementing the @Jezz's awesome explanation: after all steps above, the programmer can build and install the library by mkdir build && cd build/ && cmake .. && sudo make install (or sudo make install/strip to install the striped library version). silvioprog Jan 16, 2018 at 4:04
- 2 Do you have a technique for passing down library dependencies? For example if mylib depended on liblog4cxx, what would be a good way of flowing that all the way through to mylib.pc? mpr Jul 3, 2018 at 20:45
- 1 @mpr If liblog4cxx provide a .pc file, add Requires: liblog4cxx to your mylib.pc, else, you can just add llog4cxx to Libs: . Jérôme Pouiller Jul 4, 2018 at 7:07
- 2 How would I use this library in another project? Could you extend your example? Damir Porobic Aug 12, 2018 at 8:06
- add\_library should be used without **STATIC/SHARED**, BUILD\_SHARED\_LIBS must be used. cgold.readthedocs.io/en/latest/tutorials/libraries/... None Apr 16, 2021 at 6:06 /



This minimal CMakeLists.txt file compiles a simple shared library:

108





M

```
cmake_minimum_required(VERSION 2.8)

project (test)
set(CMAKE_BUILD_TYPE Release)

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)
add_library(test SHARED src/test.cpp)
```

However, I have no experience copying files to a different destination with CMake. The file command with the COPY/INSTALL signature looks like it might be useful.

Share Improve this answer Follow

cmake\_minimum\_required(VERSION 2.4.0)



answered Jul 7, 2013 at 11:22



52 CMAKE\_BUILD\_TYPE should be omitted, so the decision is up to the one who compiles. – ManuelSchneid3r Sep 30, 2016 at 21:26

Does specifying \$\{CMAKE\_CURRENT\_SOURCE\_DIR}\/ in include\_directories is usefull? - Jérôme Pouiller Aug 23, 2017 at 14:58

@Jezz I don't think so, the same directory gets included without the prefix. It would matter if you were in a subdirectory, however. – Arnav Borborah Mar 16, 2018 at 12:15 /

And what if I want to mix my sources and my headers in a generic "source" directory? Is there a "post generation" possibility to create the "header" directory from my sources? (install commands maybe) – Sandburg Oct 22, 2019 at 7:39



I'm trying to learn how to do this myself, and it seems you can install the library like this:

35







project(mycustomlib)

# Find source files
file(GLOB SOURCES src/\*.cpp)

# Include header files
include\_directories(include)

# Create shared library
add\_library(\${PROJECT\_NAME} SHARED \${SOURCES})

# Install library
install(TARGETS \${PROJECT\_NAME} DESTINATION lib/\${PROJECT\_NAME}))

# Install library headers
file(GLOB HEADERS include/\*.h)
install(FILES \${HEADERS} DESTINATION include/\${PROJECT\_NAME})

2 simplest and straight-forward answer – Foxie Flakey Aug 14, 2022 at 3:17



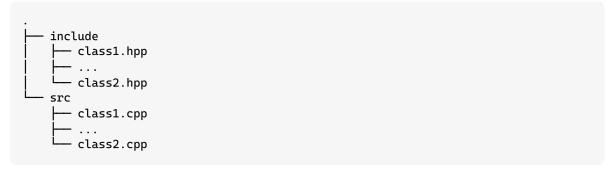
First, this is the directory layout that I am using:

```
29
```





1



After a couple of days taking a look into this, this is my favourite way of doing this thanks to modern CMake:

```
cmake_minimum_required(VERSION 3.5)
project(mylib VERSION 1.0.0 LANGUAGES CXX)
set(DEFAULT_BUILD_TYPE "Release")
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
 message(STATUS "Setting build type to '${DEFAULT_BUILD_TYPE}' as none was
specified.")
  set(CMAKE_BUILD_TYPE "${DEFAULT_BUILD_TYPE}" CACHE STRING "Choose the type of
build." FORCE)
 # Set the possible values of build type for cmake-gui
 set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS "Debug" "Release"
"MinSizeRel" "RelWithDebInfo")
endif()
include(GNUInstallDirs)
set(SOURCE_FILES src/class1.cpp src/class2.cpp)
add_library(${PROJECT_NAME} ...)
target_include_directories(${PROJECT_NAME} PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
   PRIVATE src)
set_target_properties(${PROJECT_NAME}} PROPERTIES
   VERSION ${PROJECT_VERSION}
    SOVERSION 1)
install(TARGETS ${PROJECT_NAME} EXPORT MyLibConfig
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
```

```
RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
install(DIRECTORY include/ DESTINATION
${CMAKE_INSTALL_INCLUDEDIR}/${PROJECT_NAME})

install(EXPORT MyLibConfig DESTINATION share/MyLib/cmake)

export(TARGETS ${PROJECT_NAME} FILE MyLibConfig.cmake)
```

After running CMake and installing the library, there is no need to use Find\*\*\*.cmake files, it can be used like this:

```
find_package(MyLib REQUIRED)

#No need to perform include_directories(...)
target_link_libraries(${TARGET} mylib)
```

That's it, if it has been installed in a standard directory it will be found and there is no need to do anything else. If it has been installed in a non-standard path, it is also easy, just tell CMake where to find MyLibConfig.cmake using:

```
cmake -DMyLib_DIR=/non/standard/install/path ..
```

I hope this helps everybody as much as it has helped me. Old ways of doing this were quite cumbersome.

Share Improve this answer Follow

edited Oct 14, 2018 at 9:37

answered Apr 16, 2018 at 17:27



Perfect answer, I'd remove PRIVATE src from target\_include\_directories since you're not supposed to have them "global" using #include <header.h> , rather #include "header.h" as relative path. – None Apr 1, 2021 at 12:09

What is the purpose of last line: "export(TARGETS \${PROJECT\_NAME} FILE MyLibConfig.cmake)" I can package and work correctly without that – MinhNV Jun 23 at 17:07

To my knowledge, it creates a MyLibConfig.cmake file with the necessary info for the linker to find the compiled library. − Luis Jun 27 at 13:43 ✓