

+ 22

The Case For a Standardized Package Description Format

LUIS CARO CAMPOS



20
22



The Case for a Standardized Package Description Format

Luis Caro Campos



CONAN
C/C++ Package Manager

2022 Annual C++ Developer Survey "Lite"

How do you manage your C++ 1st and 3rd party libraries?

ANSWER CHOICES	RESPONSES	
The library source code is part of my build	69.91%	827
I compile the libraries separately using their instructions	50.89%	602
System package managers (e.g., apt, brew, ...)	38.80%	459
I download prebuilt libraries from the Internet	27.56%	326
Vcpkg	18.93%	224
Conan	18.34%	217
Nuget	9.30%	110
Other (please specify)	8.37%	99
None of the above, I do not have any dependencies	1.35%	16
Total Respondents: 1,183		

Which of these do you find frustrating ...?

2022 Annual C++ Developer Survey "Lite"

	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A SIGNIFICANT ISSUE FOR ME	TOTAL	WEIGHTED AVERAGE
Managing libraries my application depends on	47.63% 563	34.77% 411	17.60% 208	1,182	2.30
Build times	43.94% 515	38.65% 453	17.41% 204	1,172	2.27
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	33.73% 394	40.75% 476	25.51% 298	1,168	2.08
Setting up a development environment from scratch (compiler, build system, IDE, ...)	27.83% 329	42.98% 508	29.19% 345	1,182	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	25.04% 293	46.67% 546	28.29% 331	1,170	1.97
Managing CMake projects	29.34% 343	38.15% 446	32.51% 380	1,169	1.97
Debugging issues in my code	17.85% 209	54.57% 639	27.58% 323	1,171	1.90

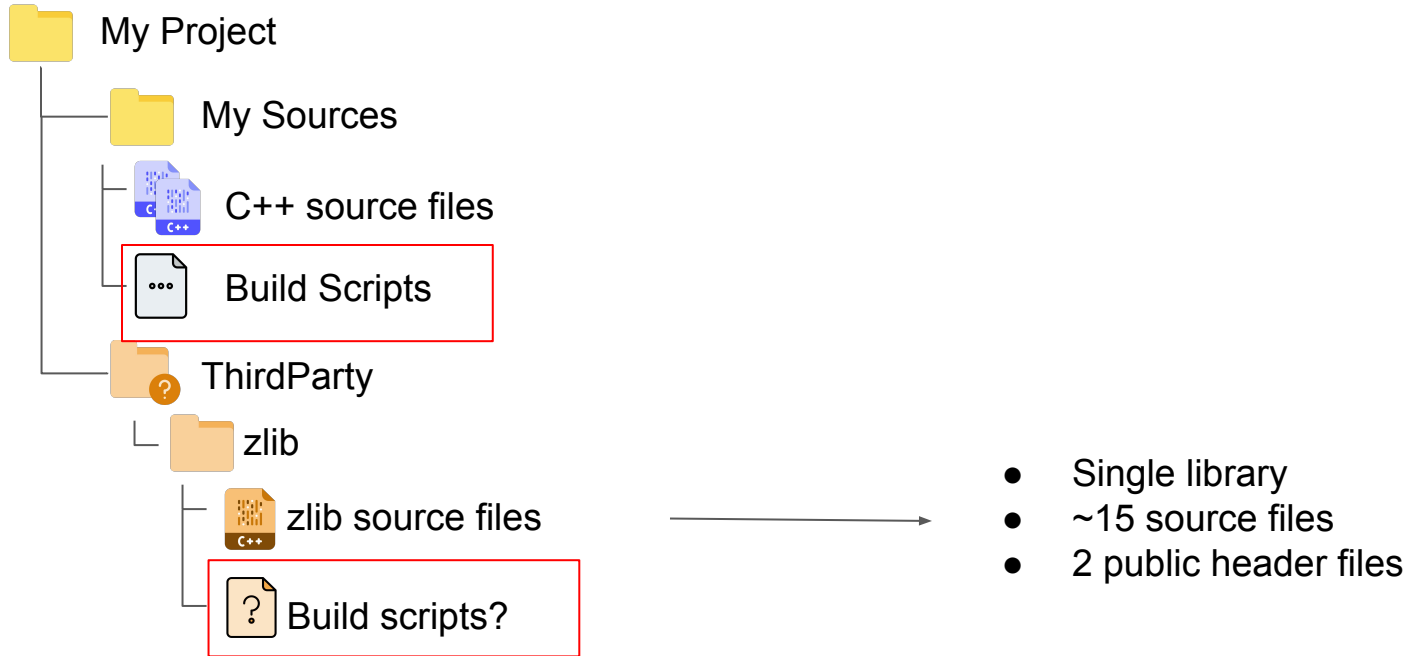
Which of these do you find frustrating ...?

2022 Annual C++ Developer Survey "Lite"

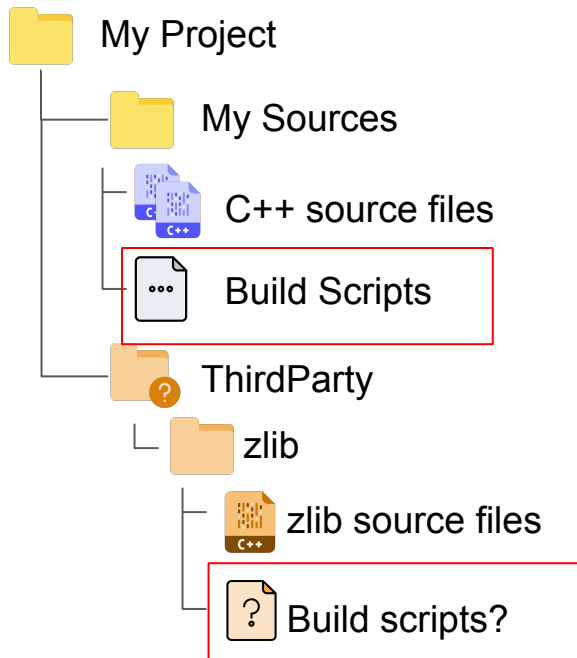
	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A	TOTAL	WEIGHTED
Managing libraries my application depends on	47.63% 563	34.77% 411			
Build times	43.94% 515	38.65% 453			
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	33.73% 394	40.75% 476			
Setting up a development environment from scratch (compiler, build system, IDE, ...)	27.83% 329	42.98% 508			
Concurrency safety: Races, deadlocks, performance bottlenecks	25.04% 293	46.67% 546	28.29% 331	1,170	1.97
Managing CMake projects	29.34% 343	38.15% 446	32.51% 380	1,169	1.97
Debugging issues in my code	17.85% 209	54.57% 639	27.58% 323	1,171	1.90



Consuming third party libraries: ZLIB



Consuming third party libraries: ZLIB



244bbd6f95

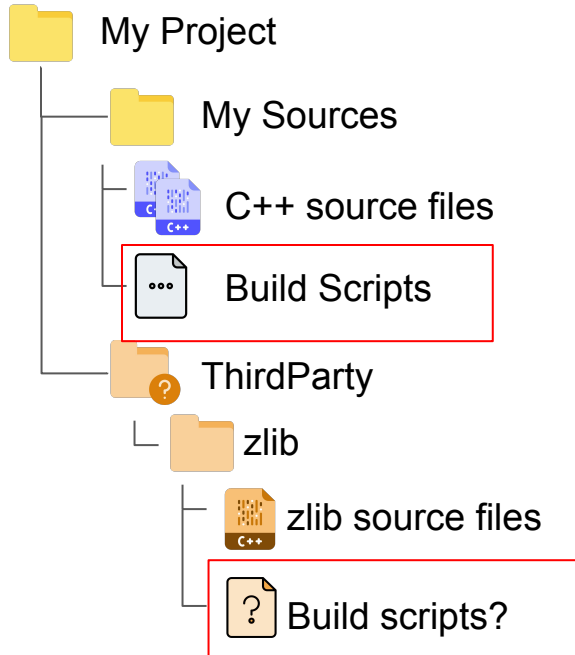
protobuf / third_party / zlib.BUILD

69 lines (63 sloc) 1.68 KB

Raw

```
1 load("@rules_cc//cc:defs.bzl", "cc_library")
2
3 licenses(["notice"]) # BSD/MIT-like license (for zlib)
4
5 exports_files(["zlib.BUILD"])
6
7 _ZLIB_HEADERS = [
8     "crc32.h",
9     "deflate.h",
10    "gzguts.h",
11    "inffast.h",
12    "inffixed.h",
13    "inflate.h",
14    "inftrees.h",
15    "trees.h",
16    "zconf.h",
17    "zlib.h",
18    "zutil.h",
19 ]
20
21 _ZLIB_PREFIXED_HEADERS = ["zlib/include/" + hdr for hdr in _ZLIB_HEADERS]
22
23 # In order to limit the damage from the `includes` propagation
24 # via `:zlib`, copy the public headers to a subdirectory and
25 # expose those.
26 genrule({
```

Consuming third party libraries: ZLIB

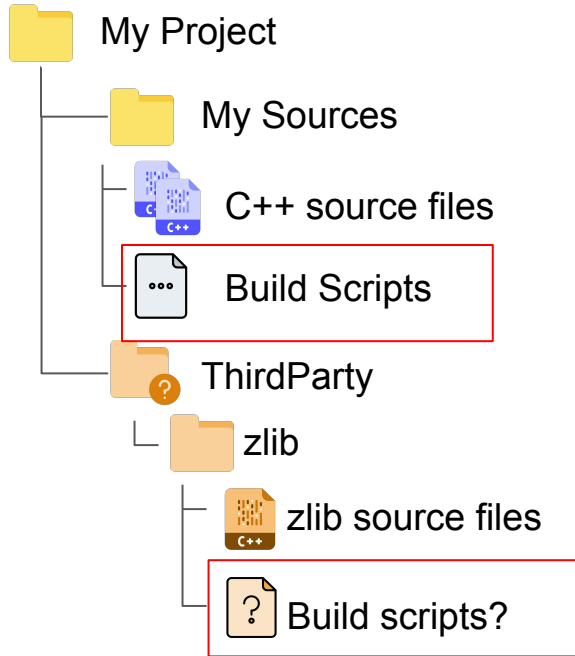


4.x ▾ **opencv** / 3rdparty / zlib / CMakeLists.txt

105 lines (91 sloc) | 2.49 KB

```
1 # -----
2 # CMake file for zlib. See root CMakeLists.txt
3 #
4 # -----
5
6 project(${ZLIB_LIBRARY} C)
7
8 include(CheckFunctionExists)
9 include(CheckIncludeFile)
10 include(CheckCSourceCompiles)
11 include(CheckTypeSize)
12
13 #
14 # Check for fseeko
15 #
16 check_function_exists(fseeko HAVE_FSEEKO)
17 if(NOT HAVE_FSEEKO)
18     add_definitions(-DNO_FSEEKO)
19 endif()
20
21 #
22 # Check for unistd.h
23 #
24 if(NOT MSVC)
25     check_include_file(unistd.h Z_HAVE_UNISTD_H)
```


Consuming third party libraries: ZLIB

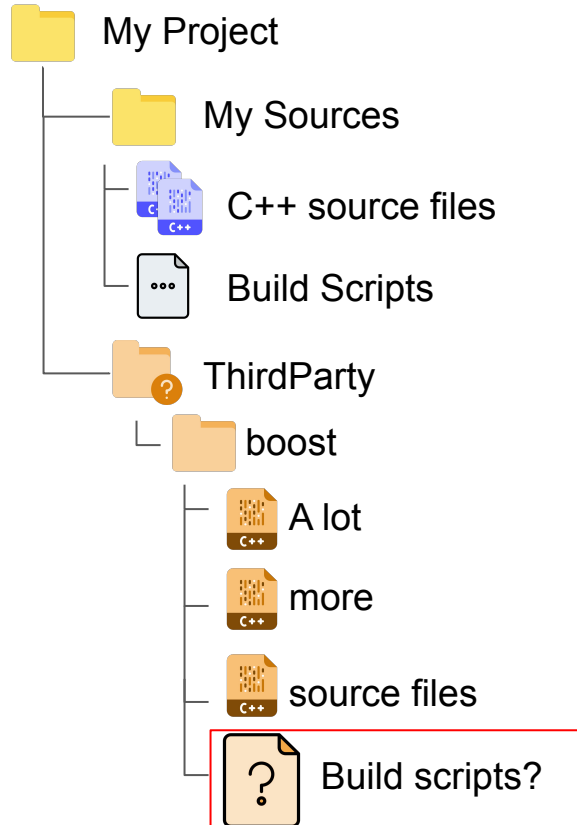


df8bd44564 ▾ qt5-base / src / 3rdparty / zlib.pri

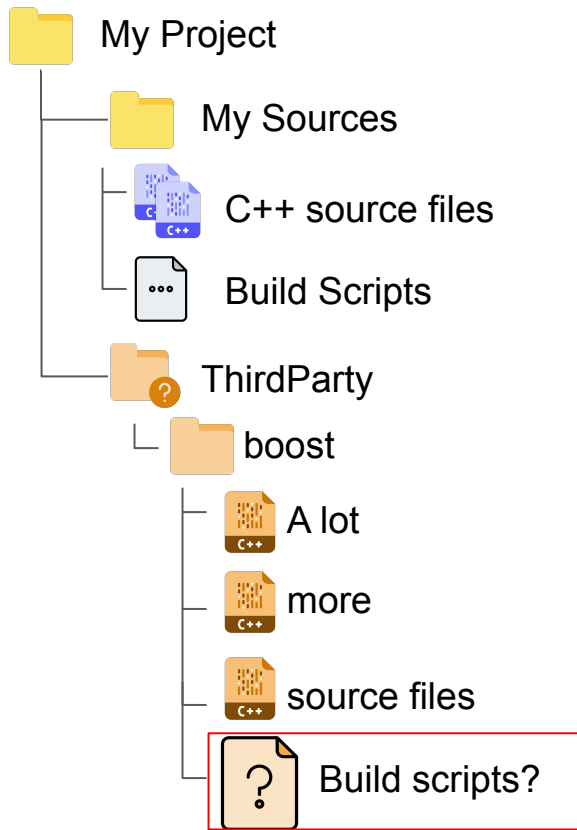
20 lines (19 sloc) | 499 Bytes

```
1 wince: DEFINES += NO_ERRNO_H
2 INCLUDEPATH = $$PWD/zlib $$INCLUDEPATH
3 SOURCES+= \
4     $$PWD/zlib/adler32.c \
5     $$PWD/zlib/compress.c \
6     $$PWD/zlib/crc32.c \
7     $$PWD/zlib/deflate.c \
8     $$PWD/zlib/gzclose.c \
9     $$PWD/zlib/gzlib.c \
10    $$PWD/zlib/gzread.c \
11    $$PWD/zlib/gzwrite.c \
12    $$PWD/zlib/infbac.c \
13    $$PWD/zlib/inffast.c \
14    $$PWD/zlib/inflate.c \
15    $$PWD/zlib/inftrees.c \
16    $$PWD/zlib/trees.c \
17    $$PWD/zlib/uncompr.c \
18    $$PWD/zlib/zutil.c
19
20 TR_EXCLUDE += $$PWD/*
```

Consuming third party libraries: Boost



Consuming third party libraries: Boost



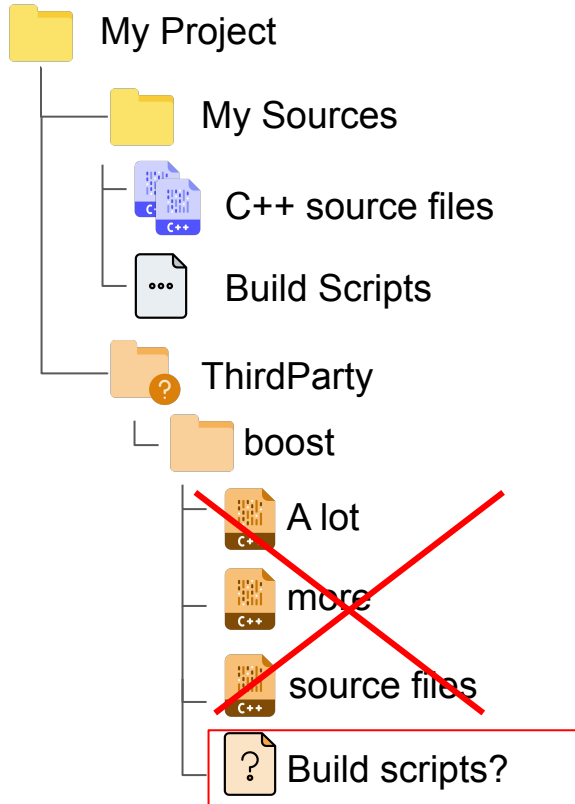
master

rules_boost / BUILD.boost

2511 lines (2311 sloc) | 48.2 KB

```
1  load("@bazel_skylib//rules:copy_file.bzl", "copy_file")
2  load("@bazel_skylib//lib:selects.bzl", "selects")
3  load("@bazel_skylib//rules:common_settings.bzl", "bool_flag")
4  load("@com_github_nelhage_rules_boost//:boost/boost.bzl", "boost_library",
5
6  _w_no_deprecated = selects.with_or({
7      ("@platforms//os:linux", "@platforms//os:osx", "@platforms//os:ios", '
8      "-Wno-deprecated-declarations",
9      },
10     "//conditions:default": [],
11 })
12
13 # Hopefully, the need for these OSxCPU config_setting()s will be obviated
14
15 config_setting(
16     name = "linux_arm",
17     constraint_values = [
18         "@platforms//os:linux",
19         "@platforms//cpu:arm",
20     ],
21 )
22
```

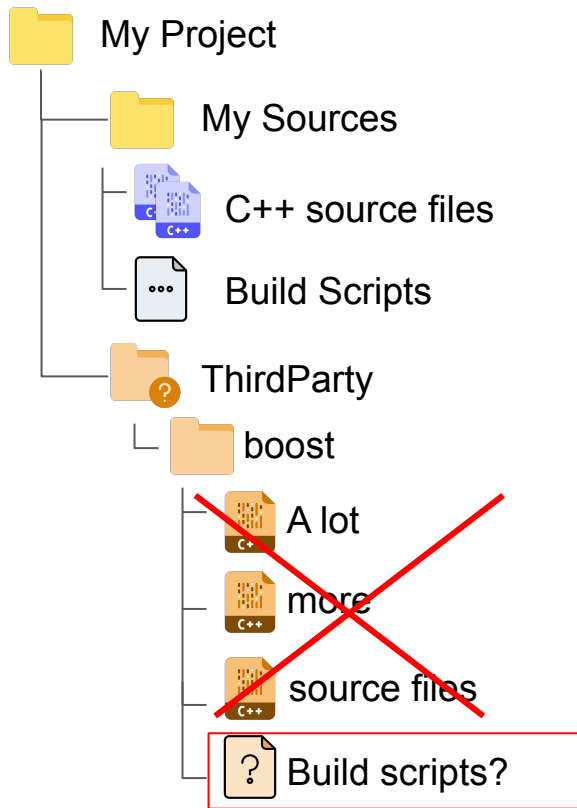
Consuming third party libraries: Boost (cont'd)



```
17 ExternalProject_Add(boost
18   URL "http://dl.dropbox.com/u/15135983/boost.tar.gz"
19   URL_MD5 66f100a77f727e21d67fef1827b6c64
20   BUILD_IN_SOURCE 1
21   UPDATE_COMMAND ""
22   PATCH_COMMAND ""
23   CONFIGURE_COMMAND ${Boost_Bootstrap_Command}
24   BUILD_COMMAND ${Boost_b2_Command} install
25     --without-python
26     --without-mpi
27     --disable-icu
28     --prefix=${CMAKE_BINARY_DIR}/INSTALL
29     --threading=single,multi
30     --link=shared
31     --variant=release
32     -i8
```

github.com/arnaudgelas/ExternalProject

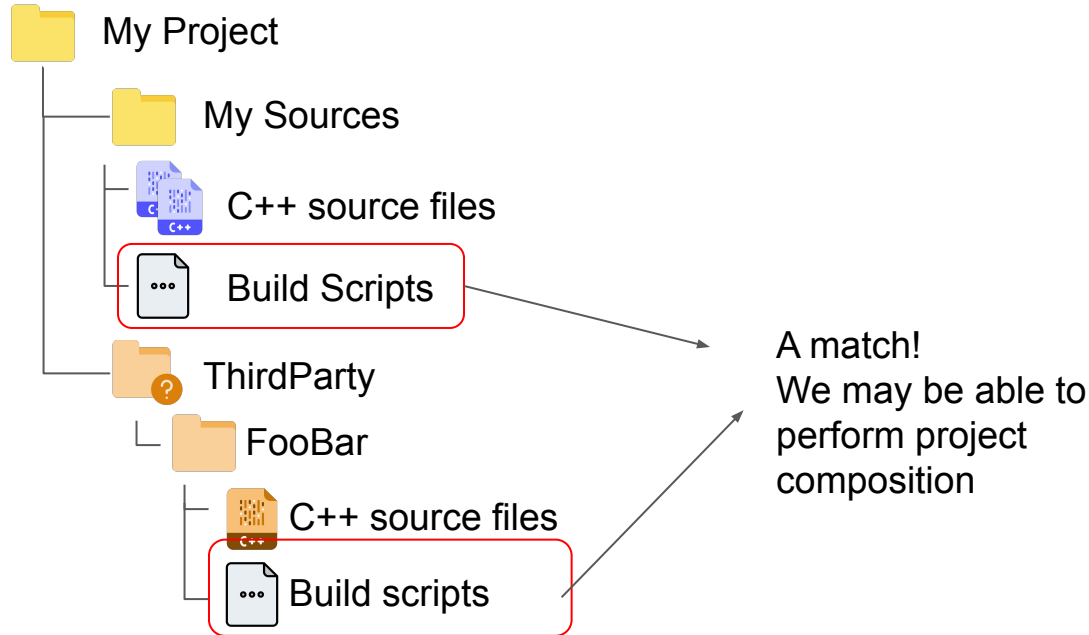
Consuming third party libraries: Boost (cont'd)



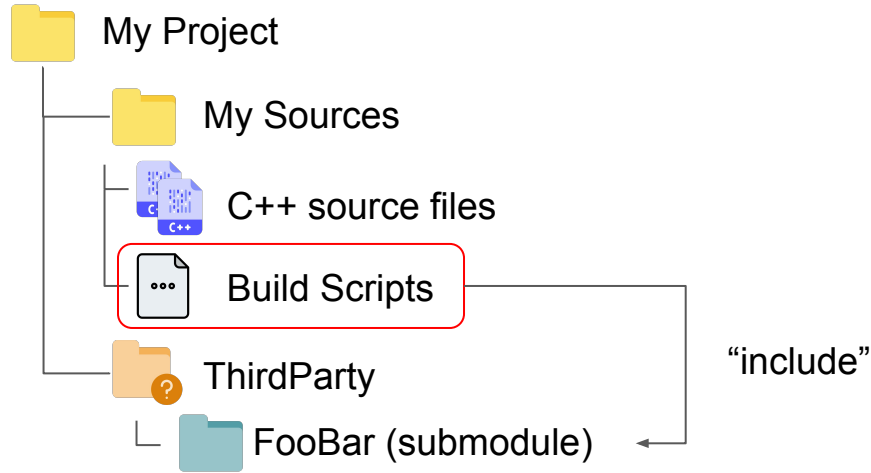
```
20 def _create_configure_script(configureParameters):
21     ctx = configureParameters.ctx
22     root = detect_root(ctx.attr.lib_source)
23
24     return [
25         "cd $INSTALLDIR",
26         "##copy_dir_contents_to_dir## $$EXT_BUILD_ROOT${}/. .".format(root),
27         "chmod -R +w .",
28         "##enable_tracing##",
29         "./bootstrap.sh {}".format(" ".join(ctx.attr.bootstrap_options)),
30         "./b2 install {} --prefix={}".format(" ".join(ctx.attr.user_options)),
31         "##disable_tracing##",
32     ]
33
```

github.com/bazelbuild/rules_foreign_cc

Consuming third party libraries: Another option




Consuming third party libraries: Another option



Header only libraries

- Simplest case
- Small footprint on your build scripts
 - Only need to expose an include directory, no need to involve the compiler or linker
- **More than 1 in 4 recipes** in ConanCenter are header only
- Library authors know this

Header only libraries

 r/cpp · Posted by u/barfyus 1 month ago

AsyncCppRpc - asynchronous transport-agnostic header-only C++ RPC library

64 upvotes 15 comments 0 awards

 r/cpp · Posted by u/Able_Armadillo491 7 months ago

A header only c++17 structure of arrays implementation

58 upvotes 28 comments 1 award

 r/cpp · Posted by u/TheCompiler95 3 days ago

ptc-print: a C++17 header-only library for custom printing to the out (basically a detailed implementation of the Python print() function w

 r/cpp · Posted by u/frozenca 2 months ago

A header-only STL-like C++20 B-Tree with disk file support

[github.com/frozen...](https://github.com/frozenca)

47 upvotes 4 comments 0 awards

 r/cpp · Posted by u/foldingarmour 2 months ago

morphologica: A header-only library for high performance OpenGL data visualization and plotting in C++

56 upvotes 13 comments 0 awards

 r/cpp · Posted by u/Ganofir 5 months ago

Goose - a small header only library for printing STL-like collections

23 upvotes 5 comments 0 awards

Header only libraries



A curated list of awesome header-only C++ libraries



 [awesome-hpp](#) Public

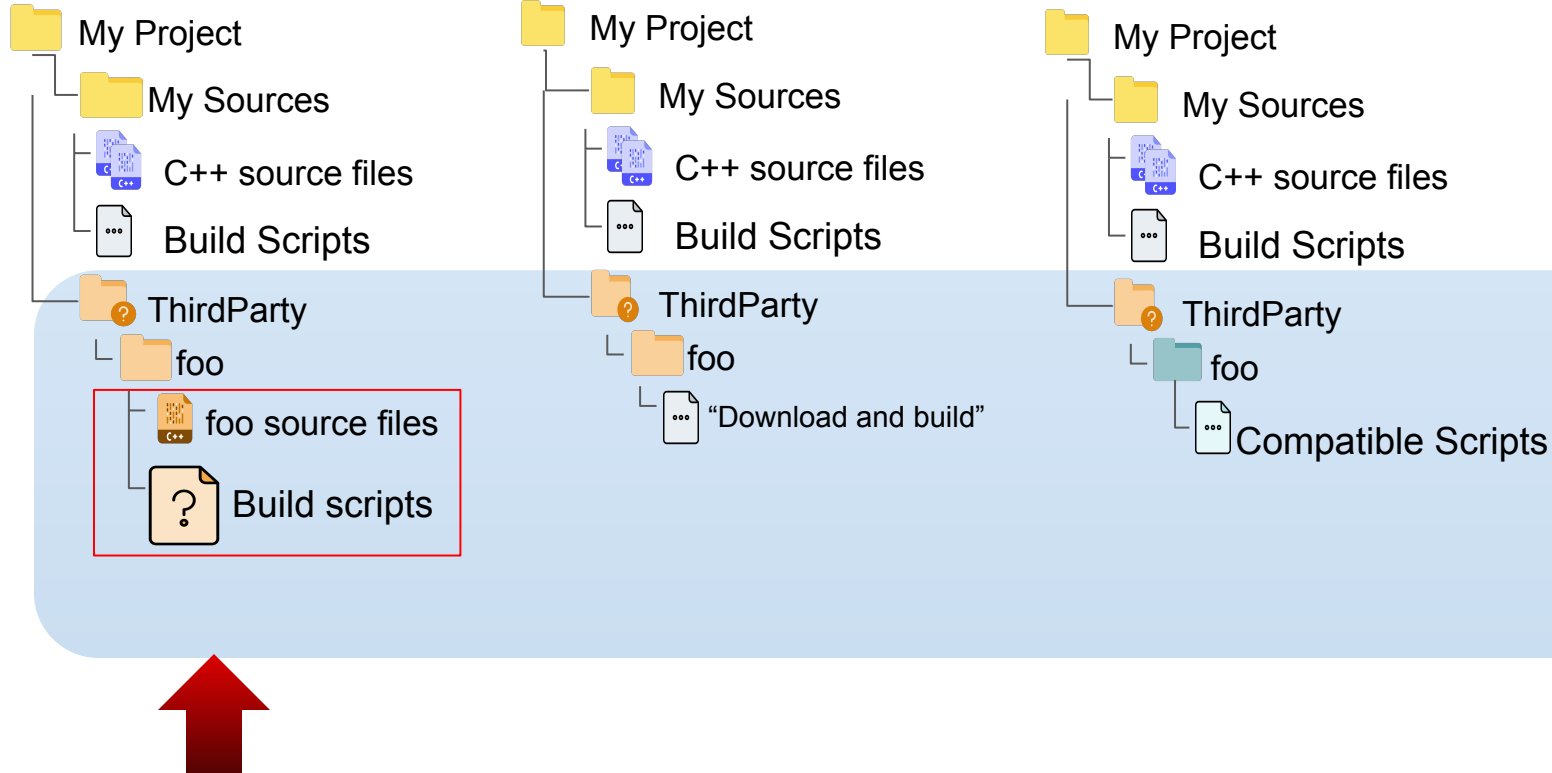
A curated list of awesome header-only C++ libraries

☆ 2k 👤 125

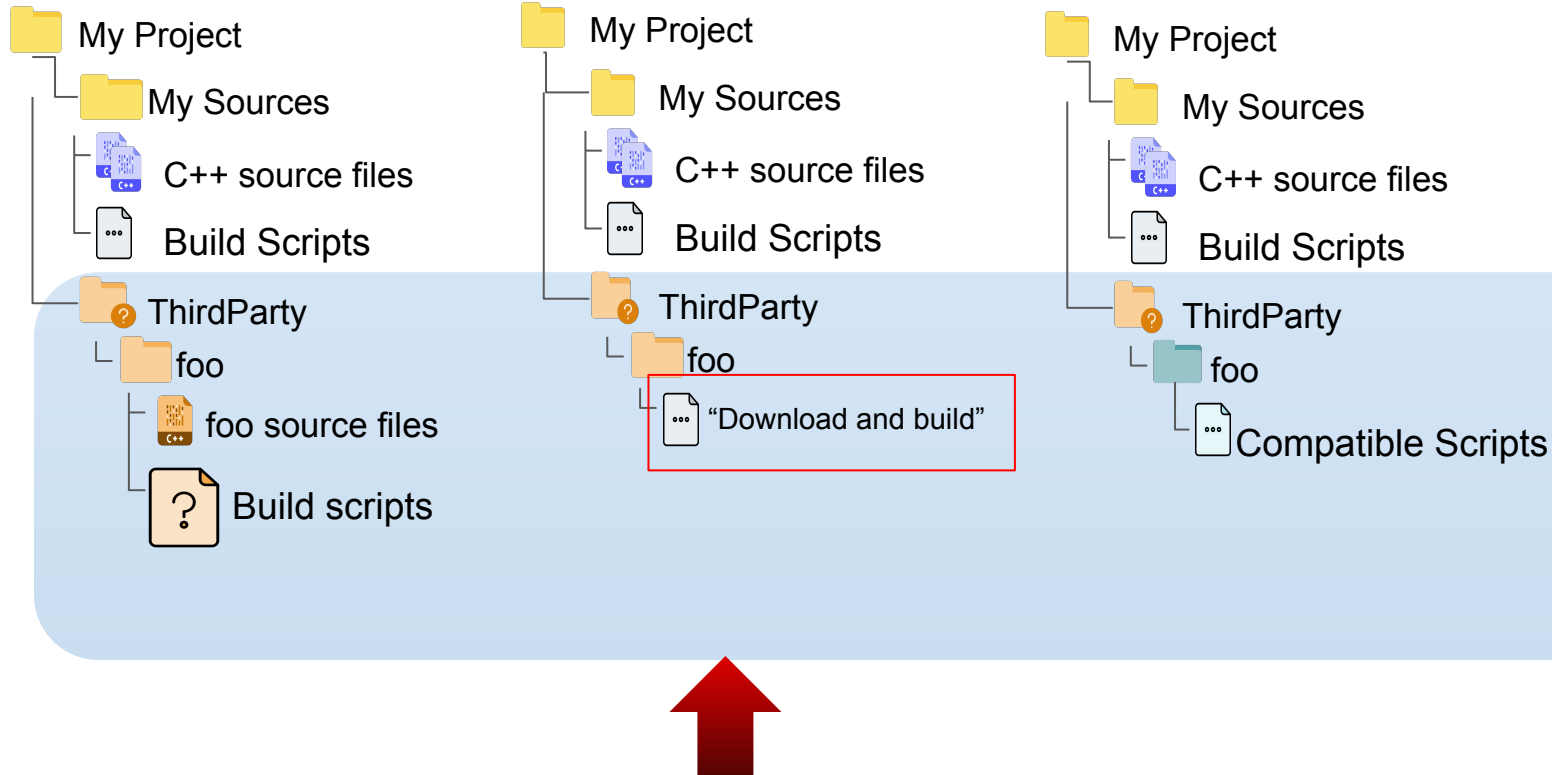
Recap

- All examples so far are different ways of **vendoring** your dependencies inside your project
- When **building** a library or application that vendors dependencies - it's very convenient
 - It might “just work”
- But when you **depend on** a library that vendors dependencies - not so much!
 - This is a big headache for package repository maintainers (especially Linux distros)
 - The dependencies of this library **might conflict** with your own dependencies
 - For library authors that want their libraries available in these public repositories, they'll have to provide a way of consuming dependencies externally

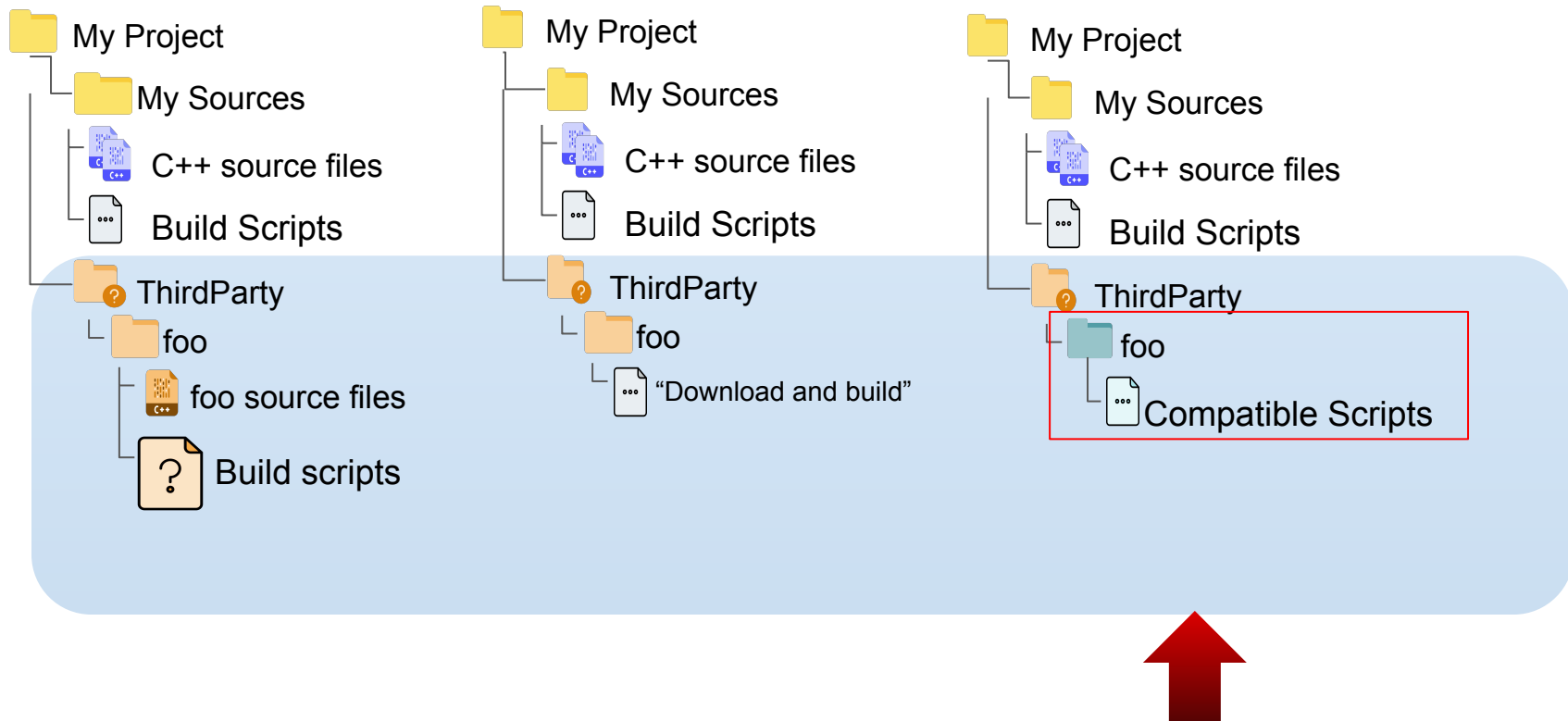
Consuming Libraries



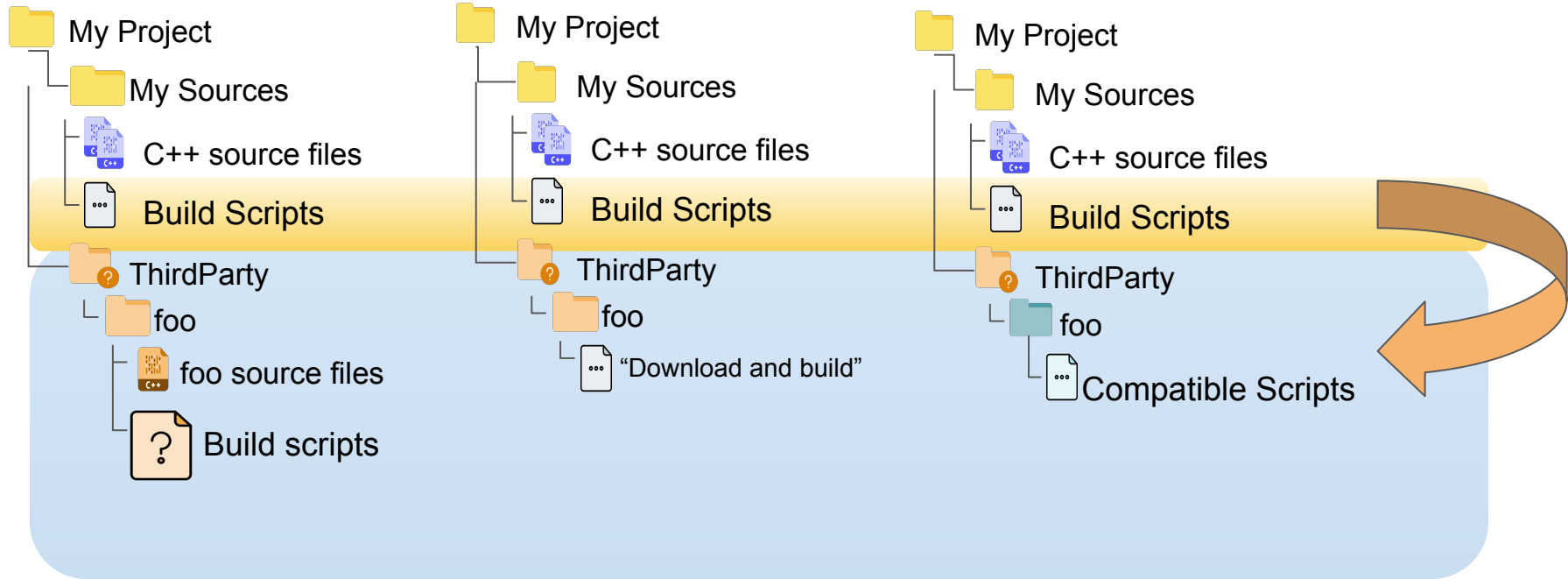
Consuming Libraries



Consuming Libraries



Consuming Libraries



Consuming Libraries (cont'd)

- How we refer to libraries depends on the abstractions provided by the build system we are using (CMake, Bazel, Makefiles, Visual Studio or Xcode Projects...)
 - The “modern” way is based on “usage requirements” -
 - But in some cases we still see build scripts that propagate “flags” explicitly



/ 14

715 qt_internal_extend_target(Core **CONDITION NOT** QT_FEATURE_system_zlib

716 LIBRARIES

717 Qt::ZlibPrivate

718)

github.com/qt/qtbase

Consuming Libraries (cont'd)

- How we refer to libraries depends on the abstractions provided by the build system we are using (CMake, Bazel, Makefiles, Visual Studio or Xcode Projects...)
 - The “modern” way is based on “usage requirements” -
 - But in some cases we still see build scripts that propagate “flags” explicitly

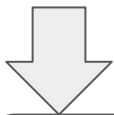


```
135     copts = COPTS,  
136     includes = ["src/"],  
137     linkopts = LINK_OPTS,  
138     visibility = ["//visibility:public"],  
139     deps = [":protobuf_lite"] + select({  
140         "//build_defs:config_msvc": [],  
141         "//conditions:default": ["@zlib//:zlib"],  
142     }),  
143 )
```

github.com/protocolbuffers/protobuf

Usage requirements

```
add_executable(hello hello.cpp)
target_link_libraries(hello PRIVATE Boost::filesystem)
```



This is what happens under the hood - flags are passed to the compiler and linker

```
c++ -DBOOST_ATOMIC_DYN_LINK -DBOOST_ATOMIC_NO_LIB -DBOOST_FILESYSTEM_DYN_LINK
-DBOOST_FILESYSTEM_NO_LIB -isystem /path/to/boost/include -MD -MT hello.cpp.o -MF
hello.cpp.o.d hello.cpp.o -c hello.cpp
```

```
c++ -Wl,-search_paths_first -Wl,-headerpad_max_install_names hello.cpp.o -o hello
-Wl,-rpath,/path/to/boost/lib /path/to/boost/lib/libboost_filesystem.dylib
/path/to/boost/lib/libboost_atomic.dylib
```

Usage requirements (cont'd)

Our code might interact with external libraries by referring to entities like these:


```
Qt::ZlibPrivate
```

```
Boost::filesystem
```

```
@boost//:algorithm
```

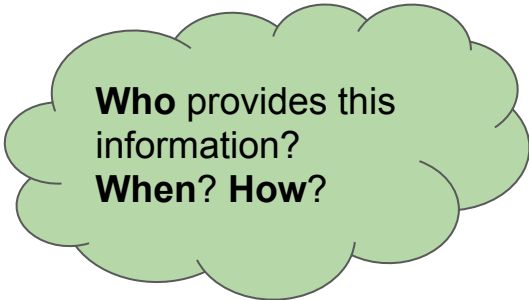
Depending on the context, these are then translated to compile and link flags.

In CMake these are called “targets”



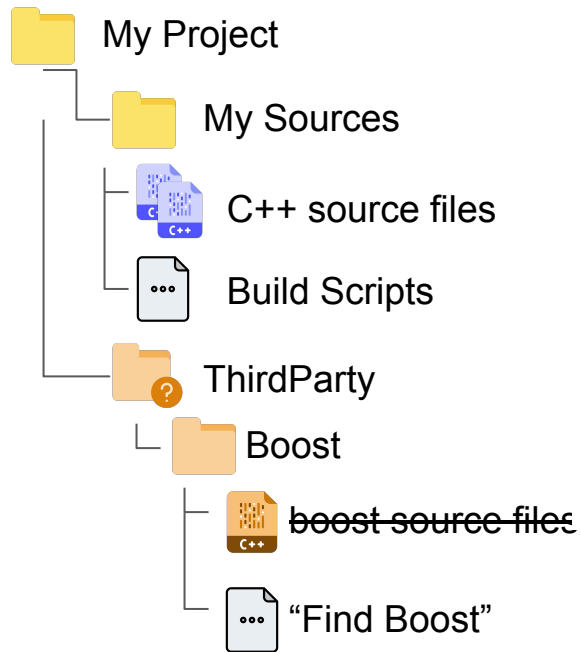
Who performs this “translation”?

This will typically be a feature of the (meta) build system: CMake, Bazel, Meson, B2, etc.



Who provides this information?
When? How?

Find logic



master CMake / Modules / FindBoost.cmake

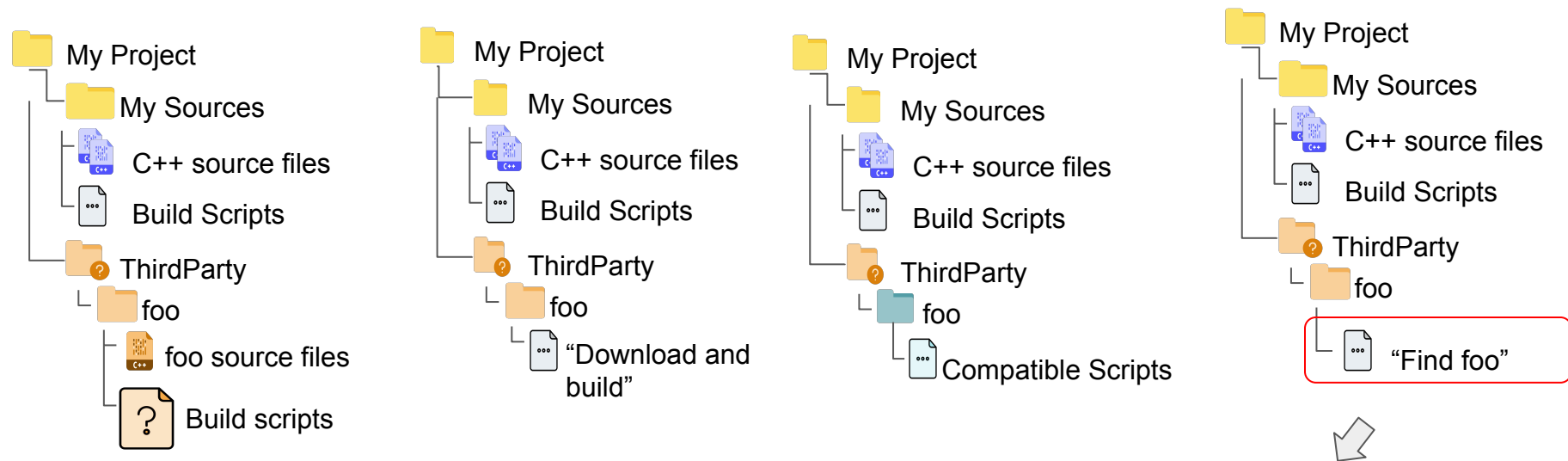
bradking FindBoost: Add support for Boost 1.79 ...

66 contributors

2589 lines (2322 sloc) | 114 KB

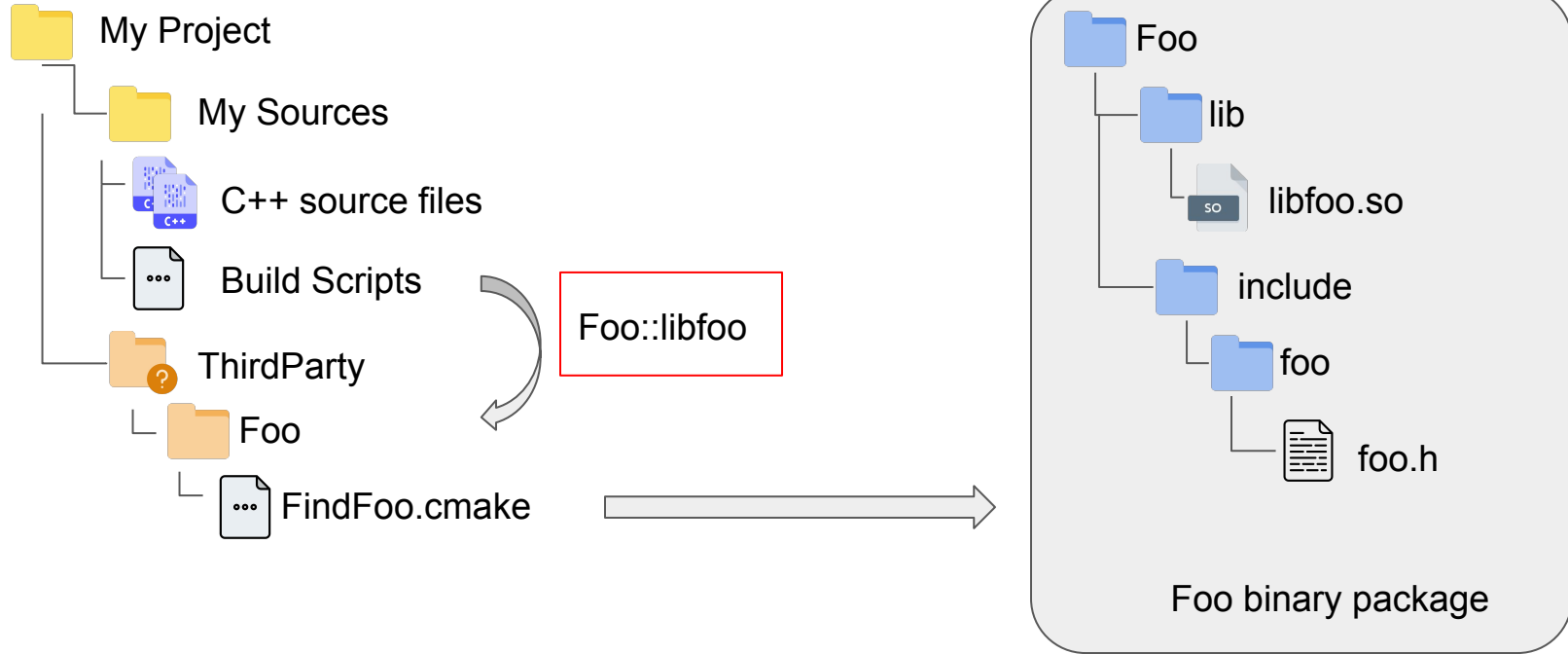
```
1 # Distributed under the OSI-approved BSD 3-Clause License. See accompanying
2 # file Copyright.txt or https://cmake.org/licensing for details.
3
4 #[=====].rst:
5 FindBoost
6 -----
```

Consuming Libraries

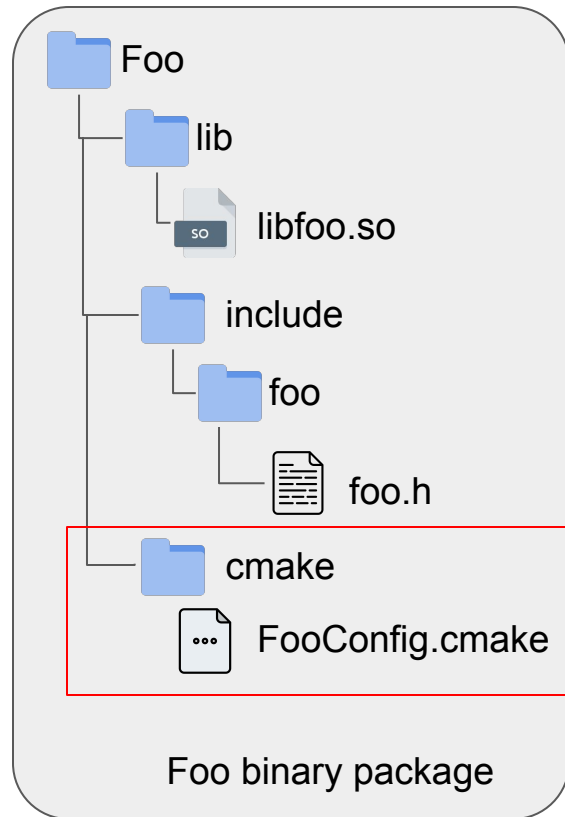
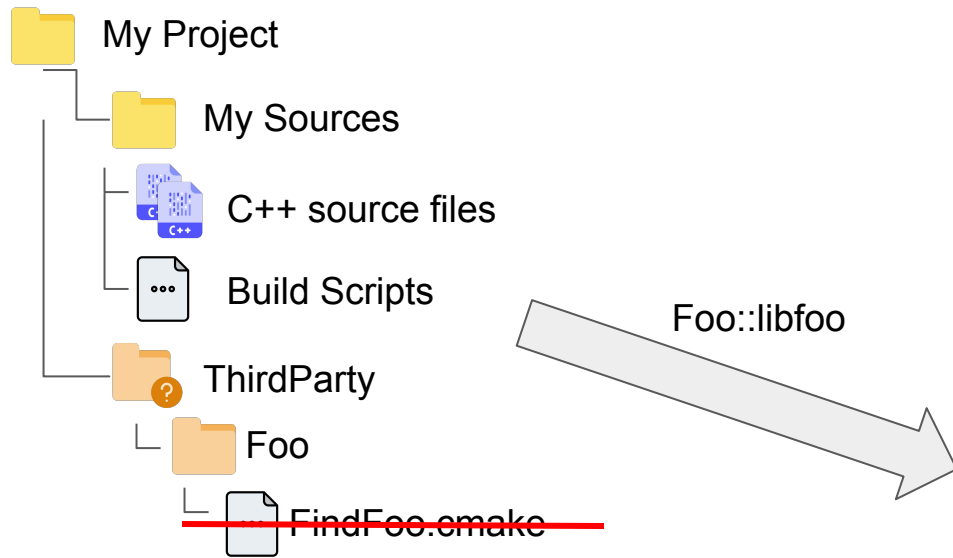


We can now consume libraries
“externally provided” -
As long as they satisfy the
assumptions we made about
them!

Encapsulation to the rescue



Encapsulation to the rescue



Config files packaged with binaries

- To “consume” a library, all we need now is:
 - The **package name**
 - The **component name**
- The following can now be **private** as far as ***our*** build scripts are concerned:
 - The file structure and filenames of the source files for the dependency
 - The build system and how to invoke it
 - The filenames of the compiled libraries
 - The file structure of the compiled “package”
 - The internal dependency graph (which sub-components depend on which others)

Config files packaged with binaries

ISO C++ Developer Survey:

How do you manage first and third party libraries?

ANSWER CHOICES	RESPONSES	
The library source code is part of my build	69.91%	827
I compile the libraries separately using their instructions	50.89%	602
System package managers (e.g., apt, brew, ...)	38.80%	459
I download prebuilt libraries from the Internet	27.56%	326
Vcpkg	18.93%	224
Conan	18.34%	217

Config files packaged with binaries - disadvantages

- There are still a few unanswered questions:
 - how does our build system know where to locate this file?
 - Who and when is this file generated?
- And a big shortcoming:
 - CMake package config files (the current gold standard) are **not build system agnostic**
 - They are full-fledged CMake scripts and may contain statements rather than just being descriptive

CMake package config files

 CMake >  CMake > Issues > #20106



Issue created 2 years ago by  **Brad King** Owner

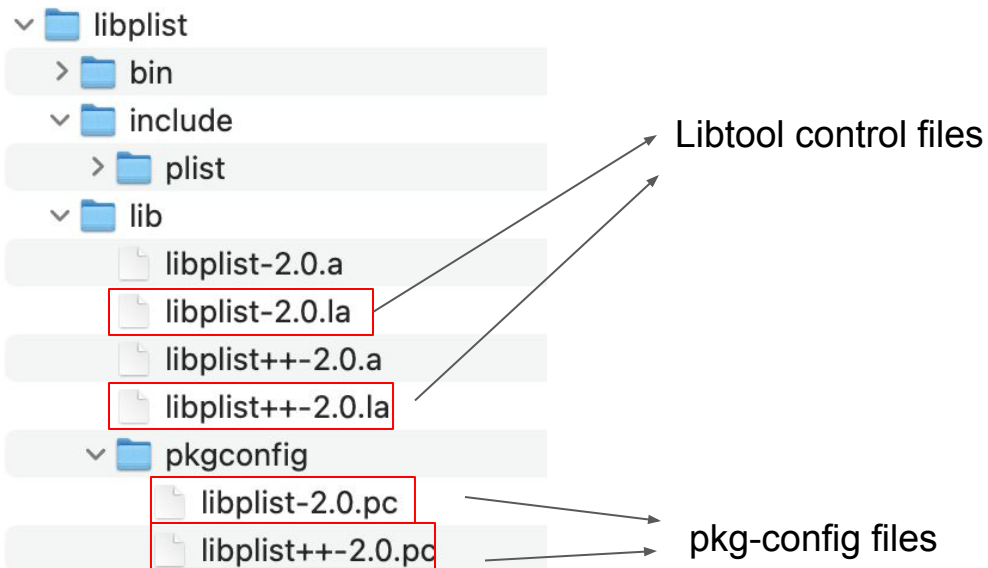


Buildsystem-agnostic package export format

The CMake package approach described in the [cmake-packages\(7\)](#) manual works well for consuming packages from CMake, but not for other tools or build systems. Ideally we should offer a format for `install(EXPORT)` and `export()` that is independent of CMake and more accessible to other tools.

pkg-config and Libtool

- Including special files alongside binary artifacts is far from a new concept
 - pkg-config and Libtool have been doing it for a really long time



Libtool

- It parses the information *implicitly* when compiler is invoked via libtool.

```
libtool --mode=link gcc -g -O -o test test.o /path/to/lib/libhello.la
```



```
gcc -g -O -o .libs/test test.o -Wl,--rpath -Wl,/path/to/lib /path/to/lib/libhello.a -lm
```

pkg-config

```
prefix=/usr
exec_prefix=${prefix}
includedir=${prefix}/include
libdir=${exec_prefix}/lib
```

```
Name: foo
Description: The foo library
Version: 1.0.0
Cflags: -I${includedir}/foo
Libs: -L${libdir} -lfoo
```

foo.pc

```
prefix=/usr
exec_prefix=${prefix}
includedir=${prefix}/include
libdir=${exec_prefix}/lib
```

```
Name: bar
Description: The bar library
Version: 2.1.2
Requires.private: foo >= 0.7
Cflags: -I${includedir}
Libs: -L${libdir} -lbar
```

bar.pc

```
gcc `pkg-config --cflags --libs bar` -o myapp myapp.c
```



-I/usr/include/foo

Libtool and pkg-config today

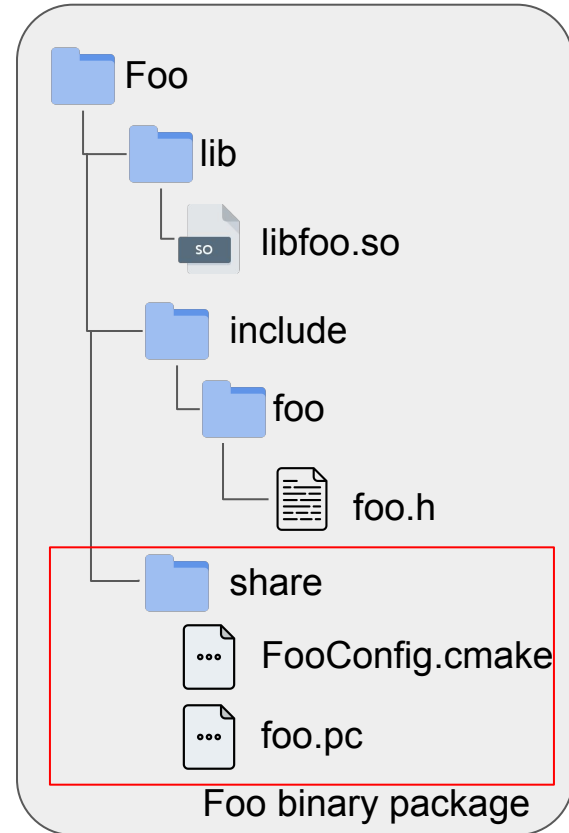
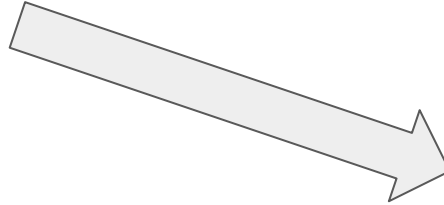
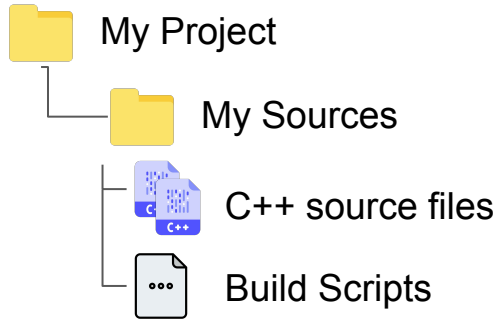
Libtool

- Requires buildsystem to explicitly invoke libtool to call the compiler for us
- Typically limited to projects that abide by the GNU Build System conventions
- Most use cases are now covered by pkg-config - distro maintainers prefer this

pkg-config

- Still very popular in Unix-like environments
- A .pc file can only describe a ***single component***
- It's oriented to flags - rather than describing properties
 - There's a desire to isolate build script maintainers from globals and rely on transitive usage requirements instead

Where we are today



Recap

- We have our project, our sources, our build scripts
- We want **our** build scripts to **not** be concerned about:
 - External library source files
 - How those files are built
- But we do want to be able to consume libraries in our code (compile+link)
- We want to be agnostic as to how/where the library was built
- The current approaches all have limitations

What is a *library*?



libfoo.so

What is a *library*?

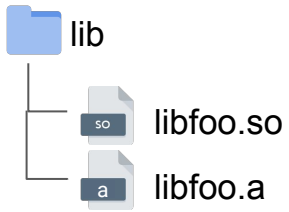


libfoo.so

Dynamic linker

Static linker

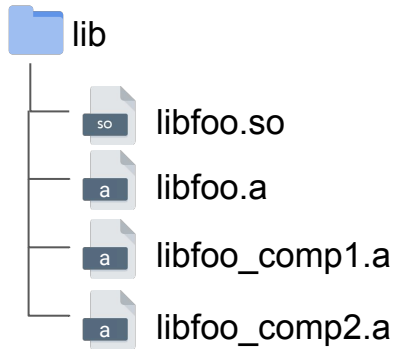
What is a *library*?



Dynamic linker

Static linker

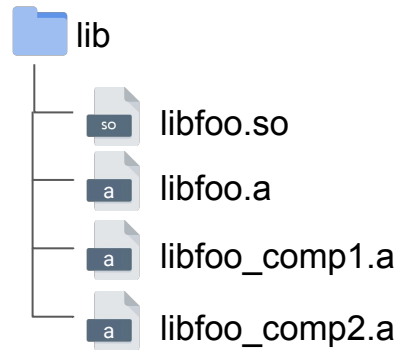
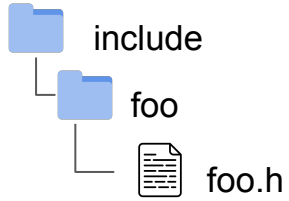
What is a *library*?



Dynamic linker

Static linker

What is a *library*?

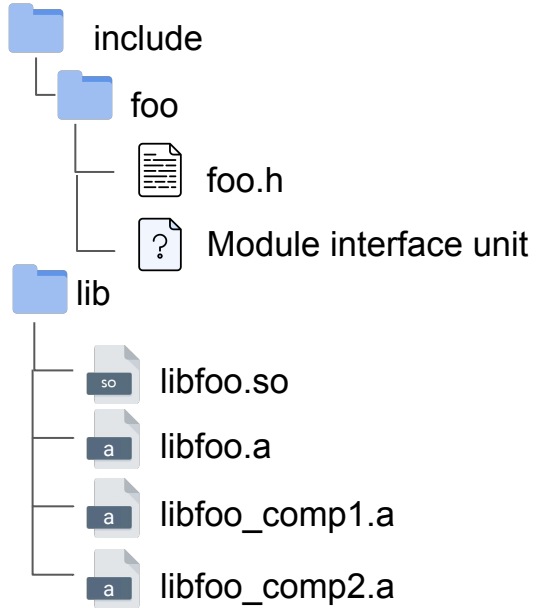


Compiler

Dynamic linker

Static linker

What is a *library*?

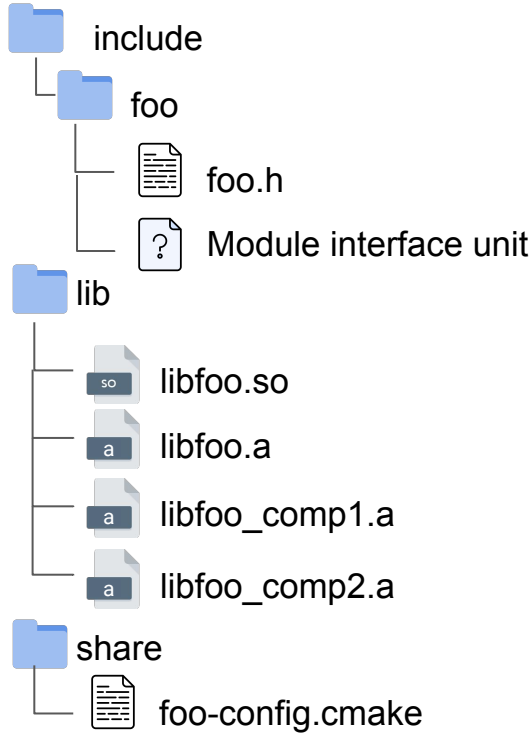


Compiler

Dynamic linker

Static linker

What is a *library*?



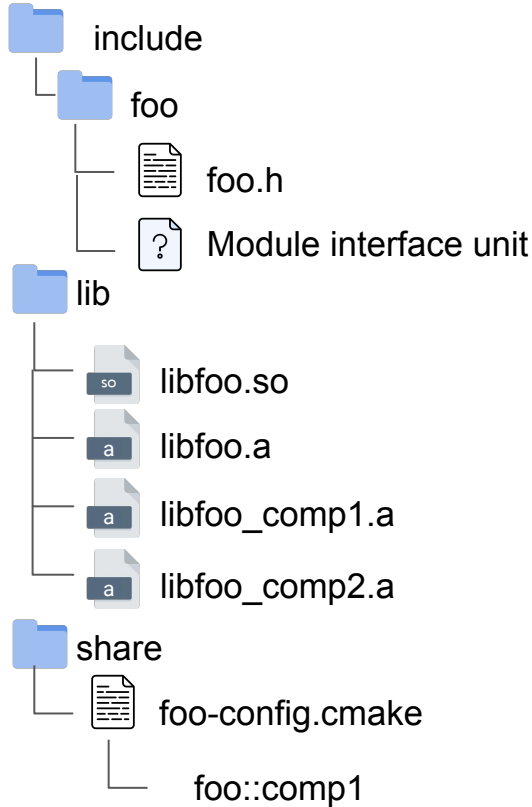
Compiler

Dynamic linker

Static linker

Build system

What is a *library*?



Compiler

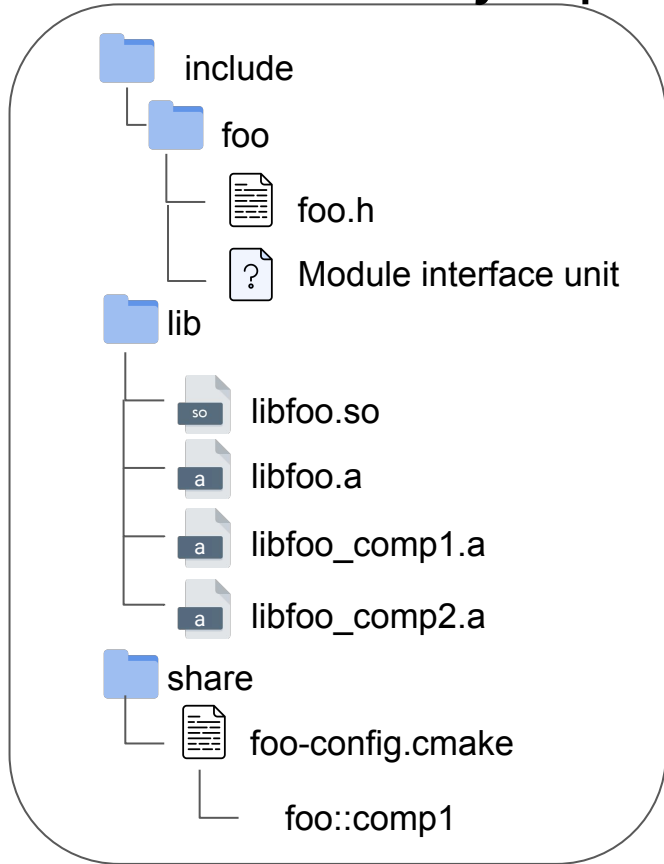
Dynamic linker

Static linker

Build system

Developer

What is a ~~library~~? package



Compiler

Dynamic linker

Static linker

Build system

Package
Manager

Developer

The way forward

- The goal: make it easier for developers to *use* libraries in their build scripts

```
target_link_libraries(my_awesome_app PRIVATE Boost::filesystem)
```

- This starts with build system support:
 - **Transitive usage requirements** in favor of compile/link flags
 - Ability to load and parse the information from an externally provided **package description** file
 - **Interoperability:**
 - Different build systems can opt to implement support for a common format
 - Completely decoupled from the build system of the libraries we are consuming

Proposed approaches



Common Package Specification (P1313R0)

Document: P1313R0

Date: 2018-10-07

SG15 - Tooling

Author: **Matthew Woehlike**

```
{
  "Name": "sample",
  "Description": "Sample CPS",
  "Version": "1.2.0",
  "Compat-Version": "0.8.0",
  "Platform": {
    "Isa": "x86_64",
    "Kernel": "linux",
  },
  "Configurations": [ "Optimized", "Debug" ],
  "Default-Components": [ "sample" ],
  "Components": {
    "sample-core": {
      "Type": "interface",
      "Definitions": [ "SAMPLE" ],
      "Includes": [ "@prefix@/include" ]
    },
    "sample": {
      "Type": "dylib",
      "Requires": [ ":sample-core" ],
      "Configurations": {
        "Optimized": {
          "Location":
"@prefix@/lib64/libsample.so.1.2.0"
        },
        "Debug": {
          "Location":
"@prefix@/lib64/libsample_d.so.1.2.0"
        }
      }
    }
  }
  ...
  ...
}
```

Proposed approaches (cont'd)

cxx-libmanR1

libman, A Dependency Manager → Build System Bridge

Author: Colby Pike

Date: 2019

- Text files with a specific syntax
- Two concepts:
 - Packages and Libraries

```
# A merged Qt5
Type: Package
Name: Qt5
Namespace: Qt5

# Some things we might require
Requires: OpenSSL
Requires: Xcb

# Qt libraries
Library: Core.lml
Library: Widgets.lml
Library: Gui.lml
Library: Network.lml
Library: Quick.lml
# ... (Qt has many libraries)
```

```
# Boost.System
Type: Library
Name: system
Uses: Boost/boost
Path: lib/libboost_system.a
```

Libman (cont'd)

§ 1.3. Goals and Non-Goals

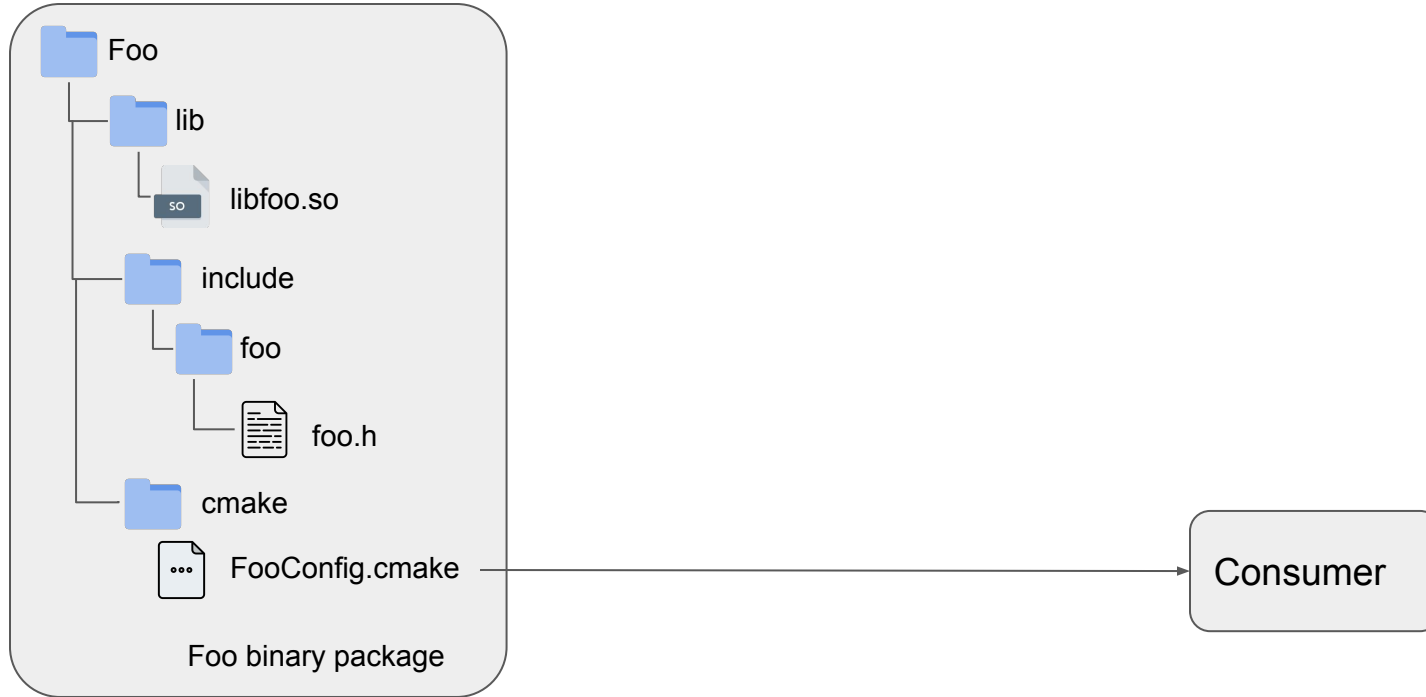
The following are the explicit goals of `libman` and this document:

1. Define a series of file formats which tell a build system how a library is to be "used"
2. Define the semantics of how a build system should interact and perform name-based package and dependency lookup in a deterministic fashion with no dependence on "ambient" environment state.
3. Define the requirements from a PDM for generating a correct and coherent set of `libman` files.

Perhaps just as important as the goals are the non-goals. In particular, `libman` **does not** seek to do any of the following:

1. Define the semantics of ABI and version compatibility between libraries
2. Facilitate dependency resolution beyond trivial name-based path lookup
3. Define a distribution or packaging method for pre-compiled binary packages
4. Define or aide package retrieval and extraction
5. Define or aide source-package building

The role of the package manager



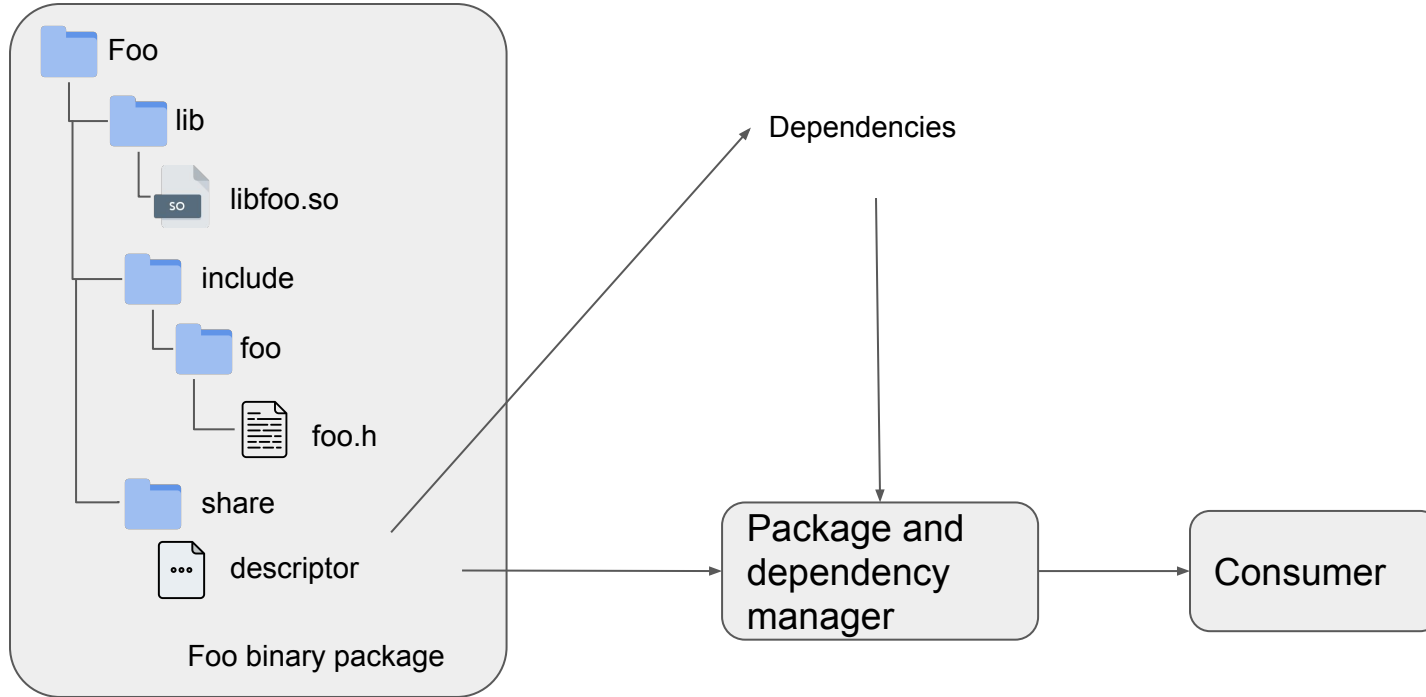
The role of the package manager (cont'd)

- We now have modern, dedicated C++ package managers
 - Dedicated support to build libraries from source
 - Ability to model ABI compatibility
 - Integrations with build systems...
 - Ability to resolve dependency-graphs with C++ concepts (visibility, transitivity, package variants/options).
- There's also a fundamental aspect:
 - The package manager knows *where*

```
c++ -DBOOST_ATOMIC_DYN_LINK -DBOOST_ATOMIC_NO_LIB -DBOOST_FILESYSTEM_DYN_LINK  
-DBOOST_FILESYSTEM_NO_LIB -isystem /path/to/boost/include -MD -MT hello.cpp.o -MF  
hello.cpp.o.d hello.cpp.o -c hello.cpp
```

```
c++ -Wl,-search_paths_first -Wl,-headerpad_max_install_names hello.cpp.o -o hello  
-Wl,-rpath,/path/to/boost/lib /path/to/boost/lib/libboost_filesystem.dylib  
/path/to/boost/lib/libboost_atomic.dylib
```

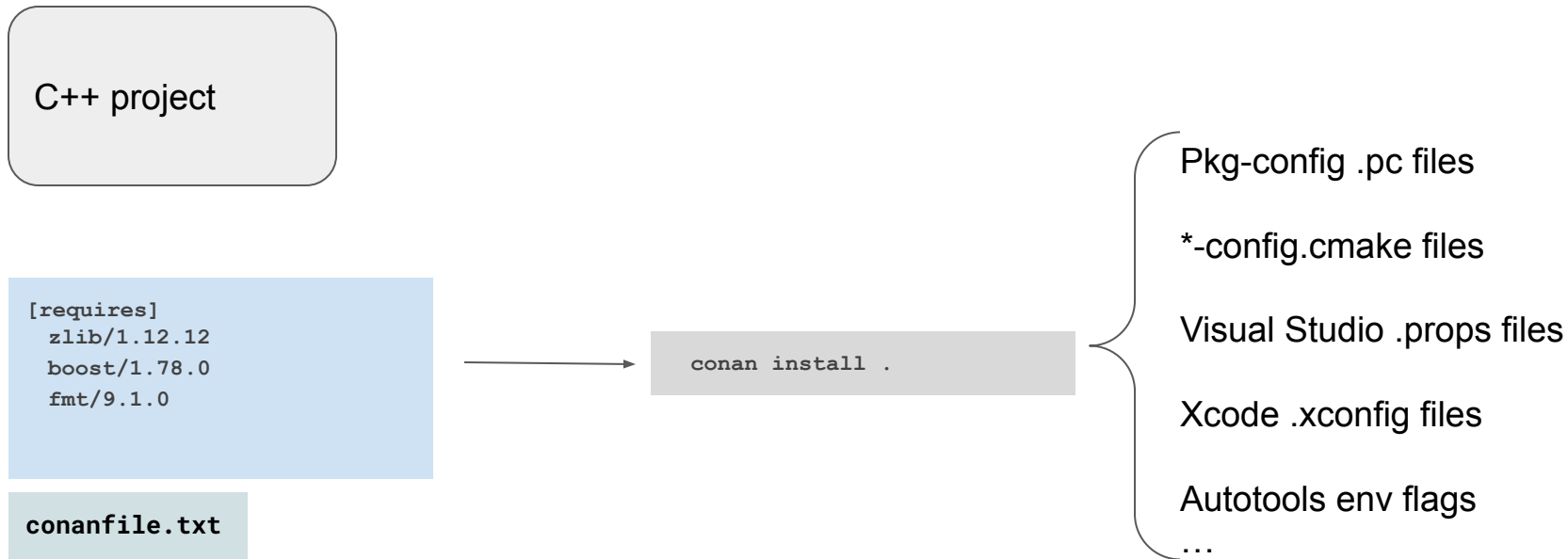

The role of the package manager (cont'd)



The role of the package manager (cont'd)

- Generated pkg-config, libtool and exported CMake files *often* contain **hardcoded absolute paths**
 - Pointing to locations that may only exist in the machine the package was built
 - This must be known at package creation time (and in some cases, at compile time)
 - Imposing constraints on where consumers can place precompiled binary packages
- This may be okay in some scenarios:
 - System wide package managers (apt, homebrew, ...)
 - Workflows that impose building from source on the machine consuming the package

Interoperability with Conan



Interoperability with Conan (cont'd)

- This functionality requires recipes expressing all the information that would otherwise be contained in a package description file
 - Transitive usage requirements: compile flags, include directories, dependencies between library components

```
if self.options.qttools and self.options.gui and self.options.widgets:
    self.cpp_info.components["qtLinguistTools"].set_property("cmake_target_name", "Qt6::LinguistTools")
    self.cpp_info.components["qtLinguistTools"].names["cmake_find_package"] = "LinguistTools"
    self.cpp_info.components["qtLinguistTools"].names["cmake_find_package_multi"] = "LinguistTools"
    _create_module("UiPlugin", ["Gui", "Widgets"])
    self.cpp_info.components["qtUiPlugin"].libs = [] # this is a collection of abstract classes, so this is h
    self.cpp_info.components["qtUiPlugin"].libdirs = []
    _create_module("UiTools", ["UiPlugin", "Gui", "Widgets"])
    _create_module("Designer", ["Gui", "UiPlugin", "Widgets", "Xml"])
    _create_module("Help", ["Gui", "Sql", "Widgets"])
```


github.com/conan-io/conan-center-index: Conan recipe for Qt6

Interoperability with Conan (cont'd)

- This functionality requires recipes expressing all the information that would otherwise be contained in a package description file
 - Transitive usage requirements: compile flags, include directories, dependencies between library components

```
if self.options.qttools and self.options.gui and self.options.widgets:
    self.cpp_info.components["qtLinguistTools"].set_property("cmake_target_name", "Qt6::LinguistTools")
    self.cpp_info.components["qtLinguistTools"].names["cmake_find_package"] = "LinguistTools"
    self.cpp_info.components["qtLinguistTools"].names["cmake_find_package_multi"] = "LinguistTools"
    _create_module("UiPlugin", ["Gui", "Widgets"])
    self.cpp_info.components["qtUiPlugin"].libs = [] # this is a collection of abstract classes, so this is h
    self.cpp_info.components["qtUiPlugin"].libdirs = []
    _create_module("UiTools", ["UiPlugin", "Gui", "Widgets"])
    _create_module("Designer", ["Gui", "UiPlugin", "Widgets", "Xml"])
    _create_module("Help", ["Gui", "Sql", "Widgets"])
```

github.com/conan-io/conan-center-index: Conan recipe for Qt6



Recipes need to
"Duplicate" information that the
library's build system already
knows about

Interoperability

- Libraries don't provide this information in a standard way
- **A lot of the heavy lifting** is done by repository maintainers to “connect” things together
 - Conan Center Index recipes, vcpkg ports, homebrew formulas, Linux distro packages, Conda recipes, your local “infra” team
- This leads to “lock-in” - hard for teams to try out different solutions

Closing remarks

- There's no standard way for libraries to communicate usage requirements to consumers
 - CMake exported targets work really well - for CMake
 - Otherwise developers fall back to a myriad of suboptimal ways to consume dependencies
 - Repository maintainers have been successful at creating isolated ecosystems
- The concept of a library has evolved over the years
 - Handled at different levels that are completely decoupled from each other
 - C++ modules are adding a new level of complexity!
- There have been proposals to fix this
 - But have gained very little traction in years
 - Difficult to get the scope right
- **Let's fix this!**

Thank you

Any questions?

Acknowledgements

Icons made by [Iconixar](https://www.flaticon.com/authors/iconixar) from www.flaticon.com