Libraries:

A First Step Toward Standard C++ Dependency Management

BILL HOFFMAN
& BRET BROWN

Libraries: A First Step Toward Standard C++ Dependency Management

October 3, 2023

Cppcon

The C++ Conference

Bloomberg

Engineering

Bret Brown, C++ Infrastructure Lead, Bloomberg
Bill Hoffman, CTO, Kitware

Hello! Welcome!

Bret Brown

C++ Infrastructure Lead, Bloomberg

mail@bretbrownjr.com

https://twitter.com/bretbrownjr

https://github.com/bretbrownjr

https://gitlab.kitware.com/bbrown105

Bill Hoffman

CTO, Kitware

bill.hoffman@kitware.org

https://github.com/billhoffman

https://gitlab.kitware.com/bill-hoffman

Goals

Non-goals (so far)

- Package archive formats
  - `.deb`, `.rpm`, `.pkg.tar.zst`, `NUPKG`
- Package managers
  - APT, RPM, PKGBUILD, WPM
- Source repository structure
  - `cxxproject.toml`, pitchfork, https://wg21.link/p1204
- Build workflows
  - `./configure && make && make install`

Why not? Scope and adoption friction!

Then what?

- Package archive contents
  - i.e., `dpkg-deb --contents fangorn.deb`
- Metadata file(s) to declare those contents
- Modest tool interop requirements

In short, declaring libraries installed on a filesystem

Goals

- ✓ A ***first step*** towards a robust packaging ecosystem
- ✓ Explicit metadata with a specification
- ✓ All architectures and environments
- ✓ Multiple languages
- ✓ Projects as portable as the code they contain!
- ✓ Projects should be “cattle,” not “pets”!

Why dependency management?

Consensus: Managing dependencies == way too hard

Q: Which of these do you find frustrating about C++ development?

- 82% – Managing the libraries my application depends on
- 80% – Build times
- 71% – Setting up CI pipelines from scratch
- 69% – Concurrency safety
- 68% – Setting up an IDE from scratch
- 66% – Managing CMake projects

Source: https://isocpp.org/files/papers/CppDevSurvey-2023-summary.pdf

Pain: What's a dependency bug?

Missing interfaces:

```
fangorn.cxx:2:10: fatal error: jsonlog/core.hxx:
No such file or directory
```

This could be caused by other root problems!

Pain: What’s a dependency bug?

Missing implementations (symbols):

```
main.cpp:(.text+0x20): undefined reference to
`jsonlog::log(string_view, int)'
```

Pain: What's a dependency bug?

Mismatching dependencies:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555144 in main () at jsonlog.cpp:44
```

Pain: What's a dependency bug?

```
$ cmake --build build/
CMake Error at CMakeLists.txt:5 (find_package):
  By not providing "Findjsonlog.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "jsonlog", but
  CMake did not find one.

  Could not find a package configuration file provided by "jsonlog" with any
  of the following names:
    jsonlogConfig.cmake
    Jsonlog-config.cmake

# ... CMake gives you some dependency management tips here ...
```

Aside: Coloring and bolding added for emphasis

Motivation: What would we *design*?

Much better:

```
fatal error: The `jsonlog' library is not installed.

fatal error: The `jsonlog' library is not listed as a build
dependency.

warning: The `jsonlog' and `fangorn' libraries have
conflicting build configurations.

fatal error: `jsonlog/core.hxx' is not associated with a library
or executable.

Installing jsonlog...
```

Motivation: Why don’t tools just “fix it”?

- ∴ Existing tools have limited context
  - Compilers
  - Linkers
  - Build systems †
- Fuzzy responsibilities ⇒ complex tools, projects
- Coping strategies are non-portable!

† Caveat: Well-governed build configurations with extra assumptions help. But not portably.

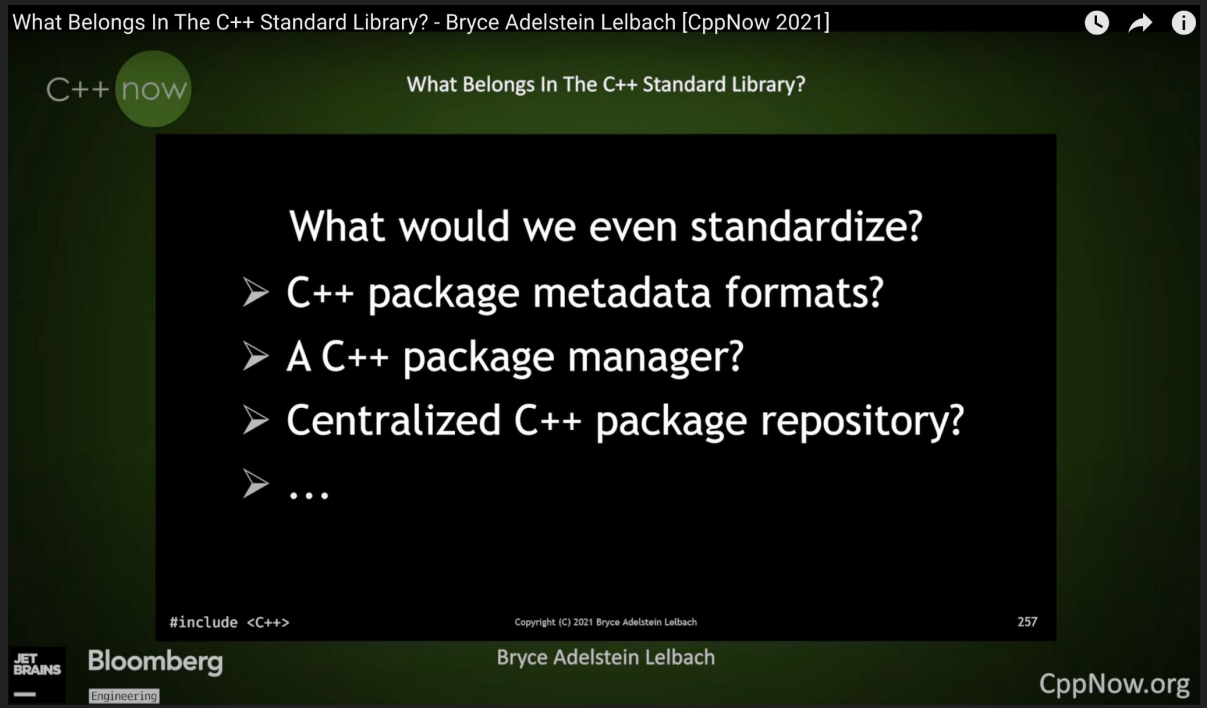
Consequence: Build system stagnation

- Most build systems sort of manage dependencies?
- Build systems become too complex to disrupt
- Even basic projects need sophisticated build systems
- Network effects > actual features

Consequence: Underdeveloped Library Ecosystem

- Quick: Name the best way to format logs in JSON
- Why not more “use this library” talks at CppCon?
- Pressure on the standard library

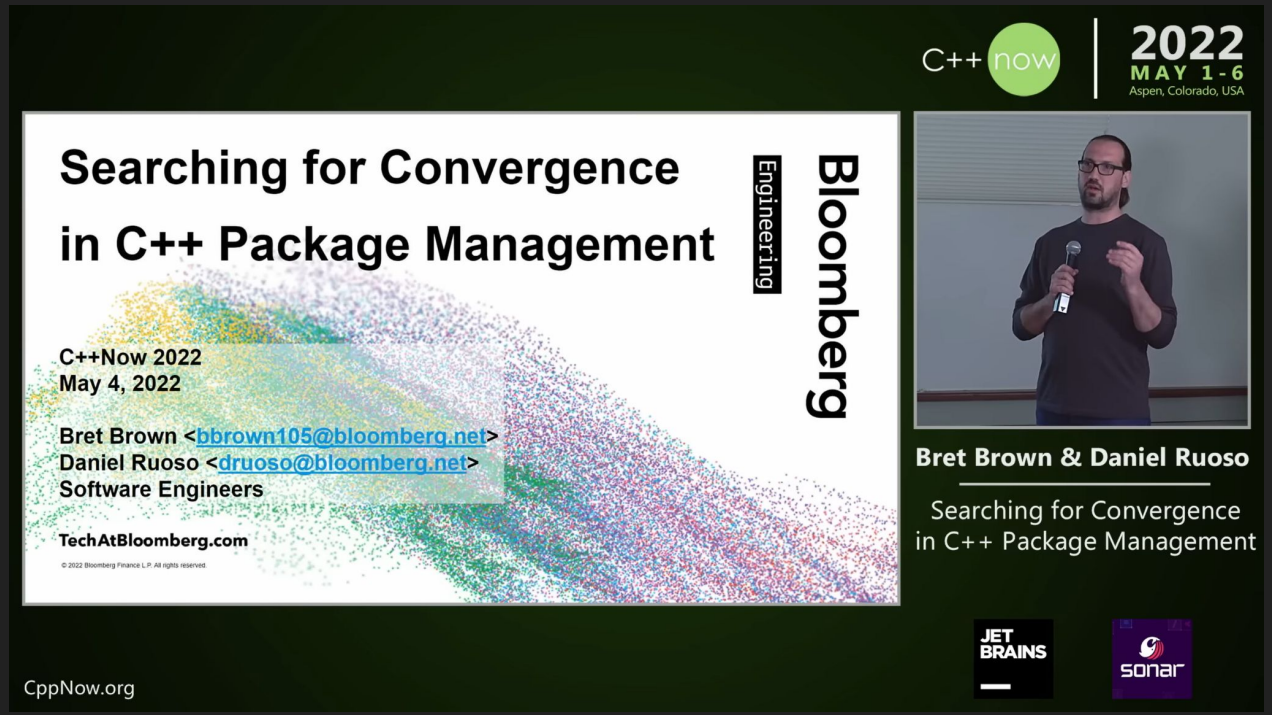
See also: Pressure on the standard library



What Belongs In The C++ Standard Library?
Bryce Adelstein Lelbach - C++Now 2021

https://www.youtube.com/watch?v=OgM0MYb4DqE

See also: Case for packaging standards



Searching for Convergence in C++ Package…
Bret Brown & Daniel Ruoso - C++Now 2022

https://youtu.be/VCrLAmJWZFQ

Verdict: *Somebody*
needs to work on this!!

Hey… we’re somebody!

The Plan

Disclaimer: Not at 1.0 yet!

- Might need to pivot
- Subject to discussion and changes
- Please help us get there!

Early adoption

- Bloomberg needs next-gen packaging metadata
  - Upgrading from `pkg-config` for project ⇔ project interop
  - Don’t need or want a unique solution!
- About Bloomberg’s C++
  - More than 30K C++ projects, Debian-based packaging
  - Over 500 million lines of code
  - Same open source libs you use: `zlib`, `fmt`, `protobuf`, `boost`, etc.
    - …using the build systems they support

More: `pkg-config` limitations

- P2800
- Ben Boeckel
  - Kitware
  - ISO C++ Tooling Study Group
- 2023-09-20

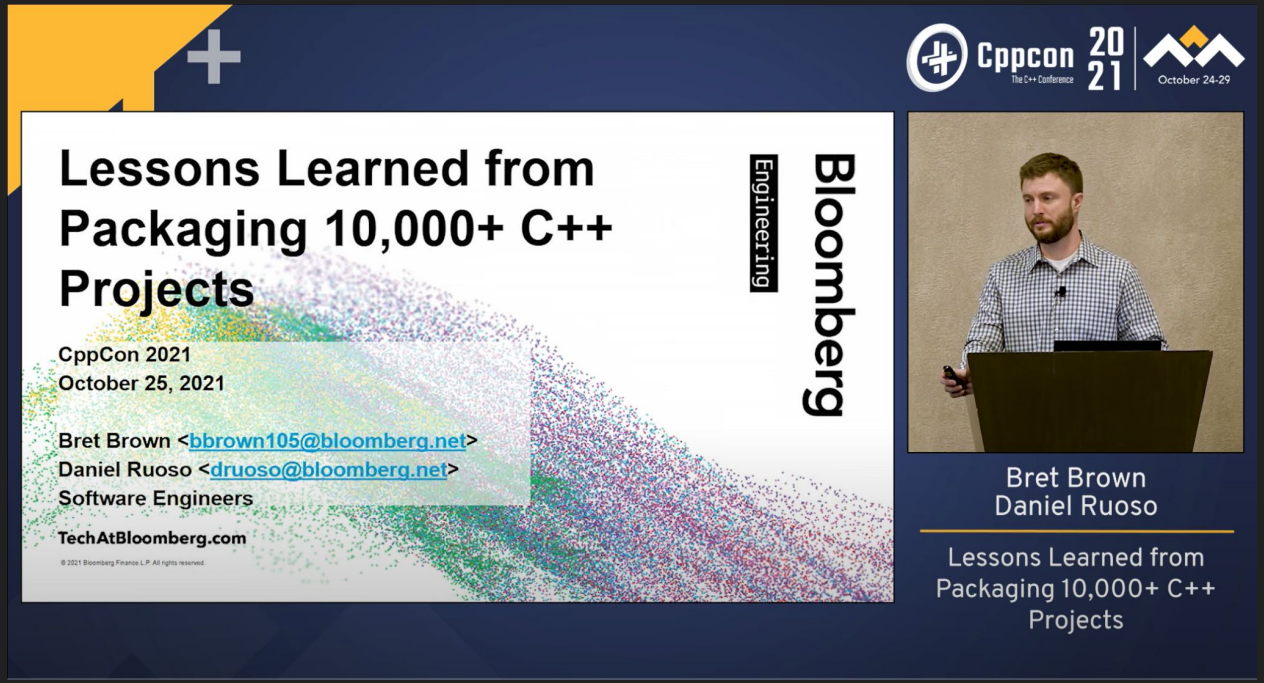
Dependency flag soup needs some fiber

Ben Boeckel
<ben.boeckel@kitware.com>
version P2800R0, 2023-09-20

Table of Contents

https://mathstuf.fedorapeople.org/flag-soup/flag-soup.html

More: Bloomberg on Packaging



"Lessons Learned from Packaging 10,000+ C++ Projects" @ CppCon 2021
Bret Brown, Daniel Ruoso
https://www.youtube.com/watch?v=R1E1tmeqxBY

Spec: Common Packaging Specification (CPS)

- Proposed by Kitware's Matthew Woehlke
- Presented to ISO C++ Tooling Study Group
  - https://wg21.link/p1313

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1313r0.html

Let's Talk About Package Specification

Document: P1313R0
Date: 2018-10-07
Project: ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience: Study Group 15 (Tooling)
Author: Matthew Woehlke (mwoehlke.floss@gmail.com)

Abstract

This paper explores the concept of a package specification — an important aspect of interaction between distinct software components — and recommends a possible direction for improvements in this area.

Contents

Momentum: Postmodern (?) CMake

- CMake support ⇒ trivial upgrades
- Replace CMake modules with CPS JSON
- Trivially adoptable for current CMake users
  - Ideally via normal CMake version upgrades
  - Use existing `find_package(...)` calls
  - Use existing install(EXPORT …) calls
  - Should work with existing packaging approaches
    - Conan 2.0, vcpkg, Debian, etc.

Upside: Easier CMake interop

- Build system interop and freedom!
- Generate via templated `foobar.cps.in` files and find/replace
  - *Very* common to see `foobar.pc.in` files now!
- JSON interop commoditized at this point
  - Even `jq` calls from CLI could work in some cases!

Now. We’re ready.

C++ modules need dependency metadata

- Modules upped the urgency
- Awkward designs for shipping modules
- ✖ Choosing between dependencies & build systems
  - {fmt} is only modular for CMake users & monorepos?

Problem: Dependency management local maxima

- Local solutions have been effective enough
- Large organizations have figured out bespoke approaches
- Small organizations seem to muscle through
- Starting over sounds expensive

Solution: Optimize for adoption velocity

https://youtu.be/VMYVbA2gg0g

Solution: Features for monorepos

Why should monorepo and git submodule users care?

- Standard ways to work at boundaries of build system
- Tools not in monorepo build
  - Compiler explorer
- Host machines have prebuilt dependencies installed!
- Convergence & consistency help in all workflows

Problem: Low Expectations

- Not using existing package managers
- Using unpackaged dependencies
- Header-only libraries

Solution: Raised expectations

- Not using existing package managers
  - Package manager adoption growing
  - Standards should accelerate adoption
- Using unpackaged dependencies
  - Problematic for security detection and patching
  - But we expect our python to be in PYPI?!
- Header-only libraries
  - Unclear these will survive a C++ modules transition
  - Cannot declare dependencies anyway!

CPS and software bills of materials (SBOM)

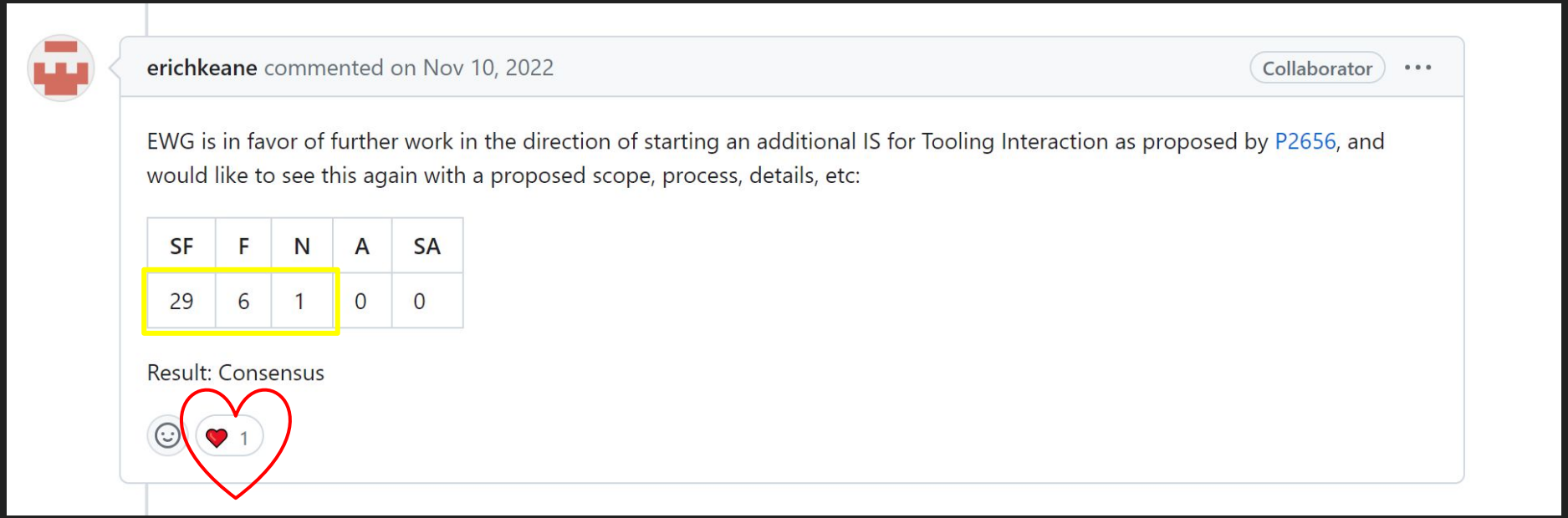
- SBOM is a hot topic
  - Ensuring software transparency
  - Managing open-source software and third-party dependencies
  - Identifying and mitigating security vulnerabilities
  - Complying with legal and regulatory requirements
- CPS would enable easier SBOM creation

Problem: Not in scope for ISO C++ standard

- Lots of people don’t understand that
- I didn’t!

C++ Standards Committee - Fireside Chat Panel - CppCon 2021 https://youtu.be/HQ7_jbN0Svc

Solution: C++ Ecosystem International Standard



erichkeane commented on Nov 10, 2022 — Collaborator

EWG is in favor of further work in the direction of starting an additional IS for Tooling Interaction as proposed by P2656, and would like to see this again with a proposed scope, process, details, etc:

SF	F	N	A	SA
29	6	1	0	0

Result: Consensus

1

Big thanks: René Ferdinand Rivera Morell

Concepts

Now: What is a library?

- Any combination of:
  - A set of header files
  - A set of cpp files
  - A binary static archive
  - A binary dynamic library
  - Module interface units
  - Module implementation units
  - Compilation rules
  - Linker arguments
- Real use case: A “metalibrary”
  - Absolutely nothing but dependencies on other libraries

```
lib/libspdlog.1.12.0.dylib
lib/libspdlog.a
lib/libspdlog.1.12.dylib
lib/libspdlog.dylib
```

```
include/spdlog/
├── async.h
├──
async_logger-inl.h
├──
async_logger.h
...
```

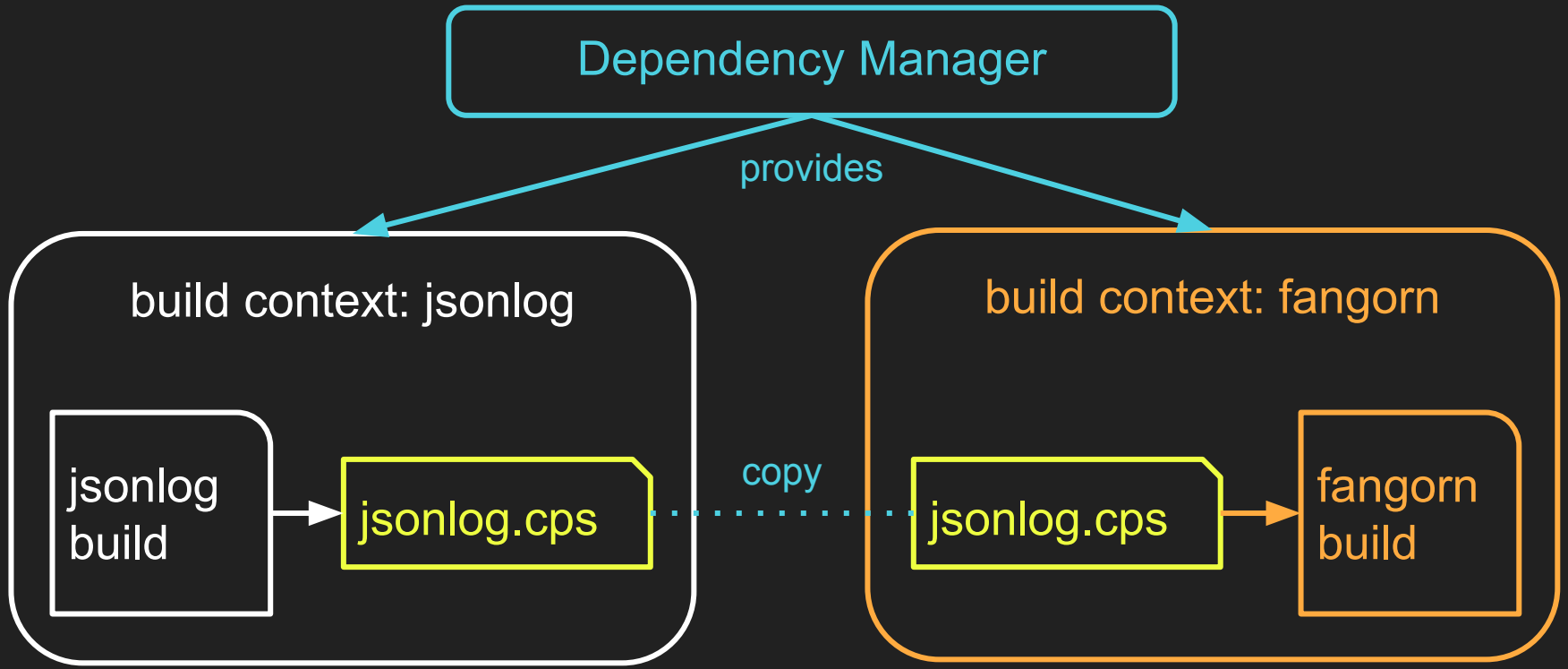
CPS: What is a library?

- See the contents of the `*.cps` file

Dependency management is implicitly required

- The C++ standard just assumes coherent dependencies
  - One Definition Rule (ODR)
  - Ill-formed Program; No Diagnostic Required (if;ndr)
    - Concession that toolchains can’t really help you!
- All symbols (objects) linked together have to have consistent (enough) understanding of one another
- No specific ways to detect incoherency
- No specific correct way to deal with redundant symbols

Build system interop



Consistent naming required

We will benefit from

- Identifier standardization
  - For the same reasons we restrict C and C++ names
- Name collision prevention
  - i.e., names like job, build, util, etc.
  - Name registry? Namespaces?
- Consistent name spellings across the ecosystem
  - ✖ Different names for different package managers!?
  - ✓ Just spell it `SSL` everywhere!

And you may ask yourself,
"Well, how did we get here?"

autotools

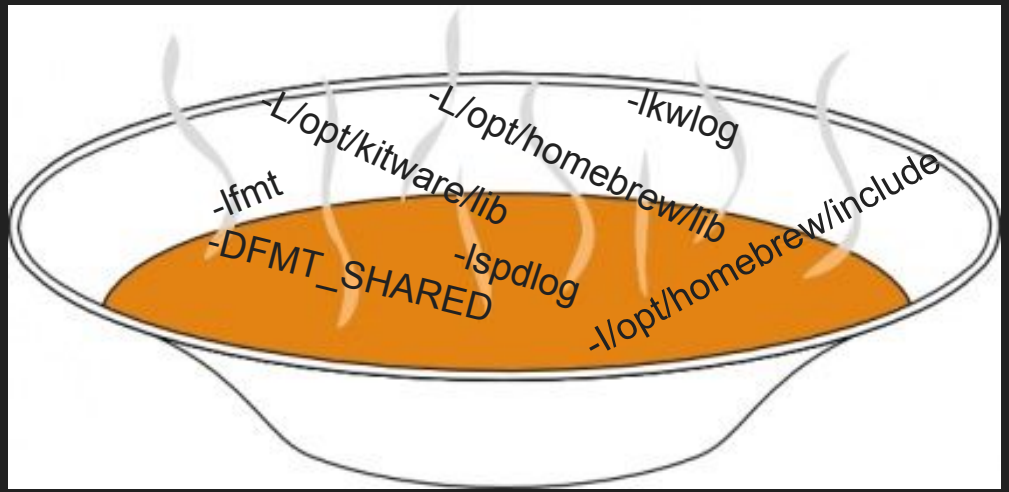
```
AC_CHECK_HEADER([zlib.h])
# Check for libraries
AC_SEARCH_LIBS([gzopen],[z],,[AC_MSG_ERROR([libz not
found, please install zlib (http://www.zlib.net/)])])
```

Issues with this:

- the zlib.h found might not go with the libz you found
- should just find "the library" libz which should include zlib.h

CMake: An evolution – from flag soup to import/export target

- Original CMake did not have transitive linking, mimic autotools
- Big innovation to add transitive linking to cmake
- What about all the flags that go with it `-I`, `-D`, etc?
- After many years modern CMake was created



The target and its "usage requirements" are at the heart of Modern CMake

```
add_library(MathFunctions MathFunctions.cxx)
target_compile_definitions(MathFunctions PRIVATE "USE_MYMATH")
target_compile_definitions(MathFunctions PUBLIC "MYMATH_REQUIRED_FLAG")
add_library(SqrtLibrary STATIC mysqrt.cxx)
target_link_libraries(MathFunctions PUBLIC SqrtLibrary)

add_executable(Tutorial tutorial.cxx)
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

```
% ninja -v
...
[3/6] c++  -DMYMATH_REQUIRED_FLAG  ... -c '/Users/hoffman/Work/My
Builds/cmake/Help/guide/tutorial/Step3/tutorial.cxx'
...
[6/6] : && /c++ ... CMakeFiles/Tutorial.dir/tutorial.cxx.o -o Tutorial
MathFunctions/libMathFunctions.a  MathFunctions/libSqrtLibrary.a && :
```

Library is treated the same: built-in source tree or prebuilt

```
add_library(bar)  # part of the build
find_package(bar REQUIRED) # external to the build
add_executable(foo)
target_link_libraries(foo bar) # makes no difference which bar
# bar could be in this build or could be from find_package
```

cmake/Modules/Find*.cmake

```
$ ls cmake/Modules/Find*
FindALSA.cmake              FindOpenAL.cmake
...
FindODBC.cmake              FindwxWidgets.cmake
FindOpenACC.cmake           FindwxWindows.cmake

$ ls cmake/Modules/Find* | wc -l
179
```

Lessons learned from CMake Modules / `Find*.cmake`

- Original `Find*.cmake` files looked for includes and libraries
  - Find some `.h` files, find some `.a`, `.so`, `.lib` and put them into some VARS much like the autotools example
  - Basically flag soup
  - Headers found might not go with libraries found
  - Libraries might be found in multiple `-L` locations
  - `Find*` should not ship with CMake; ship it with the thing being found!

Evolution of FindQT

cmake/Modules/FindQt3.cmake

FindQt3

Locate Qt include paths and libraries

This module defines:

```
QT_INCLUDE_DIR    - where to find qt.h, etc.
QT_LIBRARIES      - the libraries to link against to use Qt.
QT_DEFINITIONS    - definitions to use when
                    compiling code that uses Qt.
QT_FOUND          - If false, don't try to use Qt.
QT_VERSION_STRING - the version of Qt found
```

If you need the multithreaded version of Qt, set QT_MT_REQUIRED to TRUE

Also defined, but not for general use are:

```
QT_MOC_EXECUTABLE, where to find the moc tool.
QT_UIC_EXECUTABLE, where to find the uic tool.
QT_QT_LIBRARY, where to find the Qt library.
QT_QTMAIN_LIBRARY, where to find the qtmain
 library. This is only required by Qt3 on Windows.
```

cmake/Modules/FindQt4.cmake

FindQt4

Finding and Using Qt4

This module can be used to find Qt4. The most important issue is that the Qt4 qmake is available via the system path. This qmake is then used to detect basically everything else. This module defines a number of `IMPORTED` targets, macros and variables.

Typical usage could be something like:

```
set(CMAKE_AUTOMOC ON)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
find_package(Qt4 4.4.3 REQUIRED QtGui QtXml)
add_executable(myexe main.cpp)
target_link_libraries(myexe Qt4::QtGui Qt4::QtXml)
```

Note: When using `IMPORTED` targets, the qtmain.lib static library is automatically linked on Windows for `WIN32` executables. To disable that globally, set the `QT4_NO_LINK_QTMAIN` variable before finding Qt4. To disable that for a particular executable, set the `QT4_NO_LINK_QTMAIN` target property to `TRUE` on the executable.

Qt Build Tools

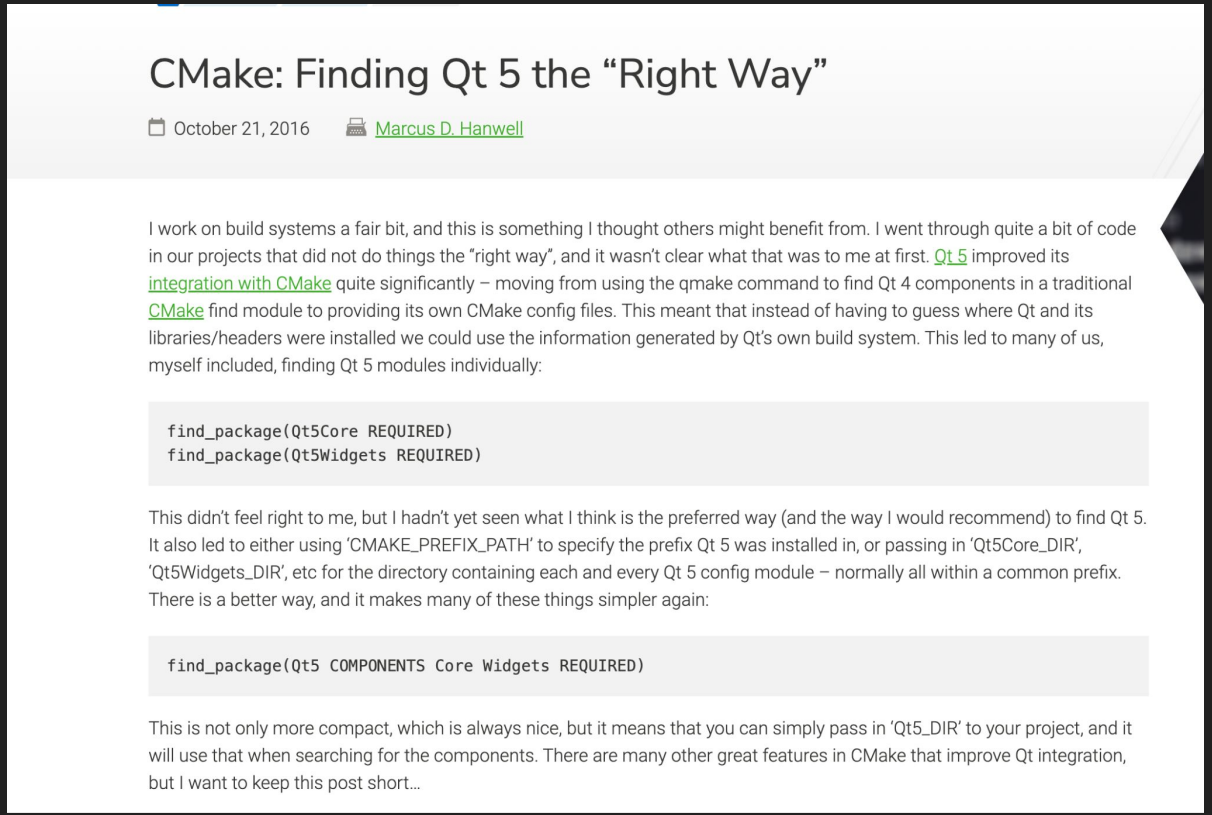
Qt relies on some bundled tools for code generation, such as `moc` for meta-object code generation,``uic`` for widget layout and population, and `rcc` for virtual filesystem content generation. These tools may be automatically invoked by `cmake(1)` if the appropriate conditions are met. See `cmake-qt(7)` for more.

CMake module that ships with CMake that queries qmake and creates imported targets using cmake commands.

FindQt5.cmake

- Does not exist!

Made possible because qmake created .cmake config files with targets to be imported



CMake: Finding Qt 5 the "Right Way"

October 21, 2016 Marcus D. Hanwell

I work on build systems a fair bit, and this is something I thought others might benefit from. I went through quite a bit of code in our projects that did not do things the "right way", and it wasn't clear what that was to me at first. Qt 5 improved its integration with CMake quite significantly – moving from using the qmake command to find Qt 4 components in a traditional CMake find module to providing its own CMake config files. This meant that instead of having to guess where Qt and its libraries/headers were installed we could use the information generated by Qt's own build system. This led to many of us, myself included, finding Qt 5 modules individually:

```
find_package(Qt5Core REQUIRED)
find_package(Qt5Widgets REQUIRED)
```

This didn't feel right to me, but I hadn't yet seen what I think is the preferred way (and the way I would recommend) to find Qt 5. It also led to either using 'CMAKE_PREFIX_PATH' to specify the prefix Qt 5 was installed in, or passing in 'Qt5Core_DIR', 'Qt5Widgets_DIR', etc for the directory containing each and every Qt 5 config module – normally all within a common prefix. There is a better way, and it makes many of these things simpler again:

```
find_package(Qt5 COMPONENTS Core Widgets REQUIRED)
```

This is not only more compact, which is always nice, but it means that you can simply pass in 'Qt5_DIR' to your project, and it will use that when searching for the components. There are many other great features in CMake that improve Qt integration, but I want to keep this post short…

FindQt6.cmake

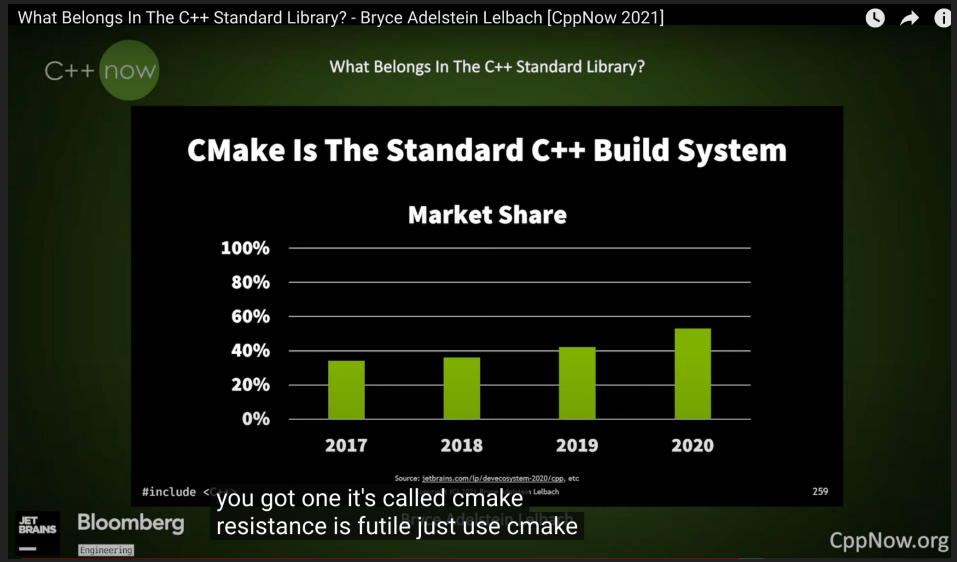
- Does not exist!
- Qt6 builds with CMake and exports targets
- Qt underwent a major build system conversion for this to happen

```
find_package(Qt6 REQUIRED COMPONENTS Core)
```

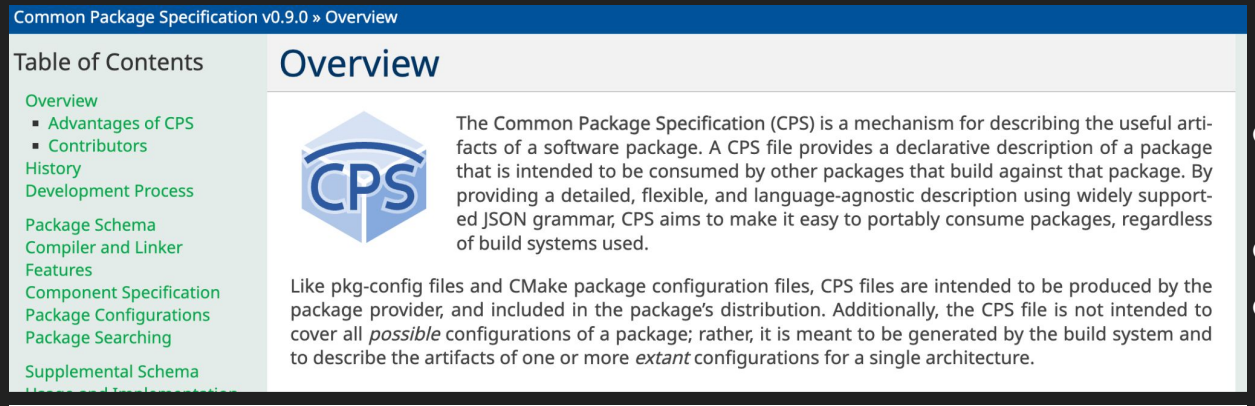
This tells CMake to look up Qt 6, and import the `Core` module. There is no point in continuing if `CMake` cannot locate the module, so we do set the `REQUIRED` flag to let CMake abort in this case.

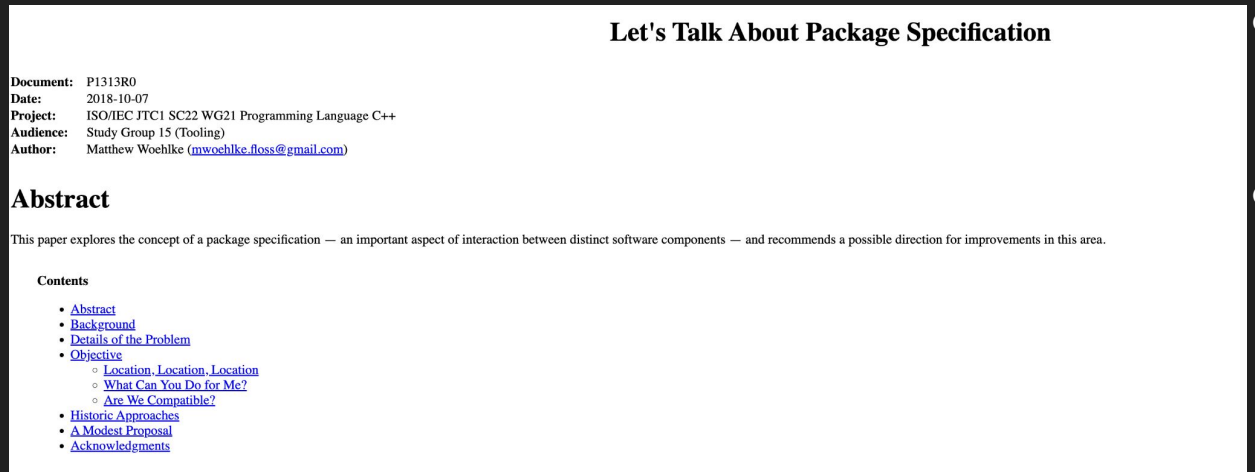
So, everything is perfect, just use CMake!

- Not everything is built with CMake
- CMake is not the only tool that wants to use libraries
- `.cmake` files are hard to read and write if you are not CMake



CPS is the logical next step in this evolution





- Started in 2016 by Matthew Woehlke
- 2018: WG21 paper
- 2022: C++Now talk by Bret and Daniel
- 2022 CppCon hallway track with Conan, vcpkg and others
- 2023: 3 talks at CppCon

CMake and CPS

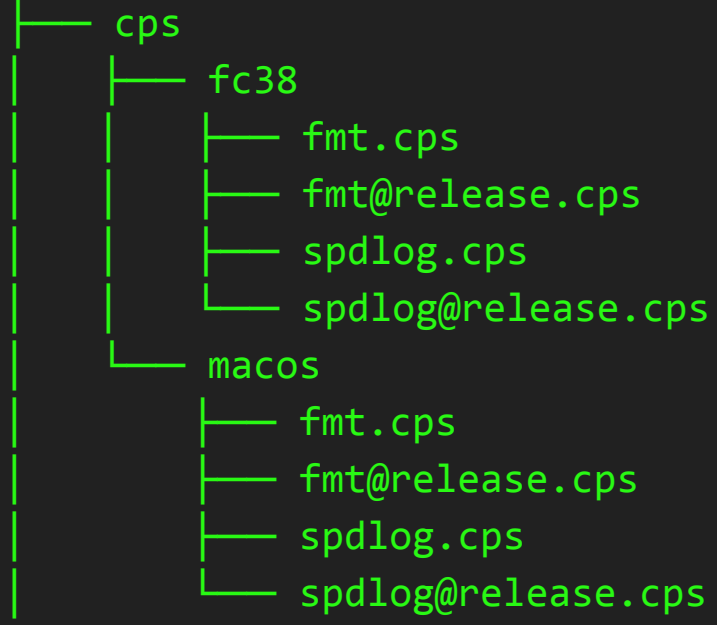
- CMake’s package files are created by CMake and commonly installed alongside libraries on Mac, Linux and Windows
- CMake can be made to create CPS files
- CMake projects will be able to transition to CPS for "free" with an upgrade of CMake

Exported targets part of install fmt and spdlog on a mac

```
./Cellar/fmt/10.1.1/lib/cmake/fmt/fmt-targets-release.cmake
./Cellar/fmt/10.1.1/lib/cmake/fmt/fmt-config.cmake
./Cellar/fmt/10.1.1/lib/cmake/fmt/fmt-config-version.cmake
./Cellar/fmt/10.1.1/lib/cmake/fmt/fmt-targets.cmake
./Cellar/fmt/9.1.0/lib/cmake/fmt/fmt-targets-release.cmake
./Cellar/fmt/9.1.0/lib/cmake/fmt/fmt-config.cmake
./Cellar/fmt/9.1.0/lib/cmake/fmt/fmt-config-version.cmake
./Cellar/fmt/9.1.0/lib/cmake/fmt/fmt-targets.cmake
./Cellar/spdlog/1.12.0/lib/cmake/spdlog/spdlogConfig.cmake
./Cellar/spdlog/1.12.0/lib/cmake/spdlog/spdlogConfigTargets-release.cmake
./Cellar/spdlog/1.12.0/lib/cmake/spdlog/spdlogConfigTargets.cmake
./Cellar/spdlog/1.12.0/lib/cmake/spdlog/spdlogConfigVersion.cmake
```

Simple CPS CMake Demo

Handcrafted `.cps` files that use system installed libraries



CPS files for spdlog for macos

cps/macos/spdlog.cps

```
  "Requires": { "fmt": { "Version": 9 } },
  "Default-Components": [ "spdlog" ],
  "Configurations": [ "release", "debug" ],
  "Components": {
    "spdlog": {
      "Type": "dylib",
      "Requires": [ "fmt:fmt" ],
      "Compile-Features": [ "threads" ],
      "Definitions": [
"SPDLOG_SHARED_LIB","SPDLOG_COMPILED_LIB",
      "SPDLOG_FMT_EXTERNAL" ],
      "Includes": [ "@prefix@/include" ]
    },
    "spdlog-header-only": {
      "Type": "interface",
      "Requires": [ "fmt:fmt" ],
      "Compile-Features": [ "threads" ],
      "Definitions": [ "SPDLOG_FMT_EXTERNAL" ],
      "Includes": [ "@prefix@/include" ]
```

cps/macos/spdlog@release.cps

```
{
  "Cps-Version": "0.8.1",
  "Name": "spdlog",
  "Configuration": "release",
  "Components": {
    "spdlog": {
      "Location":
"@prefix@/lib/libspdlog.1.12.0.dylib",
      "Link-Name":
"@rpath/libspdlog.1.12.dylib"
    }
  }
}
```

Finding `.cps` files

Package Searching

Tools shall locate a package by searching for a file `<name>.cps` in the following paths:

- `<prefix>/cps/` (Windows)
- `<prefix>/cps/<name-like>/` (Windows)
- `<prefix>/<name>.framework/Versions/*/Resources/CPS/` (macOS)
- `<prefix>/<name>.framework/Resources/CPS/` (macOS)
- `<prefix>/<name>.app/Contents/Resources/CPS/` (macOS)
- `<prefix>/<libdir>/cps/<name-like>/`
- `<prefix>/<libdir>/cps/`
- `<prefix>/share/cps/<name-like>/`
- `<prefix>/share/cps/`

Future: CPS targets part of install fmt and spdlog on a Mac

```
./Cellar/fmt/10.1.1/lib/cps/fmt/fmt.cps
./Cellar/fmt/10.1.1/lib/cps/fmt/fmt@release.cps
./Cellar/fmt/9.1.0/lib/cps/fmt/fmt.cmake
./Cellar/fmt/9.1.0/lib/cps/fmt/fmt@release.cps
./Cellar/spdlog/1.12.0/lib/cps/spdlog/spdlog.cps
./Cellar/spdlog/1.12.0/lib/cps/spdlog/spdlog@release.cps
```

Some C++ code that uses spdlog

```
#include <spdlog/spdlog.h>
int main(int, char *[])
{
    spdlog::set_level(spdlog::level::info);
    spdlog::info(
        "Hello, world! This is spdlog version {}.{}.{}!",
        SPDLOG_VER_MAJOR, SPDLOG_VER_MINOR,
        SPDLOG_VER_PATCH);
}
```

Write some CMake that loads the targets

```
# will be a CPS-aware find_package in the future!
if (SPDLOG_USES_EXTERNAL_FMT)
  load_package(${CPS_ROOT}/fmt.cps ${FMT_PREFIX})
endif()
load_package(${CPS_ROOT}/spdlog.cps ${SPDLOG_PREFIX})
```

Now just use the spdlog target to link to our demo

```
add_executable(demo demo.cpp)
# CPS supports "default components". This option controls whether we link to
# the spdlog default component(s) or to a specific component, in order to allow
# us to demonstrate both possibilities.
option(USE_DEFAULT_COMPONENTS "Link to default component(s)." ON)
if (USE_DEFAULT_COMPONENTS)
  target_link_libraries(demo spdlog)
else()
  target_link_libraries(demo spdlog::spdlog)
endif()
```

Configure

```
[0/1] Running CMake to regenerate build system...
-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to:
/Users/hoffman/Work/My Builds/cps/cps-demo/build
```

Build it!

```
$ ninja -v

[1/2] /Library/Developer/CommandLineTools/usr/bin/c++
-DFMT_SHARED -DSPDLOG_COMPILED_LIB -DSPDLOG_FMT_EXTERNAL
-DSPDLOG_SHARED_LIB -isystem /opt/homebrew/include
-std=gnu++17 -arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX13.3.sdk -MD -MT
CMakeFiles/demo.dir/demo.cpp.o -MF CMakeFiles/demo.dir/demo.cpp.o.d -o
CMakeFiles/demo.dir/demo.cpp.o -c '/Users/hoffman/Work/My Builds/cps/cps-demo/demo.cpp'

[2/2] : && /Library/Developer/CommandLineTools/usr/bin/c++ -arch arm64 -isysroot
/Library/Developer/CommandLineTools/SDKs/MacOSX13.3.sdk -Wl,-search_paths_first
-Wl,-headerpad_max_install_names  CMakeFiles/demo.dir/demo.cpp.o -o demo
/opt/homebrew/lib/libspdlog.1.12.0.dylib  /opt/homebrew/lib/libfmt.10.1.0.dylib && :
```

From CPS

Tada!

./demo

[2023-09-19 21:49:35.111] [info] Hello, world! This is spdlog version 1.12.0!

Try it yourself

https://gitlab.kitware.com/matthew-woehlke/cps-demo

https://gitlab.kitware.com/matthew-woehlke/cmake/-/tags/cps-cppcon-demo-2023

CPS and C++20 Modules

```
"foo_component": {
  "Location": "@prefix@/lib/...",
  "Module-Information": "@prefix@/src/foo.mmd"
}
```

Going Forward

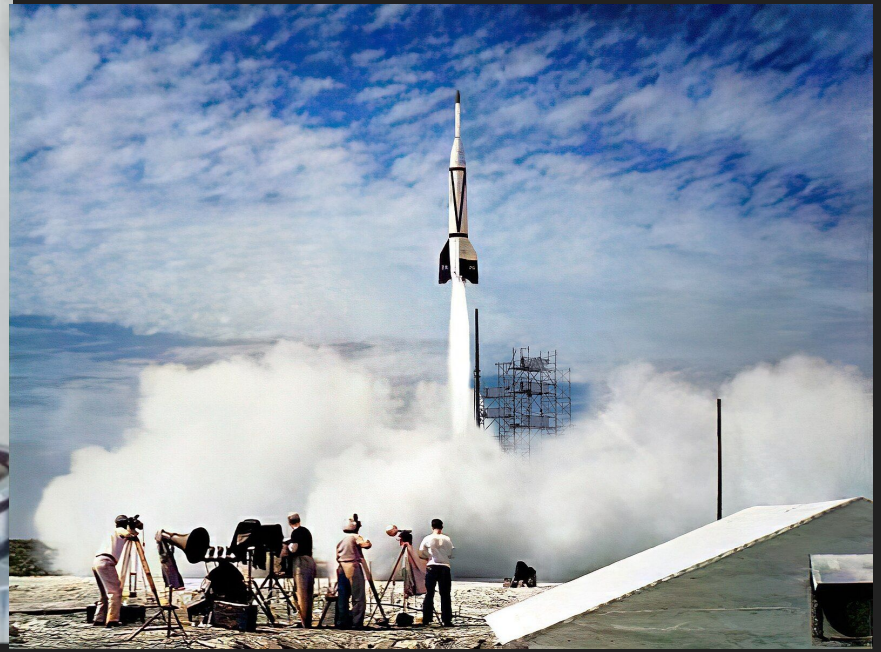
CPS plans

- Gather community feedback
  - Package maintainers: vcpkg, Conan, Conda, Spack, others
  - Build system maintainers
  - Core CMake developers
  - Toolchain and OS vendors
  - You?
- Share and confirm consensus via public ISO C++ Tooling Study Group (SG15) papers

CPS plans

- Continue with MVP in CMake
  - C++20 modules support
  - Implement import export
  - Teach `find_package` to find `.cps` files
- Explore use case inside Bloomberg
  for interop between `.pc` and `.cps`
- Develop a transition plan that
  supports gradual transition from `.cmake` to `.cps`

Not brain surgery or rocket science



CPS plans

- Open source best practices
- Open source project
  - i.e., issues, PRs, public comments, etc.
- ISO C++ review and ratification
  - …eventually targeting an ISO C++ ecosystem standard
  - See: https://github.com/cplusplus/ecosystem-is

Key resources

CPS reference docs: https://cps-org.github.io/cps/

CPS project: https://github.com/cps-org/cps

Demo repos:

- https://gitlab.kitware.com/matthew-woehlke/cps-demo
- https://gitlab.kitware.com/matthew-woehlke/cmake/-/tags/cps-cppcon-demo-2023

Slack: cpplang.slack.com #ecosystem_evolution

Ecosystem and ISO discussions: https://github.com/isocpp/pkg-fmt/

Future Goals

This tech should unlock:

- Better diagnostics for common errors
- Analysis tools to proactively detect mistakes
- Explicit manifests of headers and modules
- Link-what-you-use tooling
- More accurate information for dependency managers
  - Maybe: package repository using scraped metadata
  - Probably: better data for dependency solvers

Thank you!
Questions?

Bloomberg
Engineering

https://www.TechAtBloomberg.com/cplusplus
https://www.kitware.com/cppcon-2023