C++ now

2025

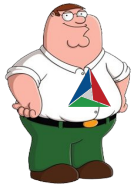
CPS in CMake

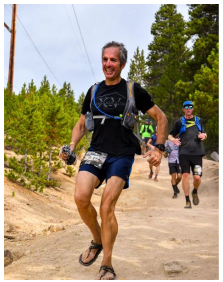
Marching Towards Standard C++ Dependency Management

Bill Hoffman

About Me

- **1990-1999 GE research in a Computer Vision group.**
  - Moved them from Symbolics Lisp Machines to C++ on Solaris and later Linux/Windows
  - Was the build and software library guy (gmake/autotools)
- **1998-Present Founder and CTO Kitware**
  - Started CMake in 1999
  - Mostly management now (pointy haired boss stuff), finding funding for CMake





kitware

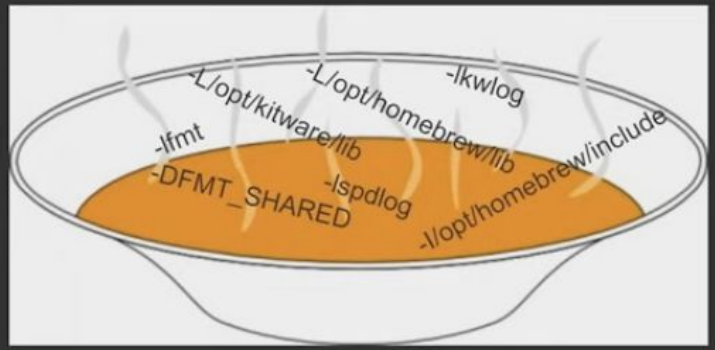
Marching Towards Standard C++ Dependancy Managment

Bret Brown & Bill Hoffman

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

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



What is CPS not doing

- **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!

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!

2024 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	45.43% 571	36.44% 458	18.14% 228	1,257	2.27
Build times	42.86% 537	37.35% 468	19.79% 248	1,253	2.23
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	30.35% 376	42.53% 527	27.12% 336	1,239	2.03
Managing CMake projects	30.38% 377	38.20% 474	31.43% 390	1,241	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	27.67% 347	41.87% 525	30.46% 382	1,254	1.97
Setting up a development environment from scratch (compiler, build system, IDE, ...)	26.27% 330	41.80% 525	31.93% 401	1,256	1.94
[illegible]	[illegible] 234	[illegible] 614	32.00% 399	1,247	1.87
Parallelism support: Using more CPU/GPU/other cores to compute an answer faster	22.94% 286	36.09% 450	40.98% 511	1,247	1.82
Memory safety: Bounds safety issues (read/write beyond the bounds of an object or array)	20.48% 257	35.86% 450	43.67% 548	1,255	1.77
Memory safety: Use-after-delete/free (dangling pointers, iterators, spans, ...)	20.03% 251	34.00% 426	45.97% 576	1,253	1.74
Managing Makefiles	19.88% 235	22.42% 265	57.70% 682	1,182	1.62

Motivation: Why don't tools just "fix it"?

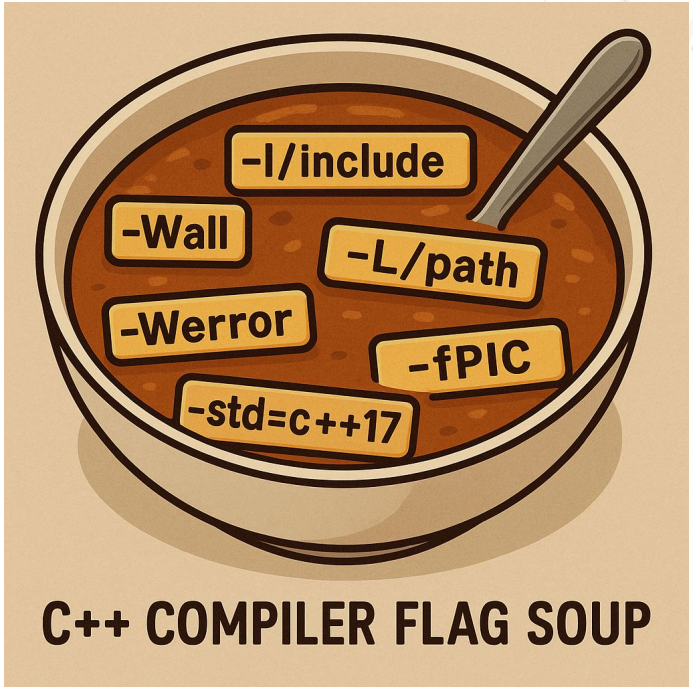
- **∴ Existing tools have limited context**
  - Compilers
  - Linkers
  - Build systems †
- **Fuzzy responsibilities ⇒ complex tools, projects**
- **Coping strategies are non-portable!**

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

The Inadequacy of "Flag Soup" for Modern Dependencies

Current methods of managing dependencies using compiler and linker flags ("flag soup") are becoming increasingly problematic and insufficient, especially with the promise of modules in C++.

- **Lack of Structure:** Relies on unstructured strings that are only fully interpretable by the toolchain.
  - -L/home/bill/foo -lbill -I/home/bill/foo/include
- **Composition Issues:** Flags from different dependencies can conflict or change meaning based on order.
  - -L directores can contain duplicate libraries
- **Ambiguities:** Difficult to express complex requirements and different usage scenarios (static vs. shared, different compiler features).
- **Module Challenges:** Modules require more precise information (e.g., compiler flags affecting BMI compatibility), which "flag soup" struggles to handle.



kitware

More: `pkg-config` limitations

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

Dependency flag soup needs some fiber

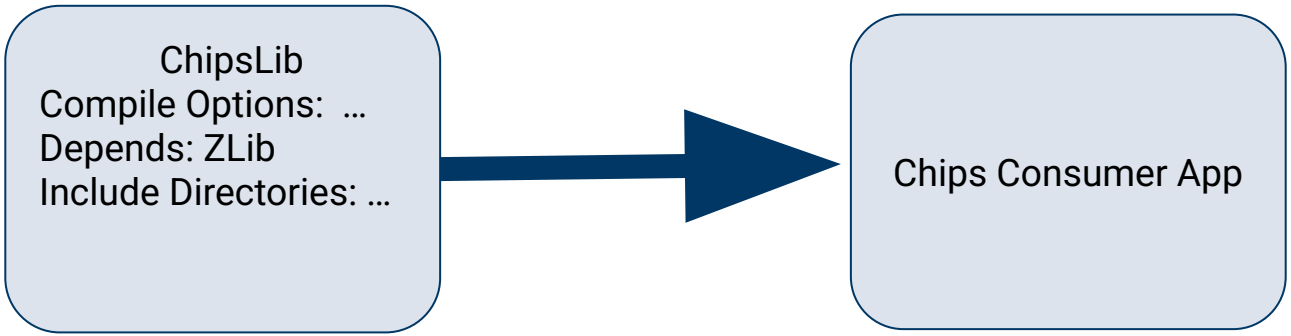
Ben Boeckel
<ben.boeckel@kitware.com>
version P2800R0, 2023-09-20

Table of Contents

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

Moving Beyond "Flag Soup": The Need for Structured "Usage Requirements"

- **"Usage requirements" encompass all the necessary information to meaningfully consume a project (compiler flags, linker flags, environment variables, etc.).**
- **Granular Control: Allows library authors to specify precisely what is needed for compilation, linking, and runtime.**
- **Context Awareness: Can potentially adapt requirements based on the consumer's context (e.g., target type, language, platform).**
- **Foundation for Modules: Provides the necessary framework to manage the more intricate dependencies and build processes required by modules.**



kitware

Common Packaging Specification (CPS)

- Proposed by Kitware's Matthew Woehlke 2018
- 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

kitware

Post-Modern 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!

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?

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**

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
```

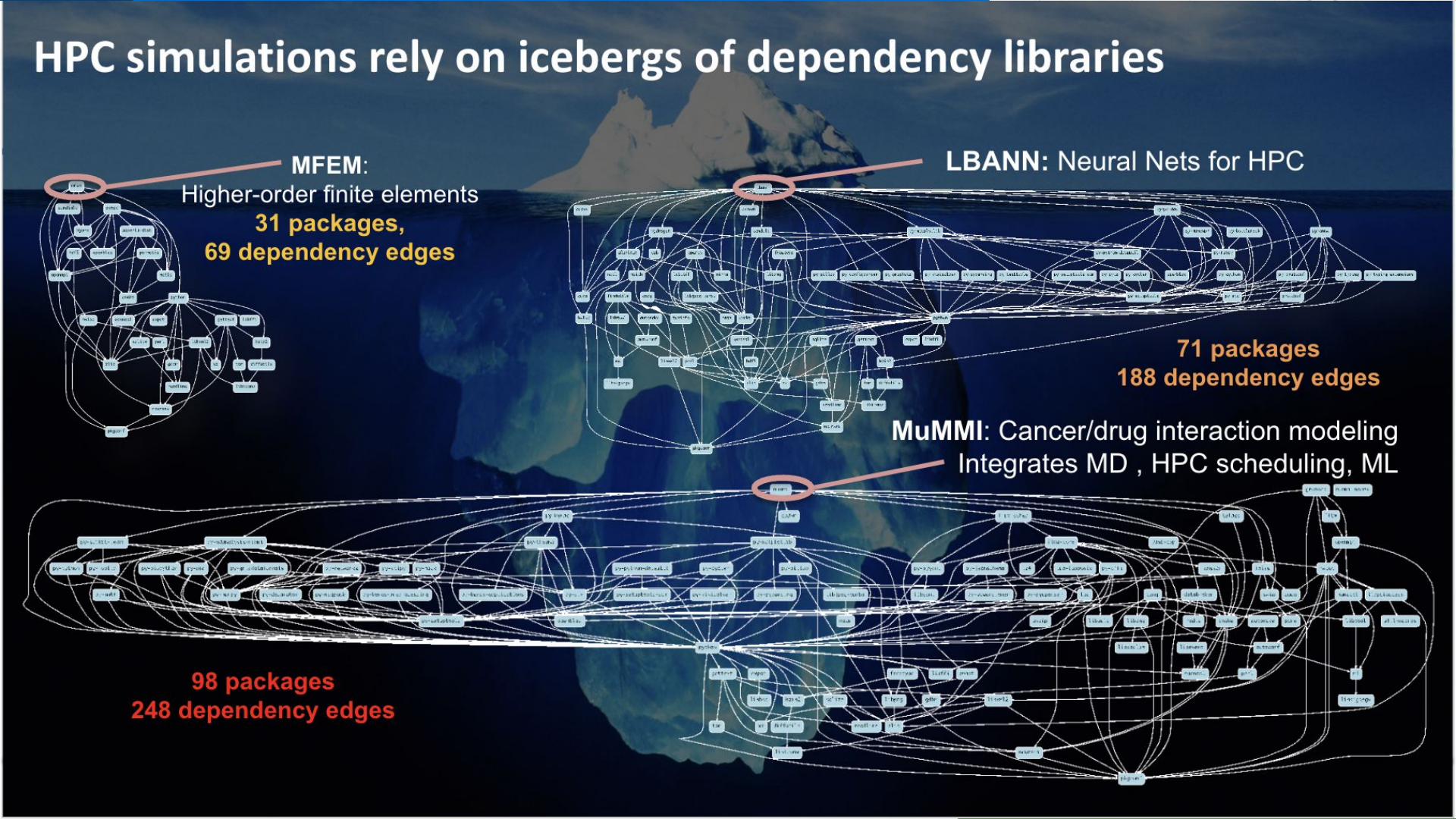
```
include/spdlog/
├── async.h
├──
async_logger-inl.h
├──
async_logger.h
...
```

kitware

CPS: What is a library?

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

HPC simulations rely on icebergs of dependency libraries



The Quest for Standardization: Introducing CPS

- The need for a tool-agnostic, declarative solution for C++ dependencies.
- Introducing the Common Package Specification (CPS) as the effort to achieve this.
- Building upon the progress discussed in recent blogs and community efforts.
- Today's focus: The significant advancement of CPS import in CMake.

Why This Matters

- Improved build reproducibility and reliability.
- Easier integration of third-party libraries.
- Streamlined development workflows and reduced “dependency hell.”
- Fostering a healthier C++ ecosystem.

Key Benefits of Standardization

- Reduced build complexity
- Increased code reuse and modularity
- Improved long-term maintainability of projects
- Stronger ecosystem and community collaboration

Today's Journey

- Looking Back: The Historical Landscape
- Renewed Interest and Initial Export Support
- What is CPS?
- CMake 4.0: The Crucial Step - Importing CPS
- Current Limitations
- The Future and Call to Action
- Q&A

The Wild West of C/C++ Packaging

- Early days characterized by chaos and inconsistent approaches.
- Libraries often had their own unique ways of being consumed (if at all).
- Minimal or no metadata about dependencies.
- “Copy-paste” development and manual configuration were common.

An Early Attempt: pkg-config

- Introduced as a step towards standardization (championed by GNOME).
- Relied on outputting compile and link flags (“flag soup”).
- Strengths: Simple, widely available, and relatively easy to use for basic cases.
- Limitations: Semantic loss, not ideal for complex dependencies, lacks versioning, and difficult to extend.

pkg-config - “Flag Soup”

% pkg-config --cflags --libs glib-2.0

-I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -pthread -lglib-2.0 -lffi -Wl,-Bsymbolic-functions -Wl,-z,relro -Wl,-z,now

-I/usr/include/glib-2.0: Include dir

-I/usr/lib/x86_64-linux-gnu/glib-2.0/include: arch specific include

-pthread: A compiler flag to enable POSIX threads support.

-lglib-2.0: Tells the linker to link against the libglib-2.0.so library (the main GLib library). The -l prefix is followed by the library name without the lib prefix and the .so extension.

-lffi: Tells the linker to link against the libffi.so library (Foreign Function Interface library).
…

As you can see, even for a single library like glib-2.0, the output of pkg-config can contain a variety of include paths, library linking flags, and even linker-specific options. When you start using multiple libraries, these flags can accumulate and become quite complex, illustrating the "compiler flag soup" problem.

CMake’s Evolution: Find Modules to Config.cmake

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.

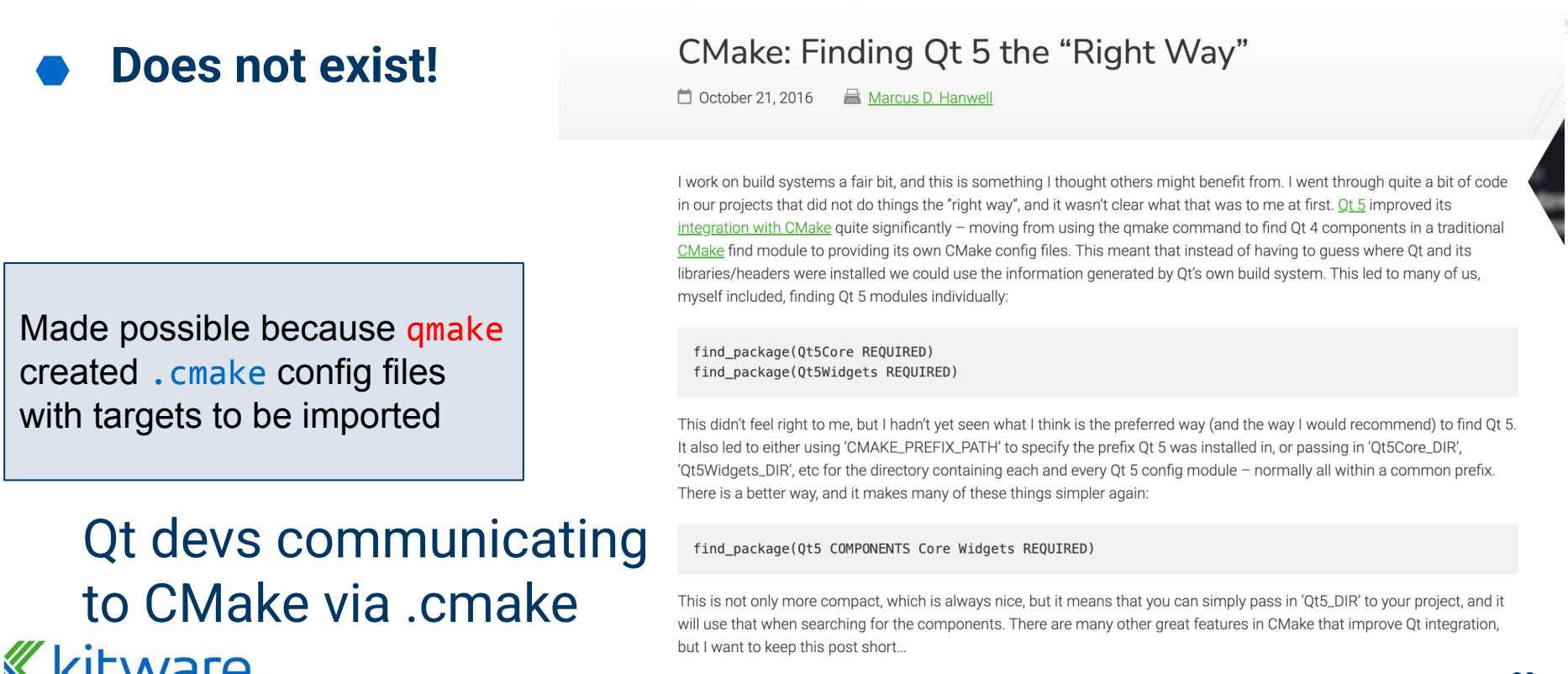
CMake module that ships with CMake that queries qmake and creates imported targets using cmake commands.

cmake/Modules/FindQt5.cmake

- **Does not exist!**

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

Qt devs communicating to CMake via .cmake



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…

kitware

cmake/Modules/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.

CMake’s Evolution: Exported Targets

- A significant improvement: Exported targets.
- Richer information about targets: include directories, link libraries, compile definitions, and more.
- Enables easier consumption of libraries within CMake projects.
- Key Limitation: Tied to CMake script, limiting tool interoperability and cross-platform adoption.

Benefits of CMake Exported Targets

- **Improved dependency management within CMake projects**
  - More precise control over build configurations
  - Encapsulation of build details
  - Support for complex dependency relationships

The Birth of CPS: A Tool-Agnostic Vision

- Motivation: Overcome CMake-specific limitations and the limitations of other build systems.
- Goal: A purely declarative format (JSON) for broad consumption by any build tool.
- Inspired by CMake's exported targets, but designed to be tool-agnostic and language-neutral.
- WG21 paper P1313 had initial lack of momentum, but the idea persisted.

The Vision of CPS

- A universal language for describing C++ packages.
- Enabling seamless integration between different build systems and package managers.
- Promoting a more collaborative and interoperable C++ ecosystem.

A New Hope: Renewed Interest and CMake Export

- **Recent resurgence of interest in C++ dependency management within the C++ community.**
  - Talks and discussions in 2023 highlighted CPS's potential and the need for a standard approach.
- **CMake 3.31 introduced experimental support for *exporting* CPS.**
- **This was the crucial first step towards broader adoption and interoperability.**

CMake 3.31 Experimental Export

- Marks the first time CMake can generate CPS descriptions.
- Allows library developers to describe their packages in a standardized way.
- Sets the stage for other tools to consume this information.

CPS Core Concepts

- A specification for describing consumable software artifacts (primarily C-ABI libraries).
- Tool Agnosticism – designed to be used by any build tool.
- Strong relationship with CMake exported targets.
- Focus on declarative descriptions, not imperative scripts, for better portability.

CPS - Declarative vs. Imperative

- “What” vs. “How”
- CPS describes *what* a package provides, not *how* to build it.
- This allows different build systems to interpret the description.
- Contrast with CMake scripts, which are imperative.

The Power of CPS: Addressing the Shortcomings

- Transitive Dependencies: Improved and more natural handling of complex dependency graphs.
- Improved Version Management: More flexible and expressive versioning schemes beyond basic version numbers.
- Supports Existing CMake Features: Multi-component packages, configurations, interface libraries, and more.

Dependencies Have Dependencies

package Canine depends on Mammal

simple example

find_package(Canine) — finds canine.cps

- internally this will call find_package(Mamal)
- could be MammalConfig.cmake or mamal.cps

Important Caveats (Transitive Dependencies)

- CMake-script dependencies must adhere to CPS naming conventions to ensure correct resolution.
- Risk of partial imports if a transitive dependency is missing, due to CMake's current limitations in handling transactional changes.
- Current behavior makes all transitive dependencies available to the consumer (future visibility restrictions are possible to improve encapsulation).

CPS: Flexible Version Management

- Beyond simple version numbers (e.g., 1.2.3).
- Support for version ranges, compatibility specifications, and semantic versioning.
- Enables tools to select appropriate versions automatically based on project requirements.
- Facilitates smoother upgrades and avoids version conflicts, improving long-term maintainability.

CMake 4.0 - Importing CPS

- CMake 4.0 marks a significant step forward in C++ dependency management.
- Introduces the ability to *import* package information in CPS format.
- Still under development but enables end-to-end experimentation and real-world testing, providing valuable feedback.

Common Package Specification

Old

```
install(
  EXPORT Example-targets
)

install(
  FILES
    Example-config.cmake
    Example-config-version.cmake
)
```

New

```
install(
  PACKAGE_INFO Example
  EXPORT Example-targets
)
```

Same

```
find_package(Example)
```

kitware

Why CPS Import is Crucial

- Completes the workflow: from describing to consuming.
- Allows CMake projects to use libraries described with CPS.
- Enables interoperability between CMake and other build systems.
- Allows package managers to experiment with CPS.
- Opens a two way communication chanel that CMake never had.

Getting Started with CPS Import

```
set(CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES
      "e82e467b-f997-4464-8ace-b00808fff261")
```

(https://gitlab.kitware.com/cmake/cmake/-/blob/v4.0.1/Help/dev/experimental.rst)

Use `find_package()` as usual to locate and import CPS packages.

The eventual goal is to have CPS support enabled by default, simplifying usage for everyone.

kitware

homebrew % find . -name "*.cmake"

```
./Cellar/llvm/20.1.3/lib/cmake/ParallelSTL/ParallelSTLConfigVersion.cmake
./Cellar/llvm/20.1.3/lib/cmake/polly/PollyExports-all.cmake
./Cellar/llvm/20.1.3/lib/cmake/polly/PollyConfig.cmake
./Cellar/llvm/20.1.3/lib/cmake/polly/PollyConfigVersion.cmake
./Cellar/c-ares/1.34.5/lib/cmake/c-ares/c-ares-targets-release.cmake
./Cellar/c-ares/1.34.5/lib/cmake/c-ares/c-ares-config.cmake
./Cellar/c-ares/1.34.5/lib/cmake/c-ares/c-ares-targets.cmake
./Cellar/c-ares/1.34.5/lib/cmake/c-ares/c-ares-config-version.cmake
./Cellar/openssl@3/3.4.1/lib/cmake/OpenSSL/OpenSSLConfigVersion.cmake
./Cellar/openssl@3/3.4.1/lib/cmake/OpenSSL/OpenSSLConfig.cmake
./Cellar/zstd/1.5.7/lib/cmake/zstd/zstdConfigVersion.cmake
./Cellar/zstd/1.5.7/lib/cmake/zstd/zstdTargets.cmake
./Cellar/zstd/1.5.7/lib/cmake/zstd/zstdConfig.cmake
./Cellar/zstd/1.5.7/lib/cmake/zstd/zstdTargets-release.cmake
```

homebrew % find . -name "*.cps" 2027

```
./Cellar/llvm/20.1.3/lib/cmake/polly/Pollly.csp
./Cellar/c-ares/1.34.5/lib/cmake/c-ares/c-ares.csp
./Cellar/openssl@3/3.4.1/lib/cmake/OpenSSL/OpenSSL.csp
./Cellar/zstd/1.5.7/lib/cmake/zstd/zstd.csp
```

kitware

CMake 4.0 - Consuming CPS Imported Packages

- Importing CPS packages creates imported targets within your CMake project.
- Use `target_link_libraries()` to link your executables and libraries against these imported targets.
- Component Naming: CPS uses `<package>:<component>` (becomes `a::b` in CMake's target names).
- Potential for a default interface target with the package name for simpler linking in common cases.

CMake 4.0 - End-to-End Example

- `cmake/zdemo` sample projects on GitHub: https://github.com/cps-org/cps-examples
- Demonstrates CPS export (zwrap) and import with transitive dependencies (ztest) in a realistic scenario.
- Good place to start to understand CPS in CMake

zdemo

```
cmake_minimum_required(VERSION 4.0)
set(
  CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_INFO
  "b80be207-778e-46ba-8080-b23bba22639e"
  )
project(zwrap CXX)
include(GNUInstallDirs)
find_package(zlib-ng REQUIRED)
add_library(zwrap INTERFACE)
target_sources(zwrap PUBLIC FILE_SET HEADERS FILES zwrap.h)
target_compile_features(zwrap INTERFACE cxx_std_17)
target_link_libraries(zwrap INTERFACE zlib-ng::zlib)
install(
  TARGETS zwrap
  EXPORT zwrap
  INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
  FILE_SET HEADERS DESTINATION
  ${CMAKE_INSTALL_INCLUDEDIR}/zwrap
  )
```

```
#pragma once
#include <zlib-ng.h>
#include <string>
namespace zwrap  {
std::string compress(std::string const& in)
{
  auto l = zng_compressBound(in.size() + 1);
  auto out = std::string(l + 1, '\0');
  auto* id = reinterpret_cast<uint8_t
const*>(in.data());
  auto* od =
reinterpret_cast<uint8_t*>(out.data());
  if (zng_compress(od, &l, id, in.size()) ==
Z_OK) {
    out.resize(l);
    return out;
  }
  return {};
}
```

zdemo

```
cmake_minimum_required(VERSION 4.0)
set(
  CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES
  "e82e467b-f997-4464-8ace-b00808fff261"
  )
project(ztest CXX)
include(GNUInstallDirs)
find_package(zwrap REQUIRED)
add_executable(ztest ztest.cpp)
target_link_libraries(ztest PRIVATE zwrap::zwrap)
install(
  TARGETS ztest
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
  )
```

```
int main(int argc, char const* const* argv)
{
  auto const op = std::string{argv[1]};
  if (op == "-c") {
    auto const in = read(std::cin);
    auto const out = zwrap::compress(in);
    putchars(std::cout, in.size());
    putchars(std::cout, out);
  } else if (op == "-d" || op == "-u") {
    auto const in = read(std::cin);
    auto const out = uncompress(in);
    putchars(std::cout, out);
  } else {
    usage();
    return 1;
  }
}
```

Finds zwrap.cps and zlib-ng

zwrap configure, build, install

```
[5/16] Performing configure step for 'zwrap'
…
-- Detecting CXX compile features - done
CMake Warning (dev) at CMakeLists.txt:24 (install):
  CMake's support for exporting package information in the Common Package
  Specification format is experimental.  It is meant only for experimentation
  and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.
…
-- Build files have been written to: /Users/hoffman/Work/cps/cps-examples/cmake/zdemo/b/zwrap
[5/16] Performing build step for 'zwrap'
[7/16] Performing install step for 'zwrap'
[0/1] Install the project...
-- Install configuration: ""
-- Installing: /Users/hoffman/Work/cps/cps-examples/cmake/zdemo/b/install/include/zwrap/zwrap.h
-- Installing: /Users/hoffman/Work/cps/cps-examples/cmake/zdemo/b/install/lib/cps/zwrap/zwrap.cps
```

ztest configure, build, install

```
[13/16] Performing configure step for 'ztest'
...
CMake Warning (dev) at CMakeLists.txt:11 (find_package):
  CMake's support for importing package information in the Common Package
  Specification format (via find_package) is experimental.  It is meant only
  for experimentation and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.
...
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/cps/cps-examples/cmake/zdemo/b/ztest
[13/16] Performing build step for 'ztest'
[2/2] Linking CXX executable ztest
[15/16] Performing install step for 'ztest'
[0/1] Install the project...
-- Install configuration: ""
-- Installing: /Users/hoffman/Work/cps/cps-examples/cmake/zdemo/b/install/bin/ztest
[16/16] Completed 'ztest'
```

zwrap cps file

```
find . -name "*.cps"
./install/lib/cps/zwrap/zwrap.cps
./zwrap/CMakeFiles/Export/11bbab4b5433dca43a5382837be7320f
/zwrap.cps
```

```
zwrap.cps
{
  "components" :
  {
    "zwrap" :
    {
      "compile_features" : [ "c++17" ],
      "includes" : [ "@prefix@/include" ],
      "requires" : [ "zlib-ng:zlib" ],
      "type" : "interface"
    }
  },
  "cps_path" : "@prefix@/lib/cps/zwrap",
  "cps_version" : "0.13.0",
  "name" : "zwrap",
  "requires" :
  {
    "zlib-ng" :
    {
      "components" : [ "zlib" ]
    }
  }
}
```

zlib-ng installation

kitware

zlib-ng found inside ztest build tree

```
% grep zlib-ng CMakeCache.txt
//The directory containing a CMake configuration file for zlib-ng.
zlib-ng_DIR:PATH=/Users/hoffman/Work/cps/install/lib/cmake/zlib-ng
```

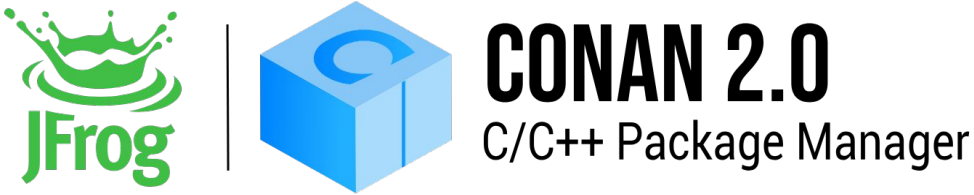
Areas for Improvement

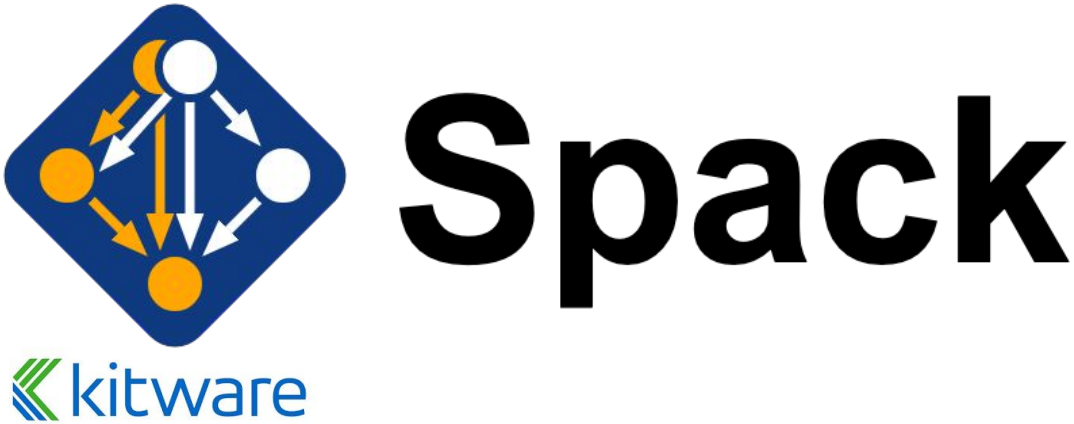
- CPS export is currently only supported for installed packages, limiting development-time usage and iterative development.
- Issues with exporting in both CMake and CPS formats when there are cross-references between exports, requiring careful workarounds.
- Tool-agnostic C++ module support in CPS is not yet implemented, a crucial area for modern C++ and build system integration.
- Import limitations regarding transitive dependencies and the potential for partial imports, which need to be addressed for robustness.

C++ 20 Modules

- C++ modules (no more include what you use)
- Require compiler flag compatibility
- Require consumers to create BMI
- Require the exporting of module interface files

CPS and package managers





Demo1: CMake CPS round-trip (generation + usage)

Slide from: Diego Rodriguez-Losada Gonzalez <diegor@jfrog.com> CppCon 2024

Read CPS

myapp (uses mypkg library)

Generate and consume CPS files in CMakeLists.txt

mypkg

```
|-include
|    mypkg.h
|
|--lib
|    mypkg.lib
|
|--cps
|    mypkg.cps
|    mypkg@release.cps
```

Creates CPS

mypkg (library)

CMake Round Trip CPS

```
cmake_minimum_required(VERSION 3.30)
project(mypkg CXX)

set(CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_INFO
"b80be207-778e-46ba-8080-b23bba22639e")
add_library(mypkg src/mypkg.cpp)
target_sources(mypkg PUBLIC
  FILE_SET HEADERS
  FILES include/mypkg.h
)
target_include_directories(mypkg PUBLIC
            $<INSTALL_INTERFACE:include>)
install(TARGETS mypkg EXPORT mypkg)
install(PACKAGE_INFO mypkg EXPORT mypkg)
```

```
cmake_minimum_required(VERSION 3.30)
project(PackageTest CXX)
set(CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES
e82e467b-f997-4464-8ace-b00808fff261)
find_package(mypkg CONFIG REQUIRED)
add_executable(example src/example.cpp)
target_link_libraries(example mypkg::mypkg)
```

CMake Round Trip CPS

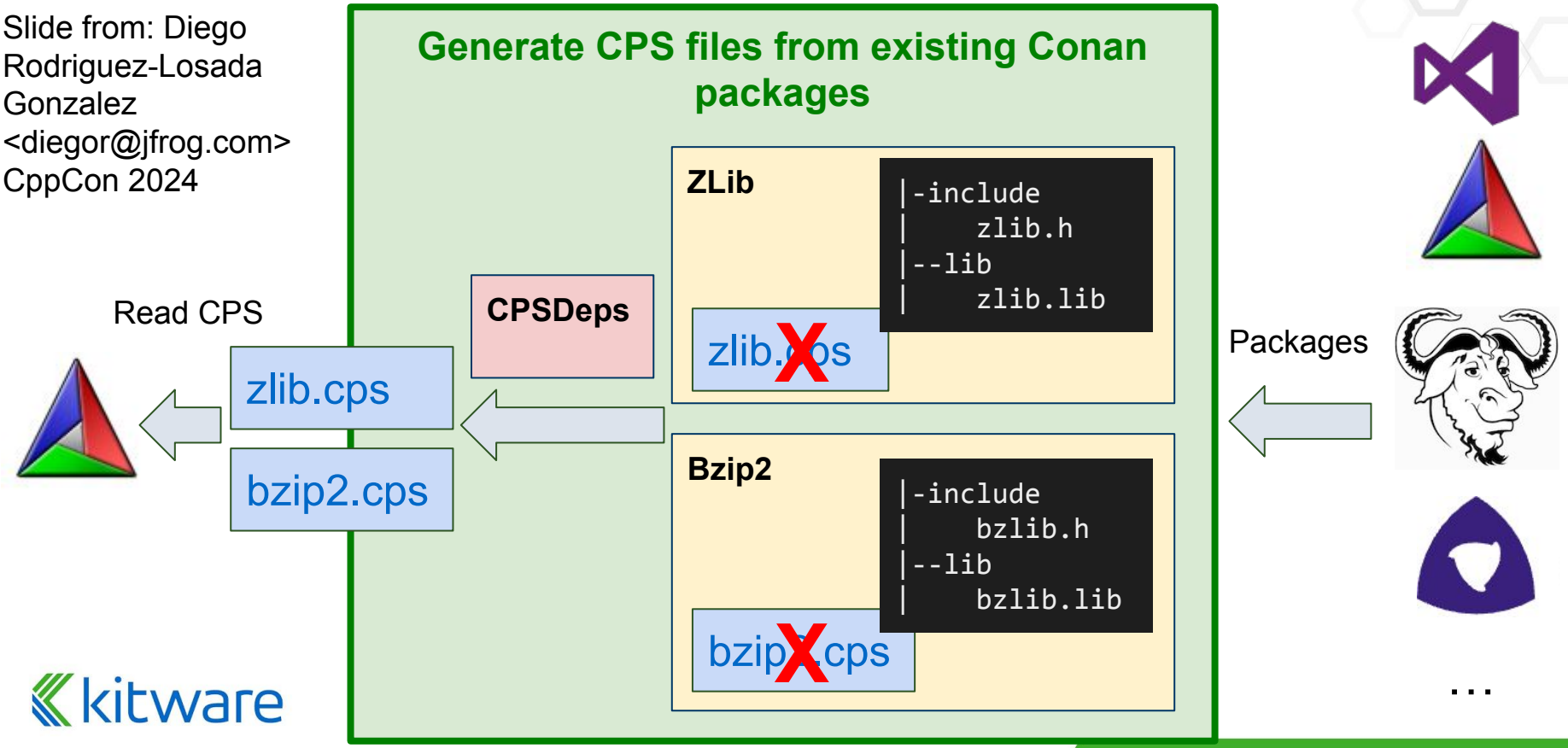
```
CMake Warning (dev) at CMakeLists.txt:19 (install):
  CMake's support for exporting package information in the Common Package
  Specification format is experimental.  It is meant only for experimentation
  and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.
-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/pkg/build
[100%] Built target mypkg
-- Up-to-date: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/pkg/install/lib/libmypkg.a
-- Up-to-date: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/pkg/install/include/mypkg.h
-- Up-to-date: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/pkg/install/lib/cps/mypkg/mypkg.cps
-- Up-to-date: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/pkg/install/lib/cps/mypkg/mypkg@release.cps
CMake Warning (dev) at CMakeLists.txt:6 (find_package):
  CMake's support for importing package information in the Common Package
  Specification format (via find_package) is experimental.  It is meant only
  for experimentation and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/cps/using25/1_cmake_roundtrip/app/build
[ 50%] Linking CXX executable example
[100%] Built target example
```

CMake Round Trip CPS

```
% ./app/build/example
My cool APP!!!!
mypkg/0.1: Hello World Release!
  mypkg/0.1: __aarch64__ defined
  mypkg/0.1: __cplusplus199711
  mypkg/0.1: __GNUC__4
  mypkg/0.1: __GNUC_MINOR__2
  mypkg/0.1: __clang_major__16
  mypkg/0.1: __apple_build_version__16000026
```

Demo2: Generate CPS files

Slide from: Diego Rodriguez-Losada Gonzalez <diegor@jfrog.com>
CppCon 2024



kitware

Package Manager Generated CPS

```
cmake_minimum_required(VERSION 3.30)
project(MyApp CXX)

set(CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES e82e467b-f997-4464-8ace-b00808fff261)

find_package(zlib CONFIG REQUIRED)
find_package(bzip2 CONFIG REQUIRED)

add_executable(example src/example.cpp)
target_link_libraries(example zlib::zlib bzip2::bzip2)
```

Package Manager Generated CPS

```
======== Computing necessary packages ========
Requirements
    bzip2/1.0.8#00b4a4658791c1f06914e087f0e792f5:f6cfbf3d0456dd34d8046d9c3ba95d3489941457#21a1679379b9c5c9e10e01e3dc1780d1 - Cache
    zlib/1.3.1#b8bc2603263cf7eccbd6e17e66b0ed76:25eb974a4372894d34f099bdcf4bc61bbfab8215#00a6d6b7bd108a6358a227c666e94fa2 - Cache

======== Installing packages ========
bzip2/1.0.8: Already installed! (1 of 2)
zlib/1.3.1: Already installed! (2 of 2)
WARN: deprecated: Usage of deprecated Conan 1.X features that will be removed in Conan 2.X:
WARN: deprecated:     'cpp_info.names' used in: zlib/1.3.1, bzip2/1.0.8
WARN: deprecated:     'cpp_info.build_modules' used in: bzip2/1.0.8
WARN: deprecated:     'env_info' used in: bzip2/1.0.8

======== Finalizing install (deploy, generators) ========
conanfile.txt: Writing generators to /Users/hoffman/Work/cps/using25/2_
conanfile.txt: Generator 'CPSDeps' calling 'generate()'
conanfile.txt: [CPSDeps] folder /Users/hoffman/Work/cps/using25/2_consu
conanfile.txt: [CPSDeps]: dep zlib
conanfile.txt: [CPSDeps]: dep bzip2
conanfile.txt: Generating CPS mapping file: cpsmap-clang-20-armv8-gnu17-
conanfile.txt: Generating aggregated env files
conanfile.txt: Generated aggregated env files: ['conanbuild.sh', 'conan
Install finished successfully
CMake Warning (dev) at CMakeLists.txt:7 (find_package):
  CMake's support for importing package information in the Common Package
  Specification format (via find_package) is experimental.  It is meant only
  for experimentation and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/cps/using25/2_consume_conan/build
[100%] Built target example
ZLIB VERSION: 1.3.1
Bzip2 version: 1.0.8, 13-Jul-2019
Bzip2 compressed: BZh11AY&SY-???
```

```
conan install .
cmake . -B build
-DCMAKE_PREFIX_PATH=build/cps/clang-20-armv8-
gnu17-release
cmake --build build --config Release")
```

Package Manager Generated CPS

Spack

```
% git clone https://github.com/johnwparent/spack --branch spack-cps
% source spack/share/spack/setup-env.sh
% spack install zlib bzip2
% spack load zlib bzip2
```

kitware

Package Manager Generated CPS

```
% cmake -GNinja ..
-- The CXX compiler identification is AppleClang 16.0.0.16000026
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Warning (dev) at CMakeLists.txt:7 (find_package):
  CMake's support for importing package information in the Common Package
  Specification format (via find_package) is experimental.  It is meant only
  for experimentation and feedback to CMake developers.
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/cps/using25/2_consume_conan/bspack
```

Package Manager Generated CPS

```
% ninja
[1/2] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
/Users/hoffman/Work/cps/using25/2_consume_conan/src/example.cpp:14:45: warning: conversion from
string literal to 'char *' is deprecated [-Wc++11-compat-deprecated-writable-strings]
   14 |     BZ2_bzBuffToBuffCompress(buffer, &size, "conan-package-manager", 21, 1, 0, 1);
      |                                             ^
1 warning generated.
[2/2] Linking CXX executable example
hoffman@tralfamadore bspack % ./example
ZLIB VERSION: 1.3.1
Bzip2 version: 1.0.8, 13-Jul-2019
Bzip2 compressed: BZh11AY&SY-???
```

kitware

Bloomberg .pc file system

- Bloomberg can generate .pc files from the CMake target model
- Small change allows for generation of cps files
- 20,000 c++ projects, testing CPS at scale

CPS, CMake and SBOM

Software Bill of Materials

A Software Bill of Materials (SBOM) is a formal record containing the details and supply chain relationships of various components used in building software.

```
{
 "spdxId": "...",
 "type": "software_File",
 "software_copyrightText": "...",
 "verifiedUsing": [{
     "type": "Hash",
     "algorithm": "sha256",
     "hashValue": "..."
 }],
 "name": "./src/hello.c",
 "software_primaryPurpose": "source",
 "creationInfo": "_:creationInfo_0"
}
```

SPDX 3 SBOM Snippet for hello.c

Software Bill of Materials

Some things SBOM Tracks:

- Source files
- Binaries
- Build tooling
- Linkage
- Origination (“Where did you download these files from?”)

```
{
 "spdxId" : "...",
 "type" : "Relationship",
 "relationshipType" : "hasDynamicLink",
 "to" : [
   "LinkTargetID",
   "OtherLinkTargetID"
 ],
 "completeness" : "noAssertion",
 "from" : "ThingWithLinkageRequirementsID",
 "creationInfo" : "_:creationInfo_0"
}
```

SPDX 3 SBOM Snippet for Dynamic Link

Software Bill of Materials

CMake Already Tracks This Stuff

- CMake FileAPI added in 3.14
- Originally for IDEs
- Has ~80% of wha[illegible]M needs

Missing Piece:

- Origination
- Need a way to talk [illegible] package managers (we're working on it)


```
{"name": "beman.exemplar",
 "nameOnDisk": "libbeman.exemplar.a",
 "paths": {
   "build": "src/beman/exemplar",
   "source": "src/beman/exemplar"
 },
 "sourceGroups": [
   {"name": "Source Files",
    "sourceIndexes": [0]},
   {"name": "Header Files",
    "sourceIndexes": [1]}
 ],
 "sources": [
   {"compileGroupIndex": 0,
    "path": "src/beman/exemplar/identity.cpp",
    "sourceGroupIndex": 0},
   {"path": "include/beman/exemplar/identity.hpp",
    "sourceGroupIndex": 1}
 ]}
```

CMake File API Snippet for Exemplar

Software Bill of Materials

CMake support is currently an experimental branch:

https://gitlab.kitware.com/daniel.tierney/cmake/-/tree/sbomspdx3

SBOM Generation is part of install, usage is as simple as:

```
install(SBOM FORMAT SPDX)
```

Hoping to land as experimental in CMake 4.1, lots more work to do.

The Future is Bright (and Collaborative)

- CPS implementation is an active and ongoing project with continuous improvements and community contributions.
- Encourage experimentation with both export and import in CMake to gain practical experience and provide feedback.
- User feedback is crucial for shaping the future of CPS and ensuring it meets real-world needs and diverse use cases.

Future Directions of CPS

- Support for other artifact types (executables, etc.)
- Improved version resolution algorithms
- Integration with package managers
- Become the default for CMake

Get Involved! - Contribute to the Evolution

- Report specification issues/discussions on the CPS repository to help refine the standard and clarify ambiguities.
- Report CMake implementation issues on the CMake repository to improve the tools and ensure smooth adoption.
- Engage with the C++ Ecosystem Evolution group (mailing list, Slack channel `#ecosystem_evolution`) for broader discussions and community collaboration.

How to Contribute

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

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

Slack: cpplang.slack.com #ecosystem_evolution

Mailing list:
https://groups.google.com/g/cxx-ecosystem-evolution/about

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

kitware

A Year Closer to Standard C++ Dependency Management

October 22, 2024 Matthew Woehlke and Bill Hoffman

Looking Backward

In the beginning, there was chaos. Lacking any standards or even, in the early days, any conventions, hundreds of C and C++ packages developed their own bespoke approaches for making themselves available to consumers… if they heeded the problem at all.

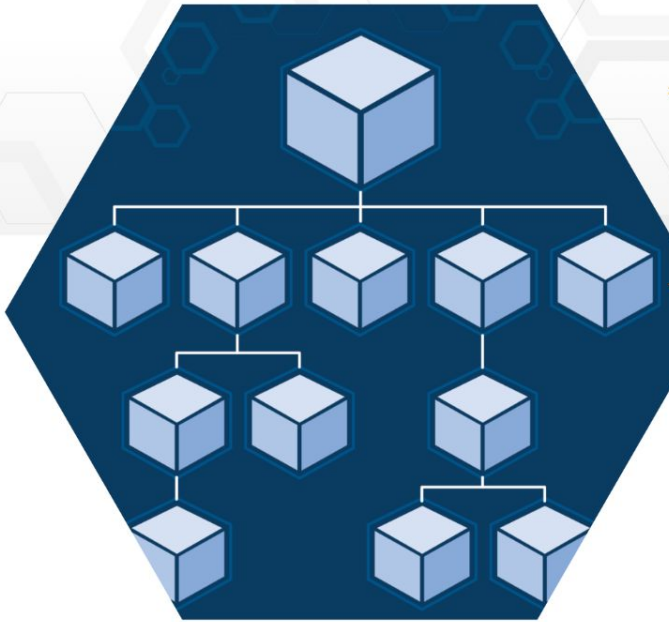
One idea that gained traction was to ship a script or micro-application which, when queried, would spit out the necessary compile and link flags to build against a given library. This approach was heavily championed by GNOME and eventually resulted in the venerable pkg-config. While more-or-less adequate for autotools, flag soup is semantically lossy, and sub-optimal for describing related but separable consumables.

CMake, wading into this anarchy at about the same time, developed in a different direction, initially creating "find modules", and later "exported targets". The latter in particular is a significant advancement in the state of package information exchange

kitware

Navigating CMake Dependencies with CPS

March 31, 2025 Matthew Woehlke



Another Step Toward Standard C++ Dependency Management

Last fall, CMake took the first step toward a new world of expressing software package information in a new format, the Common Package Specification (CPS), which aims to take the expressiveness of CMake's native export format (which is manifested in CMake script) and make it easily available to any tool. If you haven't already, please refer to our previous blog for more background information.

That first step was the ability to export package information in CPS format. While that support remains experimental and incomplete, we believe there is enough substance to enable users to start trying it out… if only the resulting files could be used. Well, with the release of CMake 4.0, we are pleased to announce a major milestone in closing that gap—CMake can now *import* CPS! While this functionality is still under development (some features are not fully implemented), users can

kitware

Summary of CPS Benefits

- Tool Agnosticism
- Transitive Dependency Management
- Flexible Versioning
- Declarative Approach
- Improved Interoperability

CMake has come a long way since 2023 CppCon

- Now
  - export
  - find_package
  - transitive depends
  - conan
  - spack
  - Bloomberg .pc to .cps

```
# 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})

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 all
# 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()
./demo

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

kitware

CMake Docs

find_package

Contents

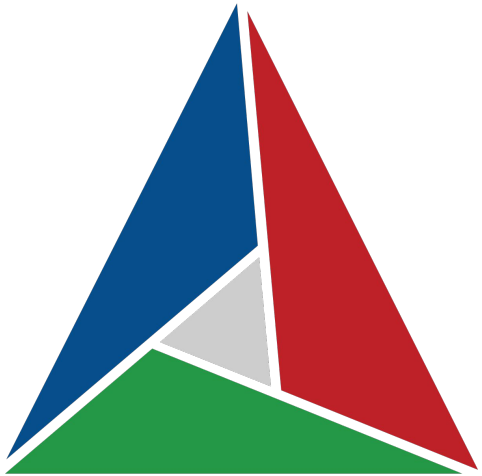
CMake Docs

```
install(PACKAGE_INFO <package-name> EXPORT <export-name>
        [APPENDIX <appendix-name>]
        [DESTINATION <dir>]
        [LOWER_CASE_FILE]
        [VERSION <version>
         [COMPAT_VERSION <version>]
         [VERSION_SCHEMA <string>]]
        [DEFAULT_TARGETS <target>...]
        [DEFAULT_CONFIGURATIONS <config>...]
        [PERMISSIONS <permission>...]
        [CONFIGURATIONS <config>...]
        [COMPONENT <component>]
        [EXCLUDE_FROM_ALL])
```

The `PACKAGE_INFO` form generates and installs a Common Package Specification file which describes installed targets such that they can be consumed by another project. Target installations are associated with the export `<export-name>` using the `EXPORT` option of the **`install(TARGETS)`** signature documented above. Unlike **`install(EXPORT)`**, this information is not expressed in CMake code, and can be consumed by tools other than CMake. When imported into another CMake project, the imported targets will be prefixed with

Thank You & Q&A

bill.hoffman@kitware.com



kitware