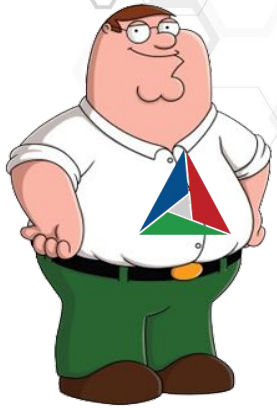
The importance of the build system target model

Bill Hoffman, Kitware

kitware

Bill Hoffman



- One of five founders of Kitware, and the CTO
- Short boring list of jobs
  - GE CRD 1990-1999
  - Kitware 1999-Present
- What am I known for

- CMake guy
- Kitware guy
- Sandal guy

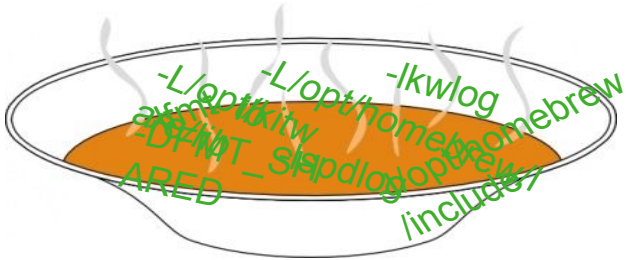
Target model

ChatGPT

In modern CMake, the term "target model" refers to a paradigm shift introduced to CMake in recent versions (3.0 and above). This model emphasizes organizing projects around **targets** rather than directories or files. A target can be an executable, a library, or some other kind of output that your project generates.

Flag Soup

- 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

What are we trying to do with targets?

- Organize large amounts of code
- Reuse code

Simple programs

```
% cxx hello.cpp

% ./a.out

Hello World
```

kitware

Can't have all the code in one file

```
% cxx hello.cpp goodbye.cpp
```

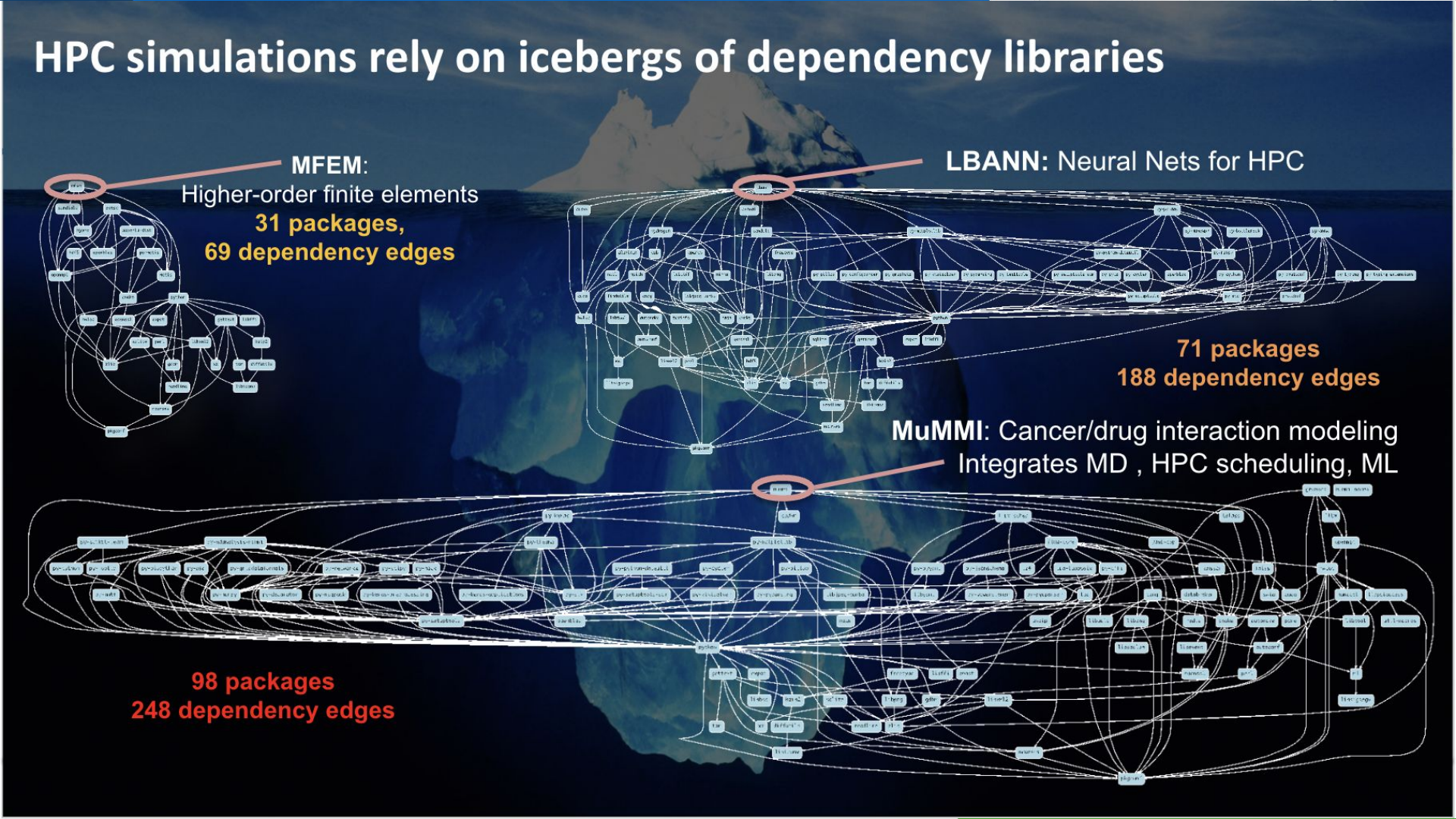
Let's put the code in archives or shared libraries

```
% cxx -c hello.cxx

% cxx -c goodbye.cxx

% ar -c hello.o goodbye.o -o libgreatstuff.a

% cxx -shared hello.o goodbye.o -o libgreatstuff.so

% cxx main.cxx -lgreatstuff
```

But wait there's more

- What about the include files?
- cxx main.cxx -I/path/to/greatstuff_include
- What about -D flags, compiler flags, etc, etc.
- The location of the .a, .lib, or .so is not enough to use it

HPC simulations rely on icebergs of dependency libraries



All abo

Communication: Library maintainers to consumers

- **Library consumers include**
  - other targets in the same build
  - other builds using the same build tool (cmake to cmake)
  - other build tools using the library (autotools to cmake)
  - package management systems
  - IDEs

```
target_link_libraries(helloworld PRIVATE Qt6::Core)
```

Finally, `target_link_libraries` tells CMake that the `helloworld` executable makes use of Qt Core by referencing the `Qt6::Core` target imported by the `find_package()` call above. This will not only add the right arguments to the linker, but also makes sure that the right include directories, compiler definitions are passed to the C++ compiler. The `PRIVATE` keyword is not strictly necessary for an executable target, but it is good practice to specify it. If `helloworld` was a library rather than an executable, then either `PRIVATE` or `PUBLIC` should be specified (`PUBLIC` if the library mentions anything from `Qt6::Core` in its headers, `PRIVATE` otherwise).

kitware

Library usage requirements

Described in the build

```
add_library(foo)
target_link_libraries(foo PUBLIC bar)
```

Used in the project build tree

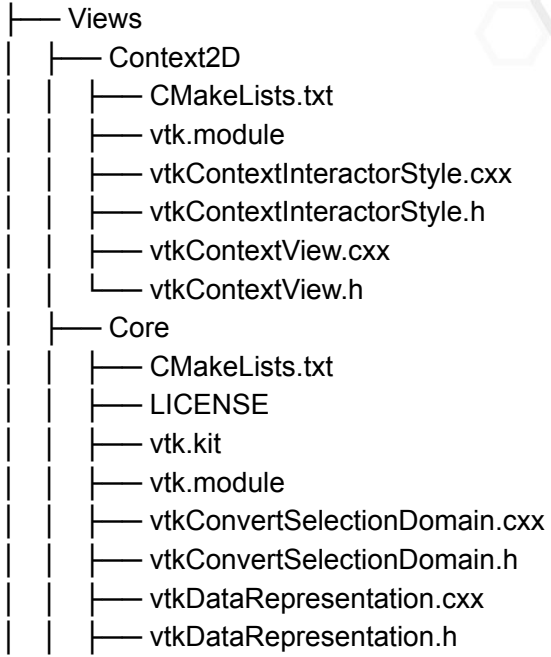
Used from the install tree

kitware

History, how did we get here

Organization of code

- Use the filesystem and store related code in directories
- Each library or executable has all of its code in its own directory



make based builds often use directory structure

- inherit flags as the directory structure is traversed

```
CC = gcc
CFLAGS = -g -Wall
CPPFLAGS = - I /usr/foo/bar # Search for header files in /usr/foo/bar

main.o: main.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c main.c
```

kitware

Target Junior Makefiles

Andrew W. Fitzgibbon and William A. Hoffman

Introduction

The Target Jr makefiles provide

- Objects placed in a separate directory for each combination of architecture, build flags, and compiler.
- O/S and compiler dependencies factored out using *configure*
- Automatic dependency checking.
- Convenience targets: clean, FORCE, echovar-*VAR* etc.
- TargetJr specifics such as documentation generation.

Key limitations are that there is expected to be only one major target (library or program) in a directory, although there is provision for a set of single-file programs (see MINI_SO_SOURCES and MINI_SO_TESTS).

Structure of a *makefile*

The general strategy employed is that a makefile declares the main target it's going to make, say a library:

```
LIBRARY = MultiView
```

Then the things that the target needs:

```
USES = Numerics Basics COOL
```

And the sources and objects that make up that library:

```
SOURCES = file1.cc file2.cc file3.c
```

Finally we include the generic rule-set into the makefile:

```
include ${SYS_IUPACKAGES}/config/top.mk
```

Now we have a makefile which makes a library called "MultiView" (plus whatever other prefices and suffixes are required for particular architectures and builds: `libMultiView.a` for a static unix build, `MULTIVIEW.DLL` for the Gates of Hades), from three source files, and supplied the appropriate include directories etc. to use the TargetJr packages Numerics, Basics and COOL.

Code bases are large and getting larger

```
// look at llvm
% git grep add_library | wc -l
    202
% git grep add_executable | wc -l
     57
 # obviously this wouldn't work:
 % "find . -name "*.cxx" | xargs cxx -c clang
```

kitware

Ways these problems have been solved

Package Managers

common package managers

- **Anaconda**
- **vcpkg**
- **conan**
- **spack**
- **Homebrew**
- **nix**
- **ports**

platform package managers:

- **dpkg**
- **rpm**

What do package managers do?

Know project X requires project Y

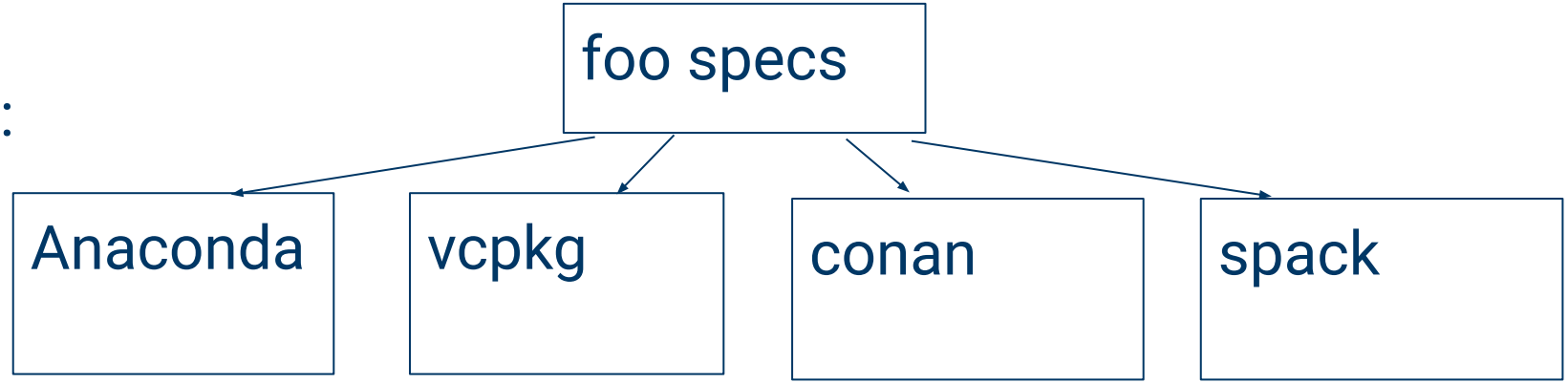
Build Y acceptably for X

Tell X where Y is (handle multiple Y installations)

Package managers are great; but duplication happens

Today:

Better:



kitware

Simple flags via single installation prefix

```
#include "foo.h"

int main(int, char*[])
{
  foo("hello");
  return 0;
}
```

```
c++ main.cxx -lfoo   // -I/usr/include and -L/usr/lib are implicit
```

kitware

autolinking

```
#pragma comment(lib, "foo")
#include "foo.h"
int main(int, char*[])
{
  foo("hello");
  return 0;
}
```

Problems:

where is foo? What about other flags, -I -D?

kitware

Compiler wrappers

17.1.1. Open MPI Wrapper Compilers

mpicc, mpic++, mpicxx, mpifort, mpijavac — Open MPI wrapper compilers

not composable
what if I want mpi and cuda nvcc mpic++?

kitware

Ways of communication (flag soup)

```
c++ `get-foo-flags` main.cxx
```

- What are those flags, what do they mean?
- Are they compatible with the compiler being used
  - -pthread (GNU) versus -Xcompiler -pthread (CUDA)
  - -fPIC (GNU) versus -qpic (IBM XL)
  - -fopenmp (GNU) versus -fiopenmp (non-MSVC frontend of Intel oneAPI) versus -openmp (MSVC)
- No standard way to convey "I am using vendor X compiler"

Project specific config tools

1. **Locate the FreeType 2 `include` directory.**

   You have to add it to your compilation include path.

   In Unix-like environments you can use the `pkg-config` program to retrieve the appropriate compilation flags; say.

   ```
   pkg-config --cflags freetype2
   ```

   to get the compilation flags.

   This program can also be used to check the version of the library that is installed on your system, as well as the required librarian and linker flags.

   Another solution is the `freetype-config` script. However, its use is deprecated since it can't be used reliably for cross compilation.

NAME

libpng16-config - get information about installed libpng library

SYNOPSIS

libpng16-config [*OPTION*] ...

DESCRIPTION

Provides information about libpng library.

Known values for OPTION are:

--prefix
: print libpng prefix

--libdir
: print path to directory containing library

--libs
: print library linking information

--ccopts
: print compiler options

--cppflags
: print pre-processor flags

--cflags
: print preprocessor flags, I_opts, and compiler options

--I_opts
: print "-I" include options

--L_opts
: print linker "-L" flags for dynamic linking

--R_opts
: print dynamic linker "-R" or "-rpath" flags

--ldopts
: print linker options

--ldflags
: print linker flags (ldopts, L_opts, R_opts, and libs)

.pc files

- **pkg-config and .pc files are widely used to provide flags**
- **Popular on Unix-like platforms**
- **Windows but not common enough to count on**
- **Prefer to use -L and -l pairs**
  - can create ambiguity if a library exists in more than one -L path

```
c++ -L/usr/foo -lfoo -L/usr/bar -lbar
```

```
/usr/foo/libfoo.a /usr/foo/libbar.a /usr/bar/libbar.a
```

Build tools solving the problem

autotools finding zlib

```
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/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
```

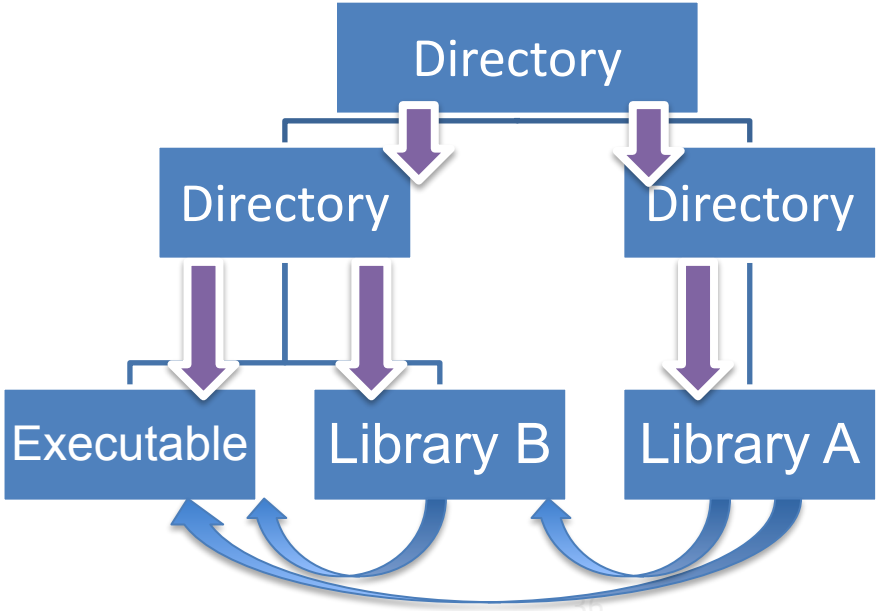
Lessons learned from CMake Modules / `Find*.cmake`

- **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!

Usage Requirements

Before Usage Requirements

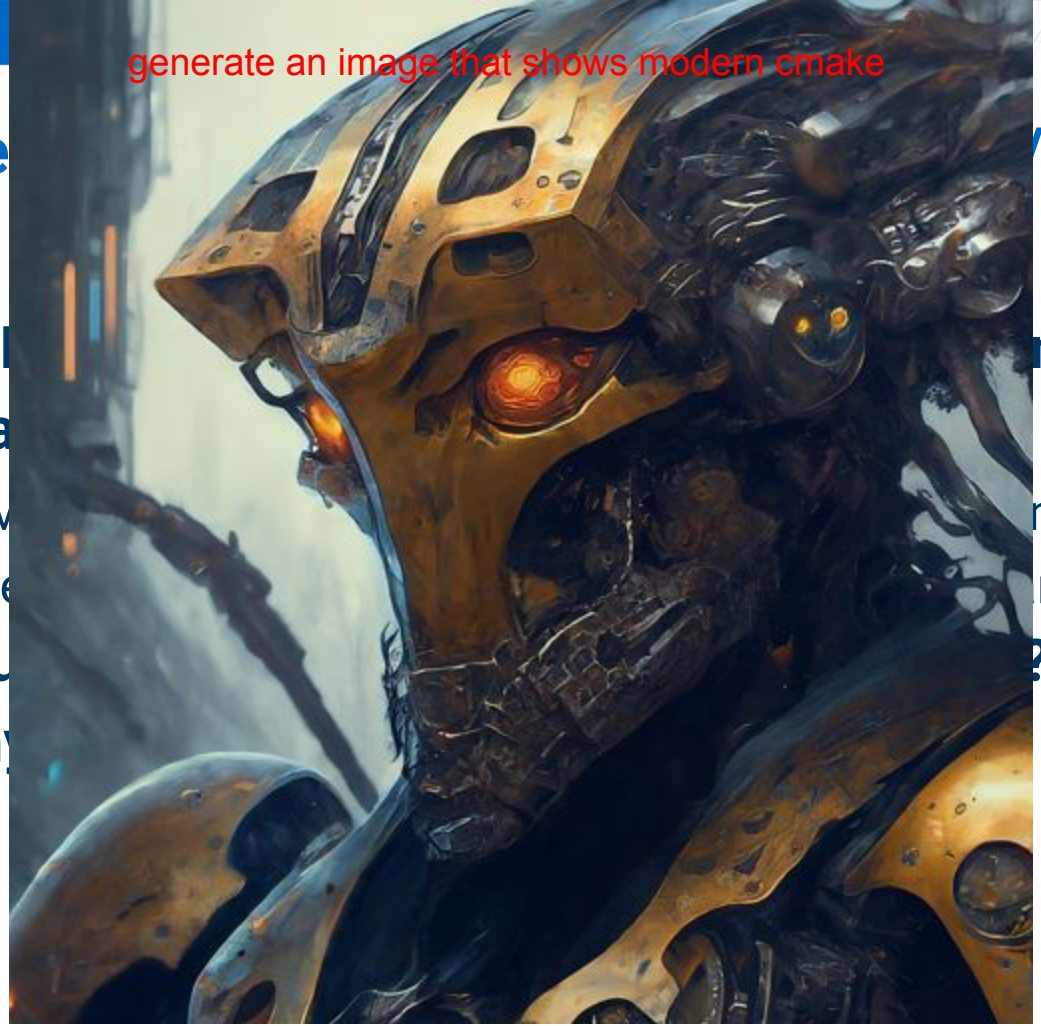
- **Before Usage Requirements existed we used directory scoped commands such as:**
  - `include_directories`
  - `compile_definitions`
  - `compile_options`
- **Consumers have to know:**
  - Does the dependency generate build tree files
  - Does the dependency use any new external package



kitware

CMake: An e /export target

- Original Cl mic autotools
- Big innova
  - transitiv m one another…
    only dire nges
- What abou ?
- After many



kitware

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
```

Evolution of FindQT

kitware

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.

qmake is communicating to CMake from the Qt library maintainers

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…

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.

Some CMake target properties

COMPILE_DEFINITIONS: a flag for the preprocessor (e.g., -Dsome_flag=value)

COMPILE_OPTIONS: a flag to give to the compiler (e.g., -fstack-protector)

COMPILE_FEATURES: a capability of the compiler that has unification logic with other features in the same "category" (e.g., -std=c++17 and -std=c++20 belong to the same "category")

PRECOMPILE_HEADERS: headers to include in the precompiled header (of which most implementations only support one per TU); may also be used to indicate "use this one already-made PCH elsewhere too"

iINCLUDE_DIRECTORIES: paths to search for textual inclusion sources (e.g., collected and passed via -I or -isystem flags)

SOURCES: sources that should be added to the consuming target's source list (e.g., crt1.o)

LINK_OPTIONS: options to pass to the linker (possibly through another frontend) (e.g., --as-needed)

LINK_DIRECTORIES: directories to search for libraries specified by name (e.g., collected and passed via -L flags)

LINK_LIBRARIES: libraries to link to (e.g., /usr/lib/libdl.so or -ldl)

Usage Requirements: INTERFACE

```
target_link_libraries(trunk INTERFACE root)
target_link_libraries(leaf PUBLIC trunk)
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libtrunk.so
             -o libtrunk.so trunk.cxx.o
```

```
/usr/bin/c++ -fPIC -shared -Wl,-soname,libleaf.so
             -o libleaf.so leaf.cxx.o libtrunk.so libroot.so
```

Trunk does *not* get linked to root, but leaf does

INTERFACE Libraries

- An INTERFACE library target does not directly create build output, though it may have properties set on it and it may be installed, exported, and imported.

```
add_library(root INTERFACE)
target_compile_features(root INTERFACE cxx_std_20)
```

Exporting Targets

Config.cmake.in produces Config.cmake

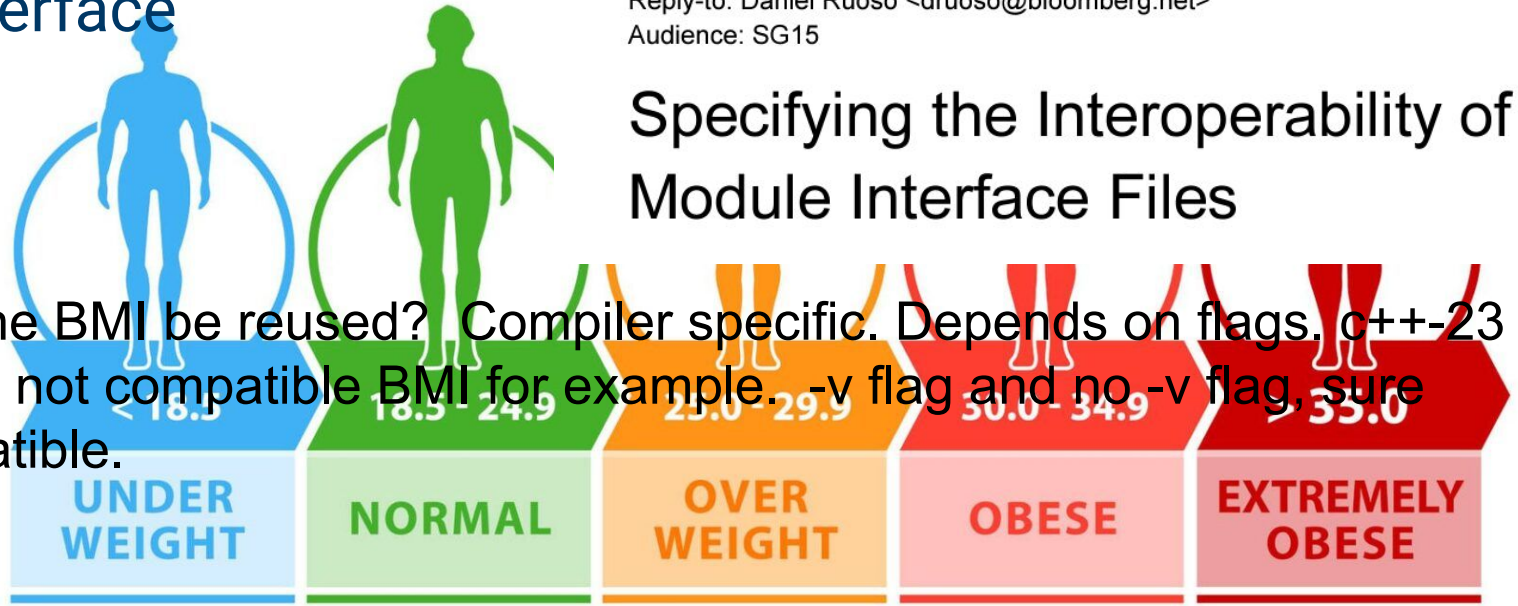
```
# Create imported target root
add_library(root INTERFACE IMPORTED)

set_target_properties(root PROPERTIES
  INTERFACE_COMPILE_DEFINITIONS "ROOT_VERSION=42"
  INTERFACE_COMPILE_FEATURES "cxx_std_11"
  INTERFACE_COMPILE_OPTIONS "\$<\$<NOT:\$<CONFIG:DEBUG>>:>;\$
)

# Create imported target trunk
add_library(trunk SHARED IMPORTED)

set_target_properties(trunk PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include/pa
)

# Create imported target leaf
add_library(leaf SHARED IMPORTED)

set_target_properties(leaf PROPERTIES
  INTERFACE_LINK_LIBRARIES "trunk"
)
```

BODY MASS INDEX BMI

Built Module Interface

Document Number: P2581R2
Date: 2022-10-12
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Specifying the Interoperability of Built Module Interface Files



Can the BMI be reused? Compiler specific. Depends on flags. c++-23 and c++20 not compatible BMI for example. -v flag and no -v flag, sure compatible.

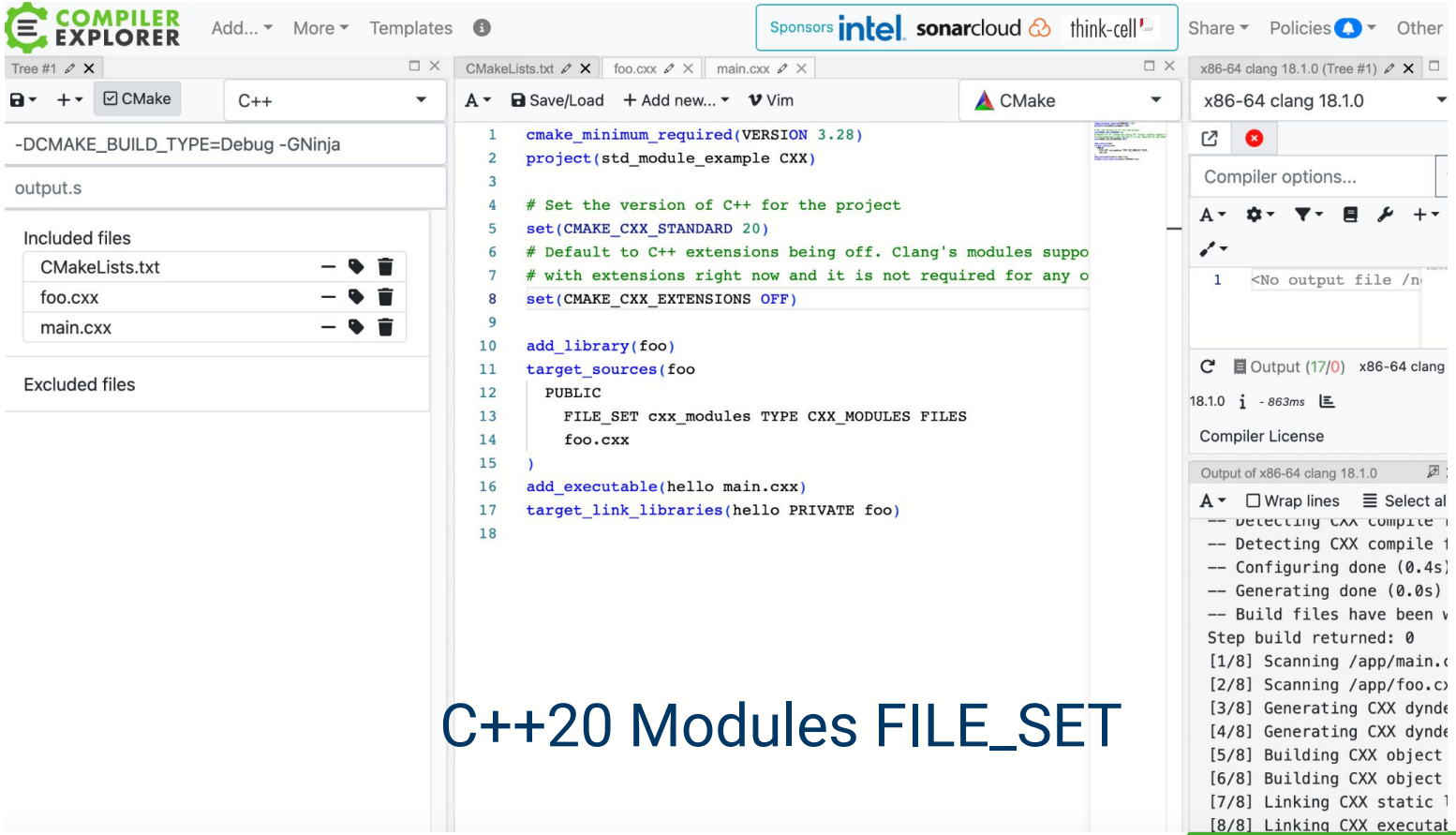
Modules BMI not compatible between c++20 and c++23

```
add_library(has_modules)
add_library(use20)
target_link_library(use20 has_modules)
add_library(use23)
target_link_library(use23 has_modules)

add_library(also_use20)
target_include_directories(also_use20 PRIVATE include/foo)
```

BMI can be shared between use20 and also_use20 because include is private

kitware

https://godbolt.org/z/KznT85hTf



C++20 Modules FILE_SET

import std

```
import std;

void hello_world(std::string const& name)
{
  std::cout << "Hello World! My name is " << name << std::endl;
}
```

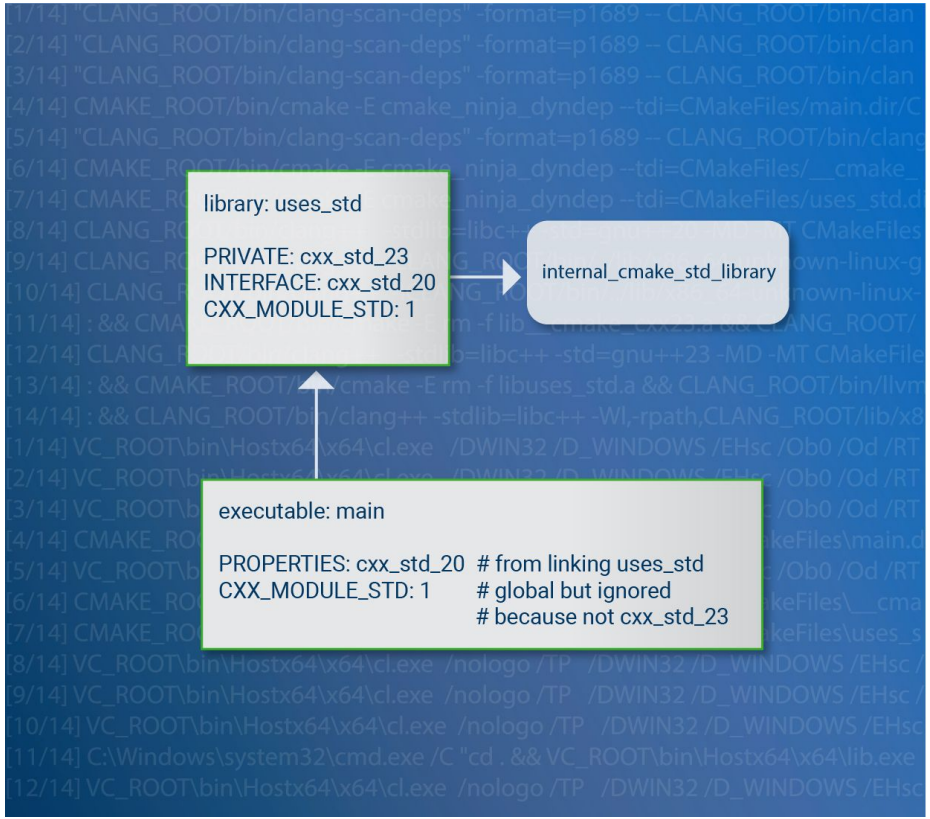
```
#include <string>

void hello_world(std::string const& name);

int main(int argc, char* argv[])
{
  hello_world(argv[0] ? argv[0] : "Voldemort?");
  return 0;
}
```

kitware

import std in CMake 3.29

```
# Tell CMake that we explicitly want `import std` for all targets
set(CMAKE_CXX_MODULE_STD 1)
# Make a library.
add_library(uses_std STATIC)
# Add sources.
target_sources(uses_std
  PRIVATE
    uses_std.cxx)
# Tell CMake we're using C++23 but only C++20 is needed to consume it
target_compile_features(uses_std
  PRIVATE   cxx_std_23
  INTERFACE cxx_std_20)
add_executable(main)
# Note that this source is *not* allowed to `import std` as it ends up
# with only C++20 support due to the `uses_std` INTERFACE requirements.
target_sources(main
  PRIVATE
    main.cxx)
target_link_libraries(main PRIVATE uses_std)
```

import std in CMake 3.29



kitware

Other build tools usage requirements

- xmake
- boost.build
- meson build
- bazel

xmake

xmake-io / xmake Public

Code | Issues 138 | Pull requests 4 | Discussi

Star 8.8k

New issue

Support private, public, int cmake #368

Closed | 9 tasks done | waruqi opened this issue on Mar

Inherit all compiler and linker settings, e.g. `add_cflags`, `add_defines`, `add_includedirs` ..

Roadmap

- [x] `add_xxx("", {public = true})`
  - [x] private = true (default)
  - [x] public = true (private + public)
  - [x] interface = true (only public)
- [x] `add_deps("", {inherit = false})`
  - [x] inherit = true (default)
- [x] `target:get("defines", {interface = true})`
- [x] `target:add("defines", "xxx", {public = true})`
- [x] `target:set("defines", "xxx", {private = true})`

References

#291

Examples

```
target("test")
    set_kind("static")
    add_files("src/*.c")
    add_includedirs("inc", {public = true})

-- inherit includedirs and link libtest.a
target("demo")
    set_kind("binary")
    add_deps("test")

-- only affect the compilation order
target("demo2")
    set_kind("binary")
    add_deps("test", {inherit = false})
```

kitware

boost.build

Most main target rules in Boost.Build have the same common signature:

```
rule rule-name (
     main-target-name :
     sources + :
     requirements * :
     default-build * :
     usage-requirements * )
```

- *main-target-name* is the name used to request the target on command line and to use it from other main targets. A main target name may contain alphanumeric characters, dashes (‘-’), and underscores (‘_’).
- *sources* is the list of source files and other main targets that must be combined.
- *requirements* is the list of properties that must always be present when this main target is built.
- *default-build* is the list of properties that will be used unless some other value of the same feature is already specified, e.g. on the command line or by propagation from a dependent target.
- *usage-requirements* is the list of properties that will be propagated to all main targets that use this one, i.e. to all its dependents.

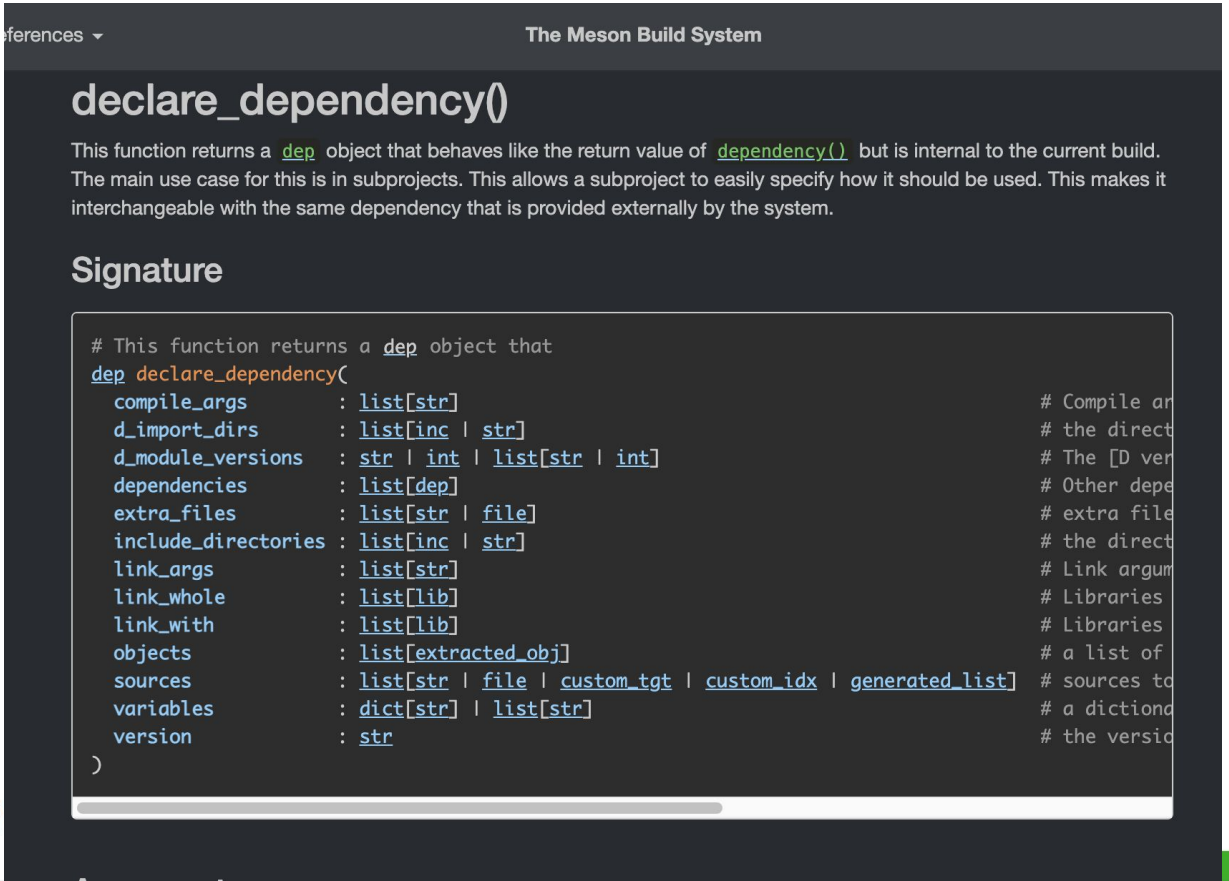
private requirement

public requirement

Some main target rules have a different list of parameters as explicitly stated in their documentation.

The actual requirements for a target are obtained by refining the requirements of the project where the target is declared with the explicitly specified requirements. The same is true for usage-requirements. More details can be found in the section called “Property refinement”

meson build



ferences ▾ The Meson Build System

declare_dependency()

This function returns a `dep` object that behaves like the return value of `dependency()` but is internal to the current build. The main use case for this is in subprojects. This allows a subproject to easily specify how it should be used. This makes it interchangeable with the same dependency that is provided externally by the system.

Signature

```
# This function returns a dep object that
dep declare_dependency(
  compile_args        : list[str]                                          # Compile ar
  d_import_dirs       : list[inc | str]                                    # the direct
  d_module_versions   : str | int | list[str | int]                        # The [D ver
  dependencies        : list[dep]                                          # Other depe
  extra_files         : list[str | file]                                   # extra file
  include_directories : list[inc | str]                                    # the direct
  link_args           : list[str]                                          # Link argum
  link_whole          : list[lib]                                          # Libraries
  link_with           : list[lib]                                          # Libraries
  objects             : list[extracted_obj]                                # a list of
  sources             : list[str | file | custom_tgt | custom_idx | generated_list]  # sources to
  variables           : dict[str] | list[str]                              # a dictiond
  version             : str                                                # the versio
)
```

kitware

bazel

To illustrate these rules, look at the following example.

```
cc_binary(
    name = "foo",
    srcs = [
        "foo.cc",
        "foo.h",
    ],
    deps = [":bar"],
)

cc_library(
    name = "bar",
    srcs = [
        "bar.cc",
        "bar-impl.h",
    ],
    hdrs = ["bar.h"],
    deps = [":baz"],
)

cc_library(
    name = "baz",
    srcs = [
        "baz.cc",
        "baz-impl.h",
    ],
    hdrs = ["baz.h"],
)
```

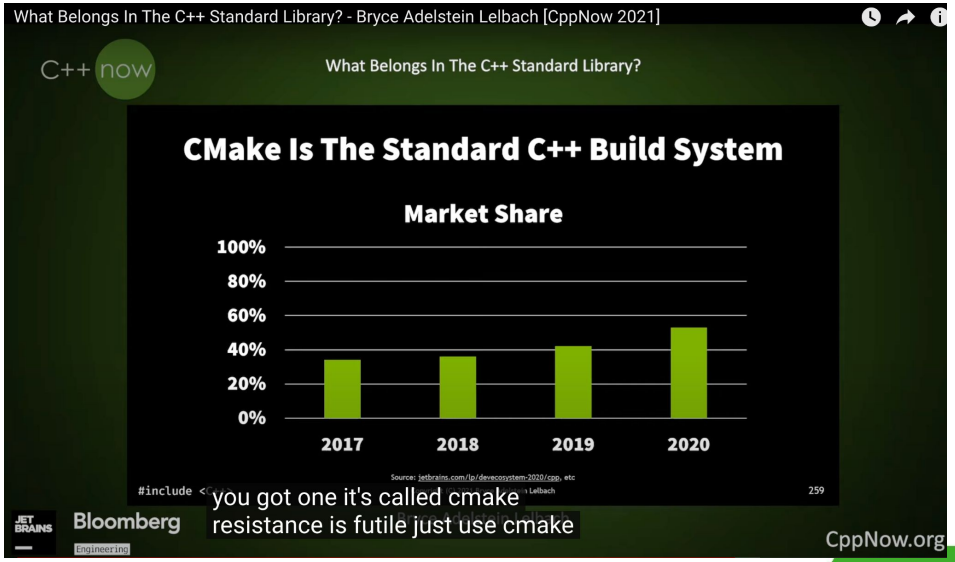
The allowed direct inclusions in this example are listed in the table below. For example `foo.cc` is allowed to directly include `foo.h` and `bar.h`, but not `baz.h`.

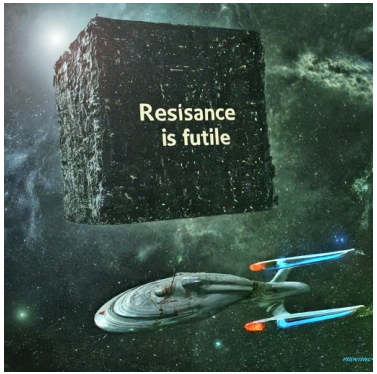
Seems to be includes only, but I could have missed something

kitware

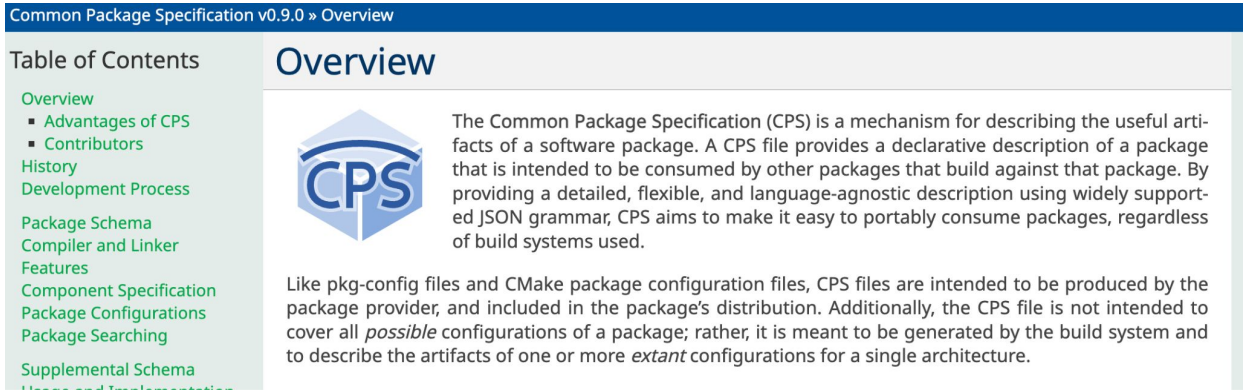
The answer just use CMake!

- 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



Common Package Specification v0.9.0 » Overview

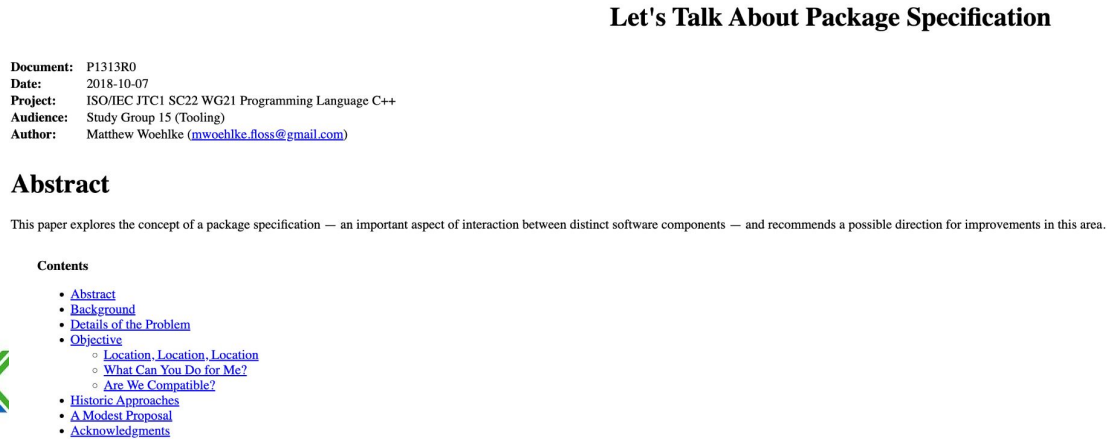
Table of Contents

- Overview
  - Advantages of CPS
  - Contributors
- History
- Development Process
- Package Schema
- Compiler and Linker Features
- Component Specification
- Package Configurations
- Package Searching
- Supplemental Schema

Overview

The Common Package Specification (CPS) is a mechanism for describing the useful artifacts of a software package. A CPS file provides a declarative description of a package that is intended to be consumed by other packages that build against that package. By providing a detailed, flexible, and language-agnostic description using widely supported JSON grammar, CPS aims to make it easy to portably consume packages, regardless of build systems used.

Like pkg-config files and CMake package configuration files, CPS files are intended to be produced by the package provider, and included in the package's distribution. Additionally, the CPS file is not intended to cover all *possible* configurations of a package; rather, it is meant to be generated by the build system and to describe the artifacts of one or more *extant* configurations for a single architecture.



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

- Abstract
- Background
- Details of the Problem
- Objective
  - Location, Location, Location
  - What Can You Do for Me?
  - Are We Compatible?
- Historic Approaches
- A Modest Proposal
- Acknowledgments

- 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
- 2024: working on export in CMake

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

CPS provides a communication path from library maintainers to consumers

- **common consumers**
  - package managers
  - build systems
  - IDEs

Open–closed principle

17 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In object-oriented programming, the **open–closed principle** (OCP) states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*";[1] that is, such an entity can allow its behaviour to be extended without modifying its source code.

The name *open–closed principle* has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

The open–closed principle is one of the five SOLID principles of object-oriented design.

Meyer's open–closed principle [edit]

Bertrand Meyer is generally credited for having originated the term *open–closed principle*,[2] which appeared in his 1988 book *Object-Oriented Software Construction*.[1]:23

- A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

The open-closed principle was introduced by Bertrand Meyer.

SOLID

Principles

Single responsibility

Open–closed

Target Model Open and Closed

- **Interface needs to be closed**
  - Consumers can just use the library
- **Interface needs to be open**
  - Maintainer needs to be able to add new features to the library

```
add_executable(myprogram)
target_link_library(myprogram PRIVATE funlibrary)
```

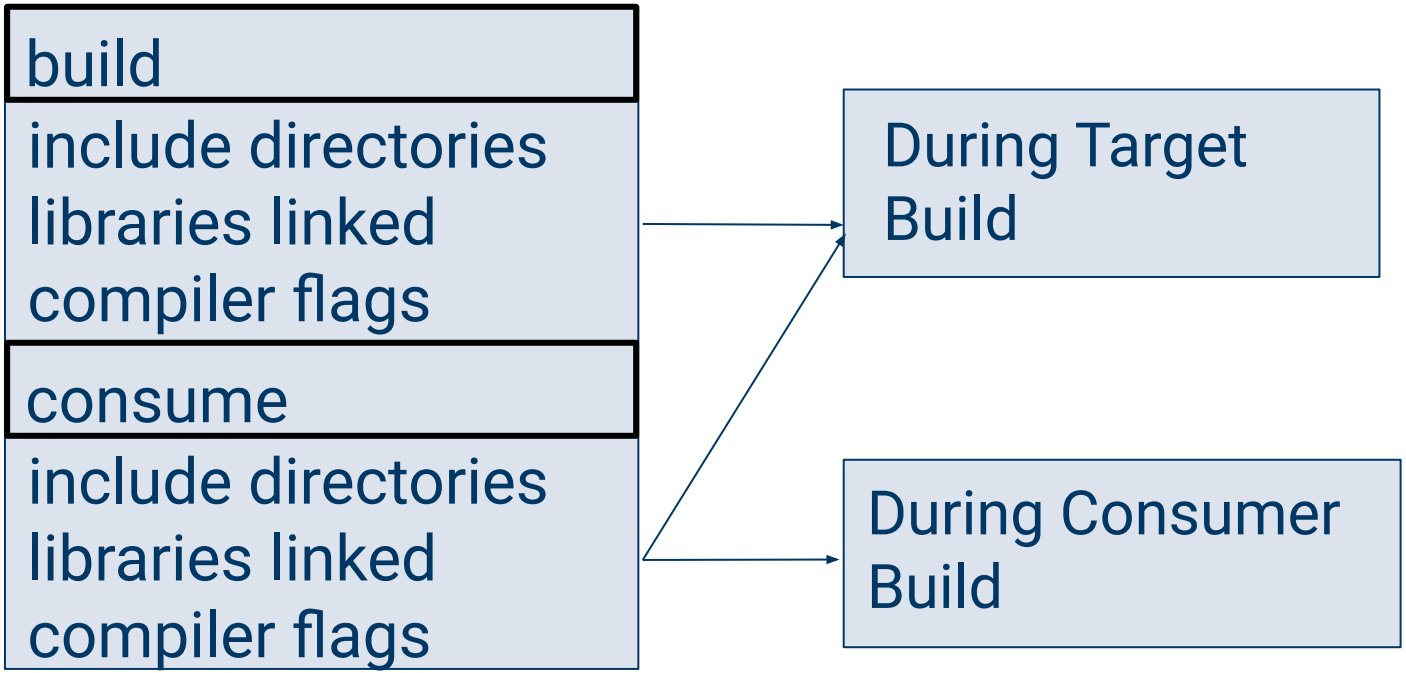
Open/Closed target funlibrary

```
# in 2024 and 2025
add_executable(myprogram)
target_link_library(myprogram PRIVATE funlibrary)
```

```
# in 2024
add_library(funlibrary)
target_link_library(funlibrary PUBLIC hiddenfun)
```

```
# in 2025
add_library(funlibrary)
target_link_library(funlibrary PUBLIC morefun)
```

Target Model



Target model impact

In the C++ Developer Survey (2023), the top issues all revolve around build issues:

1. Managing dependencies.
2. Setting up CI/CD pipelines.
3. Build times.
4. Dealing with CMake.
5. Setting up toolchains and IDEs.

Summ

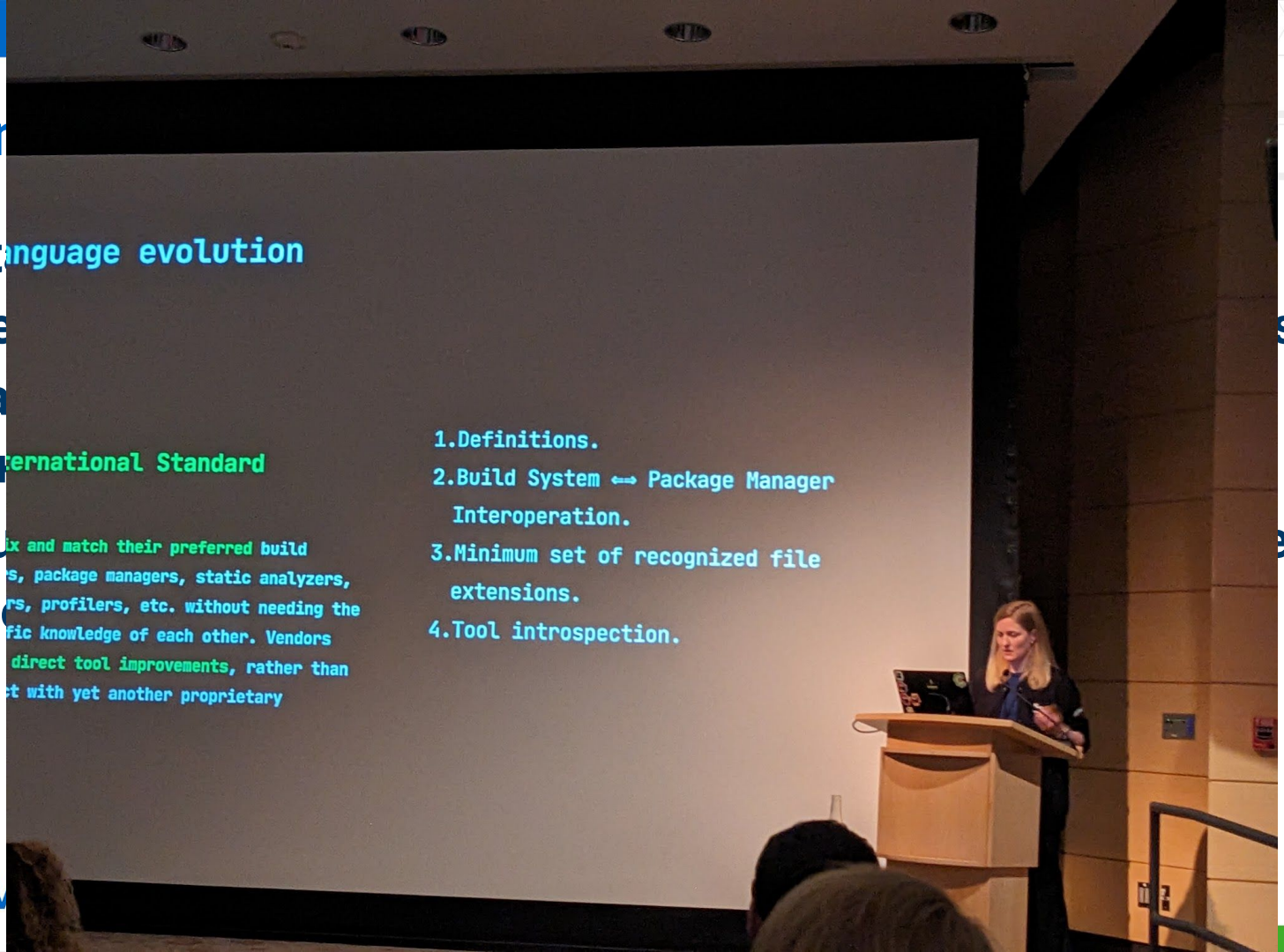
- St
- Re
- C+
- Bu

pa

th

s to

e to



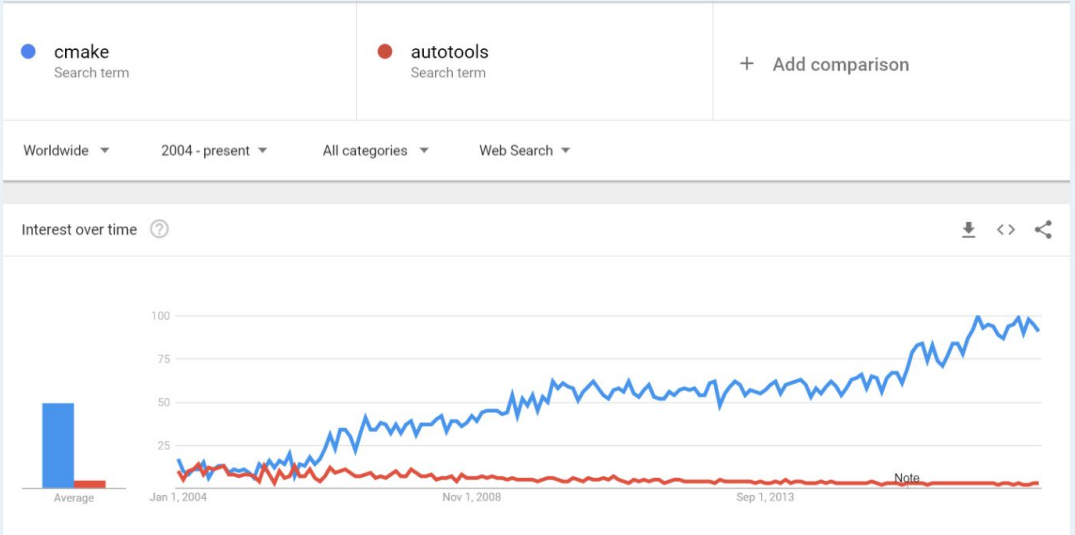
kitw

Thanks

2006

KDE switches to CMake

- KDE developer Alexander Neundorf was the champion that helped KDE adopt CMake.
- Kitware worked hard to create a prototype build system in a few short weeks.
- CMake was able to quickly build more of KDE than the scons system they were using previously.
- KDE was able to port applications to Windows and Apple very quickly with CMake.
- CMake got new features, including shared library versioning and installation rpath re-writes without relinking.
- CMake was distributed by all major linux distributions to support KDE.



kitware

Thanks

2012

Modern CMake is written by Stephen Kelly

- Comprehensive usage requirements, started by Stephen Kelly's Modern CMake, allowed projects to package up "how to use me" in a nice way (cf. VTK's build system rewrite to use them "everywhere" in 2019).

kitware

More thanks

Ben Boeckel

Brad King

Craig Scott

Matthew Woehlke

CMake open source community

C++ Now

Bloomberg

Engineering

kitware

Thank you! Questions

kitware