CMake and Conan: past, present and future.

MeetingC++23

Diego Rodriguez-Losada
Berlin, Nov-2023

JFrog

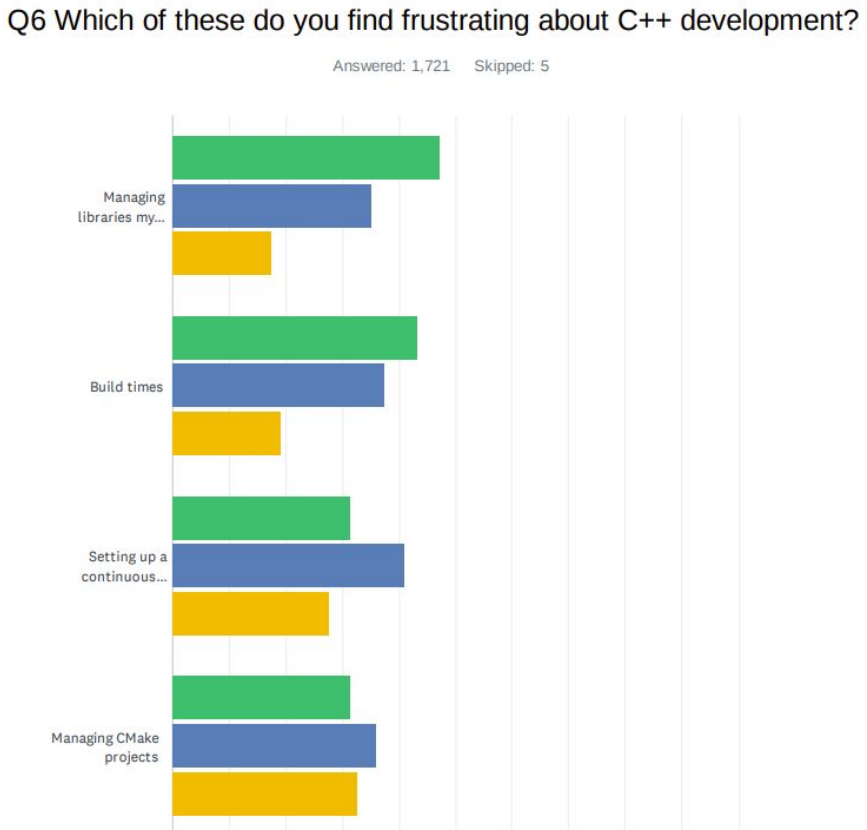
CONAN 2.0

C/C++ Package Manager

CMake and Conan in 2023

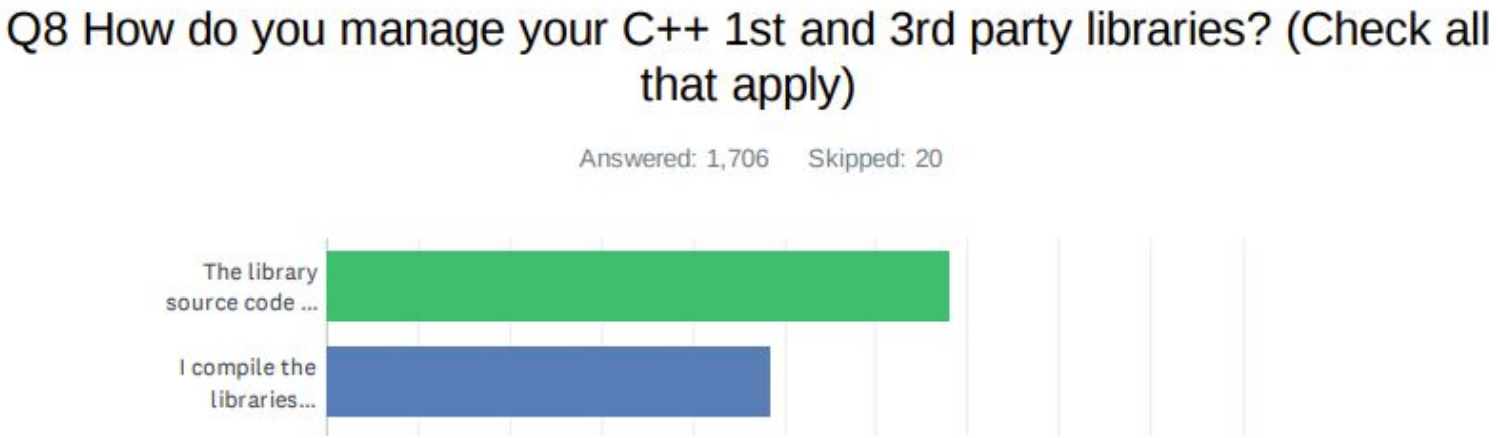
Top 4 pains:

- Managing libraries
- Build times
- Setting CI
- Managing CMake



*ISO C++ survey 2023

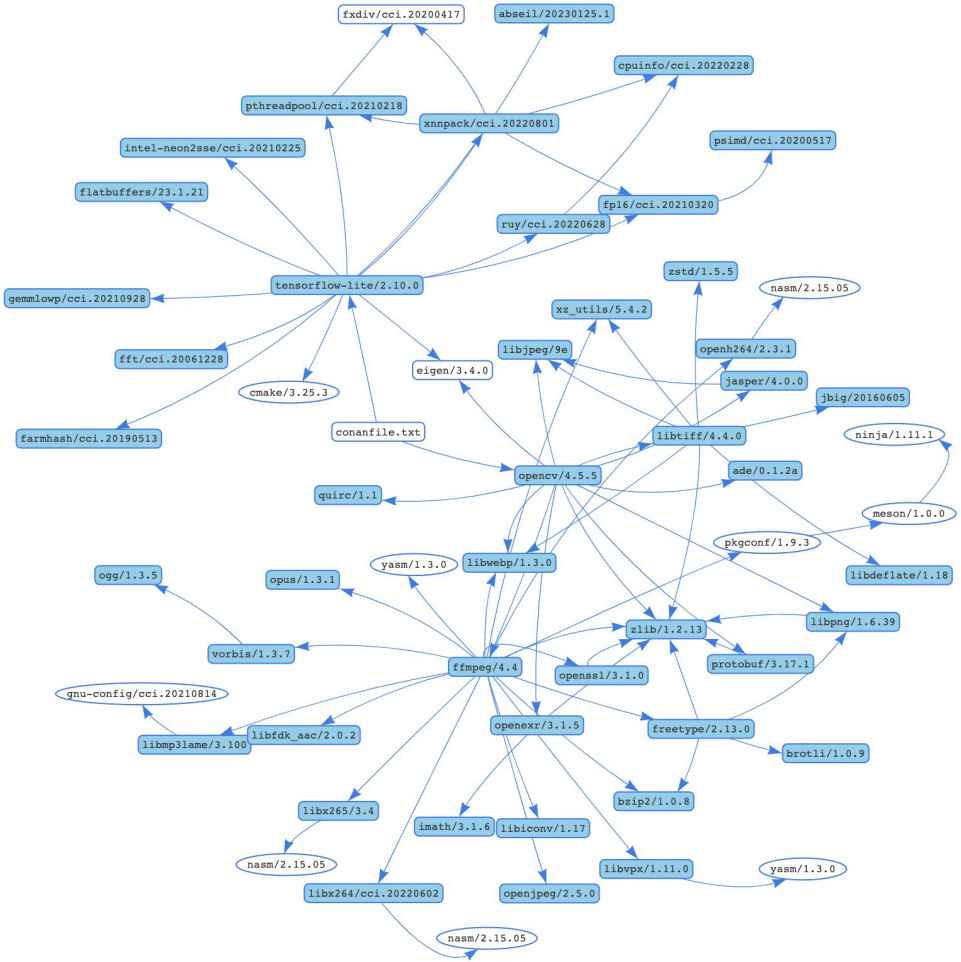
CMake and Conan in 2023



Demo

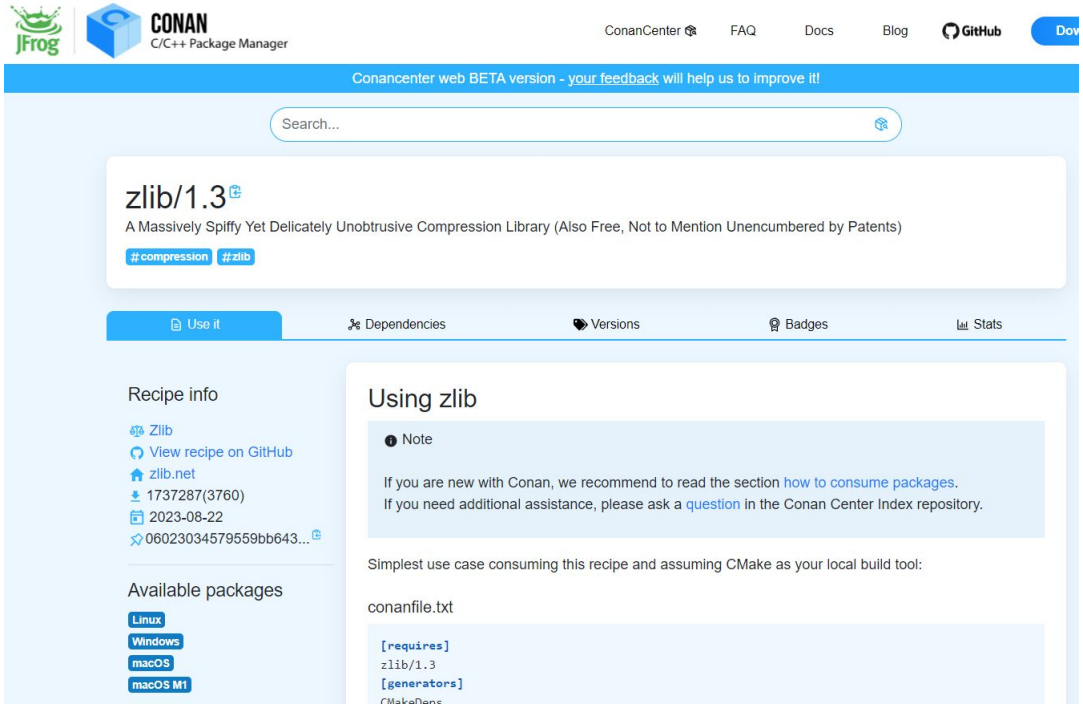
CMake and Conan in 2023

- Fully transparent integration
- Commanded by cmake
- Exactly same commands and flow in all platforms



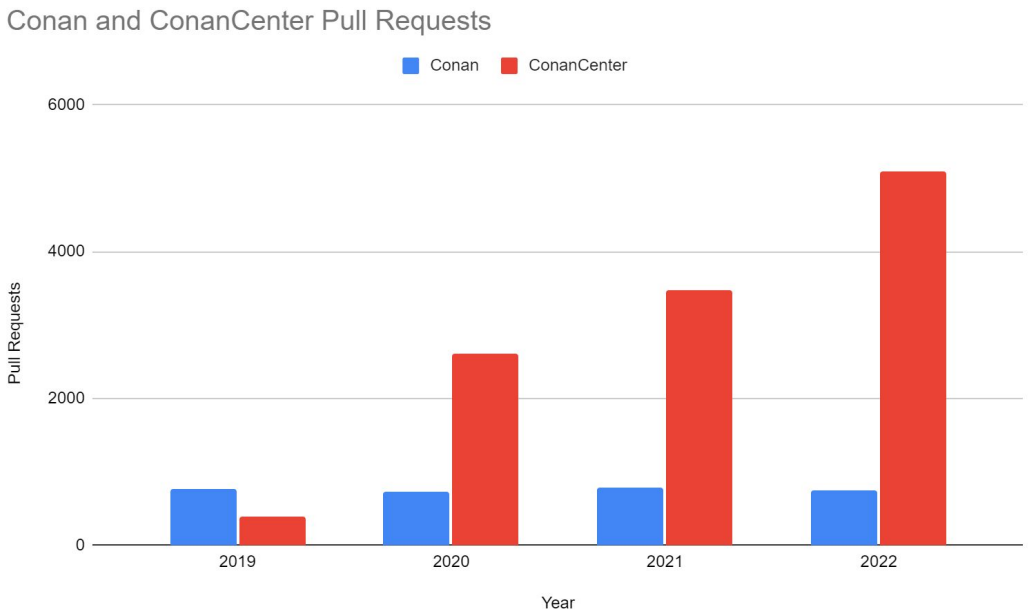
Conan in 2023

- ConanCenter:
  - 1500 recipes x 3 versions x 100 binaries = **500K packages**
  - **250K conanfile requests/day**



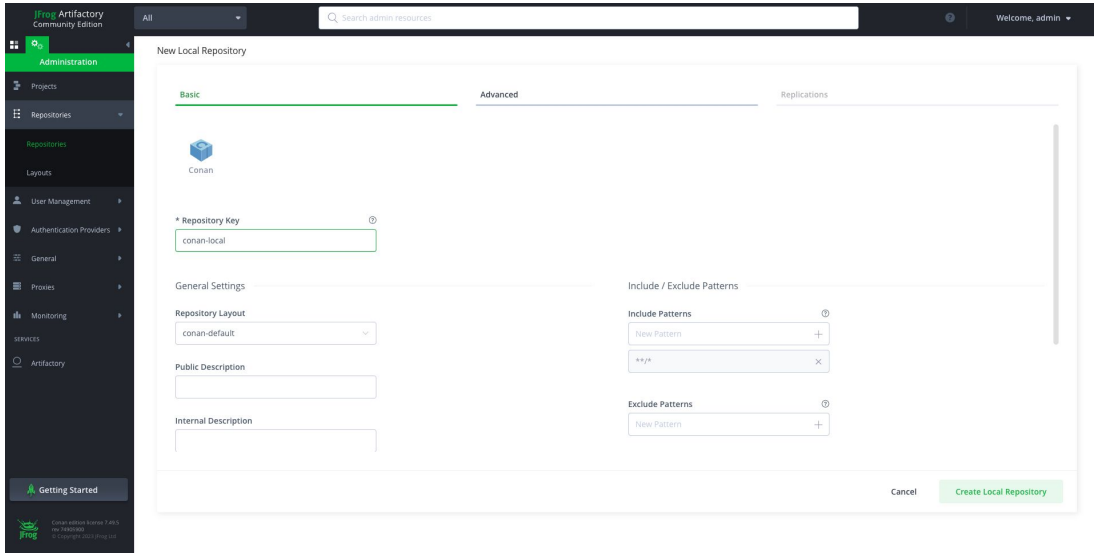
Conan in 2023

- + 5000 PRs / year



Conan in 2023

- Private packages
  - **75%** of users don't use ConanCenter
  - Client **750K downloads/month** (PyPI 1% critical project)
  - Many thousands of organizations using Artifactory-Conan in production, including many of Fortune 100



CMake and Conan: **past**, present and future.

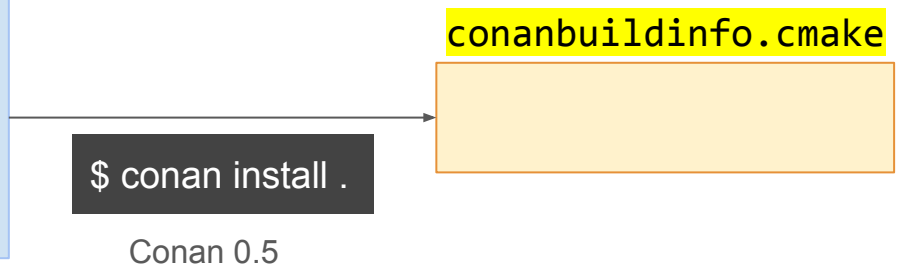
MeetingC++23

Diego Rodriguez-Losada
Berlin, Nov-2023

Once upon a time… in 2015

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake
```



CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.12)
project(PackageTest CXX)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS})
```

conanbuildinfo.cmake

```
macro(conan_basic_setup)
    # simplified!


    # dependencies info
    conan_global_flags()

    # Toolchain-related info
    conan_set_rpath()
    conan_set_fpic()
    conan_set_std()
    conan_check_compiler()
    conan_set_libcxx()
    conan_set_vs_runtime()
endmacro()
```

```
set(CONAN_INCLUDE_DIRS_ZLIB "path/.../include")
set(CONAN_LIB_DIRS_ZLIB "path/.../lib")
set(CONAN_LIBS_ZLIB zlib)
set(CONAN_CXX_STD 14)
```

```
macro(conan_global_flags)
include_directories(${CONAN_INCLUDE_DIRS})
link_directories(${CONAN_LIB_DIRS})
endmacro()
```

```
macro(conan_set_std)
   set(CMAKE_CXX_STANDARD ${CONAN_CXX_STD})
   set(CMAKE_CXX_EXTENSIONS ${CONAN_CXX_EXT})
endmacro()
```

Where the information comes from? (toolchain)

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake
```

$ conan install . (-pr=default)

Conan 0.14.0 (Oct-2016)

conanbuildinfo.cmake

default (profile)

```
[settings]
compiler=gcc
compiler.version=11
compiler.libcxx=...
compiler.cppstd=14
...
```

conanbuildinfo.cmake

```
set(CONAN_INCLUDE_DIRS_ZLIB "path/.../include")
set(CONAN_LIB_DIRS_ZLIB "path/.../lib")
set(CONAN_LIBS_ZLIB zlib)
set(CONAN_CXX_STD 14)
```

Where the information comes from? (dependencies)

conanfile.py

```
zlib/1.2.13
```

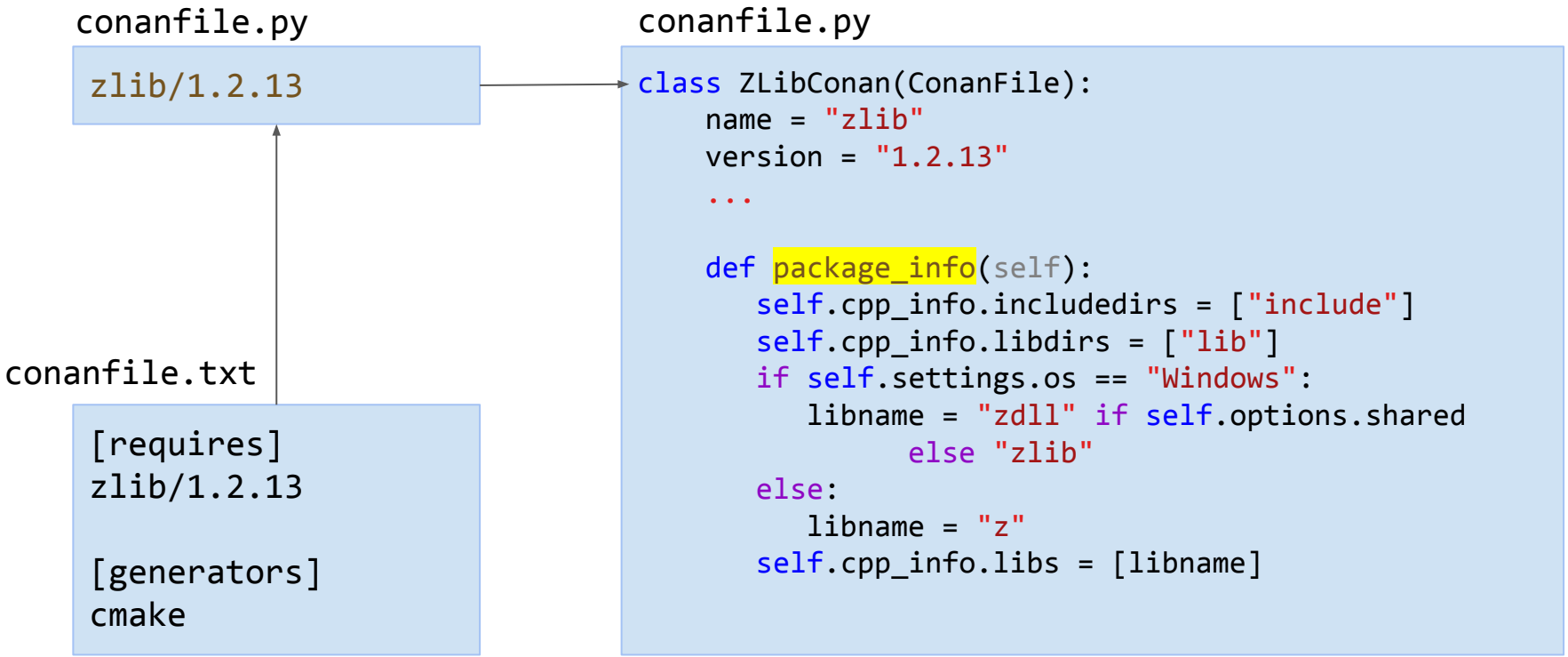
conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake
```

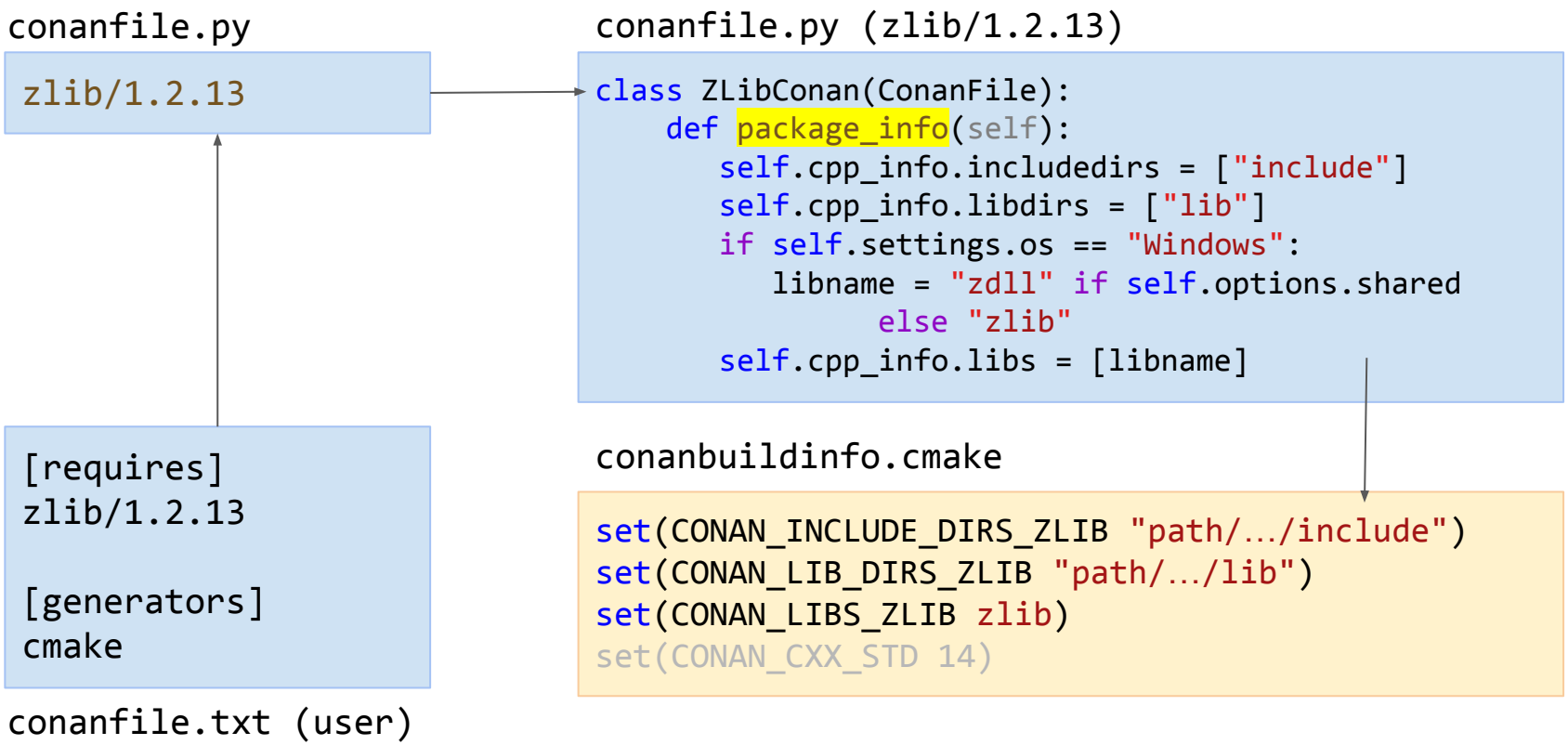
conanfile.py

```
class ZLibConan(ConanFile):
    name = "zlib"
    version = "1.2.13"
    settings = "os", "compiler", "build_type", "arch"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        self.cpp_info.libs = ["zlib"]
```

Where the information comes from? (dependencies)



Where the information comes from? (dependencies)



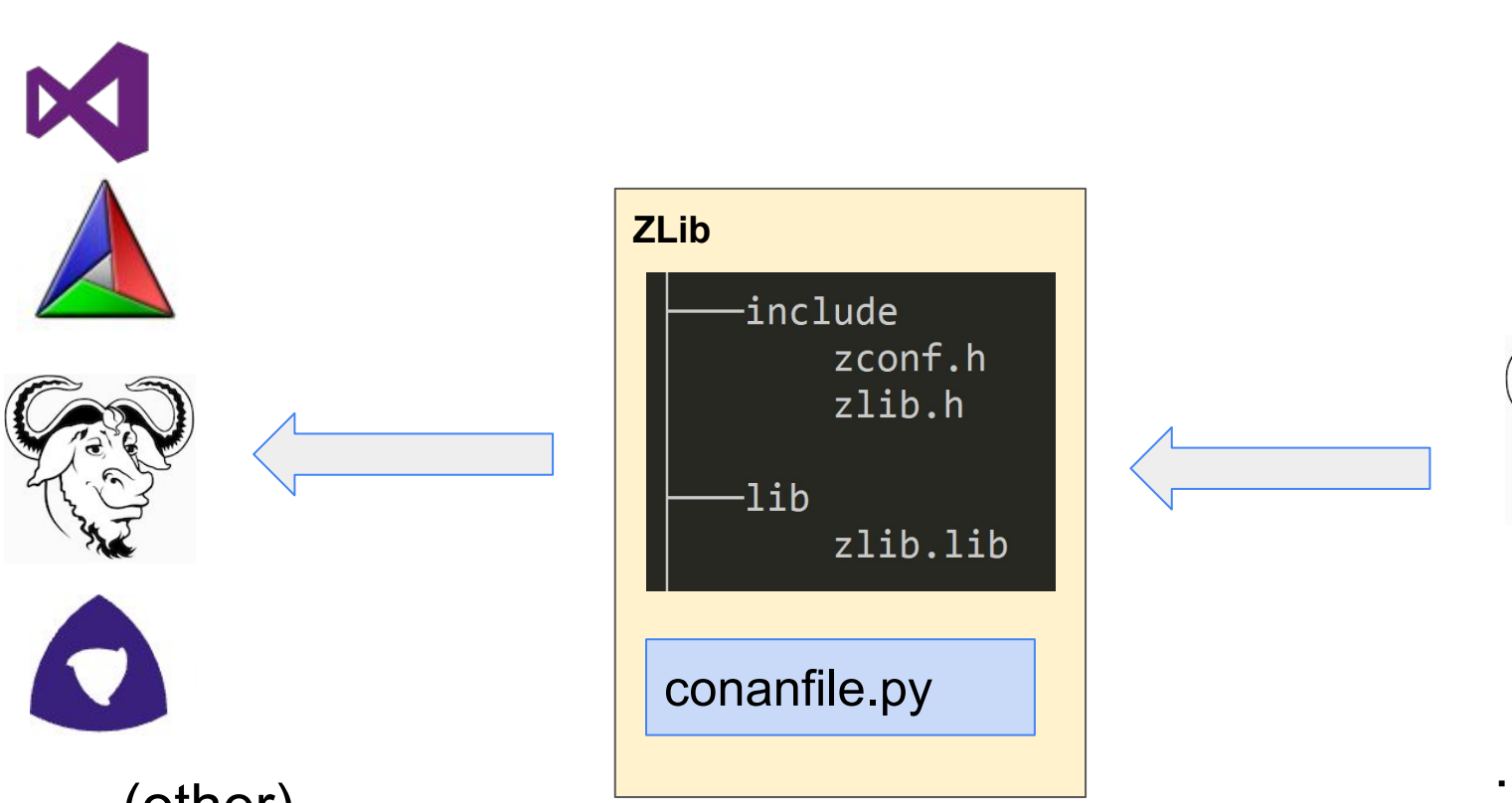
Why globals? Why not from xxx.cmake files?

- Many libraries and projects:
  - Do not have cmake find modules
  - Do not generate config.cmake files (there are other build systems!)
  - Generate them poorly
- Interoperability

> Version 3.0 was released in June 2014.[8] It has been described as the beginning of "Modern CMake".[9] Experts now advise to avoid variables in favor of *targets* and *properties*.[10]
>
> Binna, Manuel (22 July 2018). "Effective Modern CMake".

https://steveire.wordpress.com/**2017/11/05/embracing-modern-cmake**/

package_info() ⇔ Interoperability



Introducing targets

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake
```

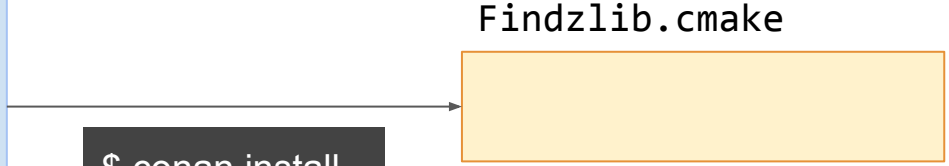
CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 2.8.12)
project(PackageTest CXX)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)

add_executable(example example.cpp)
target_link_libraries(example CONAN_PKG::zlib)
```

Aim for transparent integration: cmake_find_package

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package
```



CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(zlib)

# Global approach
if(ZLIB_FOUND)
   include_directories(${ZLIB_INCLUDE_DIRS})
   target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()
```

Aim for transparent integration: cmake_find_package

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package
```

$ conan install .

Conan 1.4 (2018)

Findzlib.cmake

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(zlib)

# Modern CMake targets approach
target_link_libraries(helloworld zlib::zlib)
```

Following “upstream” naming

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package
```

$ conan install .

Conan 1.4 (2018)

Findzlib.cmake

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(zlib)

# Modern CMake targets approach
target_link_libraries(helloworld zlib::zlib)
```

Following “upstream” naming

conanfile.py (zlib/1.2.13)

```
name = "zlib"
version = "1.2.13"

def package_info(self):
    self.cpp_info.includedirs = ["include"]
    self.cpp_info.libdirs = ["lib"]
    ...

    self.cpp_info.names["cmake_find_package"] = "ZLIB"
```

Following “upstream” naming

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package
```

$ conan install .

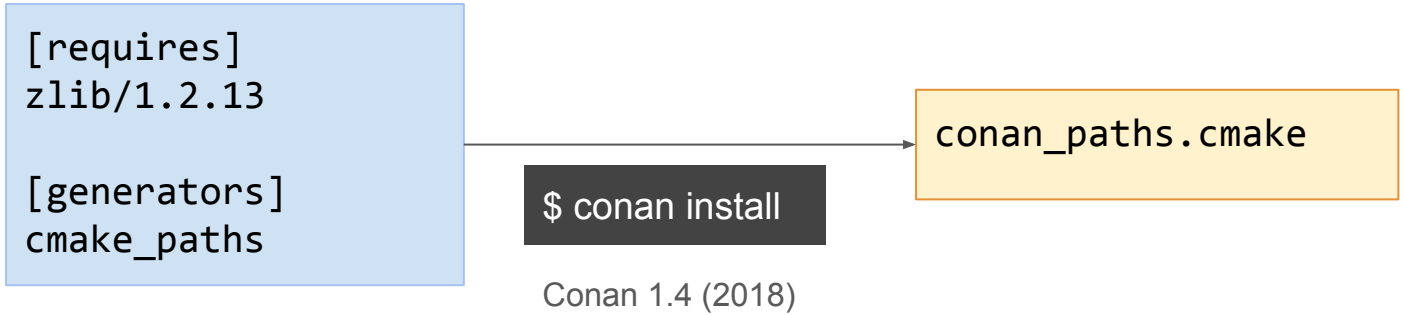
Conan 1.4 (2018)

FindZLIB.cmake

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

# Modern CMake targets approach
target_link_libraries(helloworld ZLIB::ZLIB)
```

cmake_paths

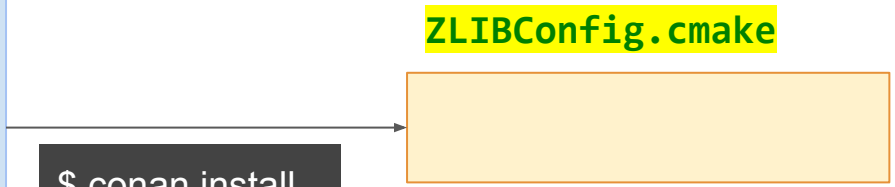



```
set(CONAN_ZLIB_ROOT "C:/../.conan/data/zlib/1.2.13/_/_/package/5a61....d48b")
set(CMAKE_MODULE_PATH ${CONAN_ZLIB_ROOT} ${CMAKE_MODULE_PATH} ${CMAKE_CURRENT_LIST_DIR})
set(CMAKE_PREFIX_PATH ${CONAN_ZLIB_ROOT} ${CMAKE_PREFIX_PATH} ${CMAKE_CURRENT_LIST_DIR})
```

From modules to config: cmake_find_package_multi

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package_multi
```



Conan 1.14 (2019)

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

target_link_libraries(helloworld ZLIB::ZLIB)
```

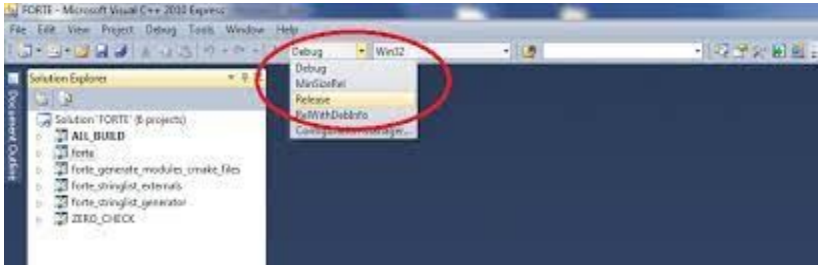
Multi-config vs single-config

Single config

```
$ mkdir release && cd release
$ cmake .. -G “Unix Makefiles” -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
$ mkdir debug && cd debug
$ cmake .. -G “Unix Makefiles” -DCMAKE_BUILD_TYPE=Debug
$ cmake --build .
```

Multi config

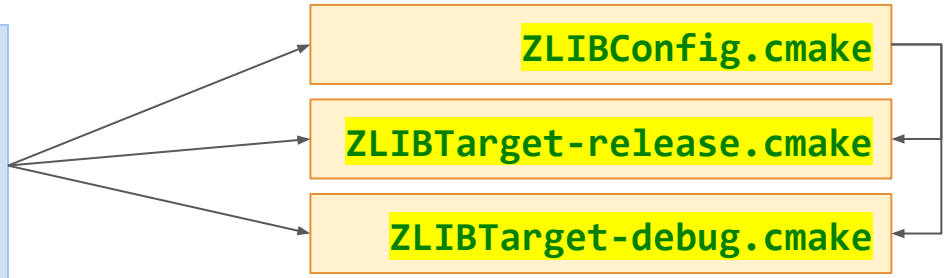
```
$ mkdir build && cd build
$ cmake .. -G “Visual Studio 17 2022”
# One project contains 2 configs
$ cmake --build . --config Release
$ cmake --build . --config Debug
```



From modules to config: cmake_find_package_multi

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake_find_package_multi
```




```
$ conan install . -s build_type=Release
$ conan install . -s build_type=Debug
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

target_link_libraries(helloworld ZLIB::ZLIB)
```

Problems

Too intrusive:

- include(.../conanbuildinfo.cmake)
- Even ${CONAN_LIBS} or CONAN_PKG::xxx

Too messy:

- cmake + (cmake_find_package | cmake_find_package_multi | cmake_paths)

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
cmake
cmake_find_package_multi
```

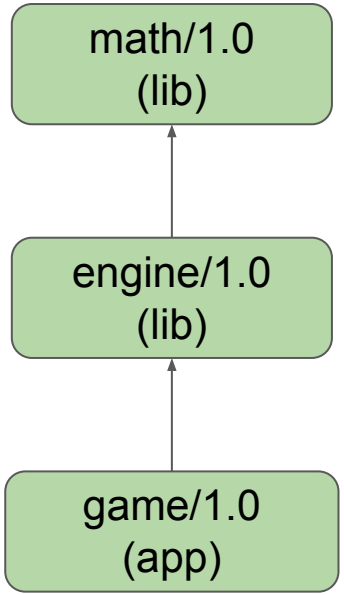
CMakeLists.txt

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

add_executable(example example.cpp)
find_package(ZLIB)
target_link_libraries(example ZLIB::ZLIB)
```

Problems

Suboptimal:

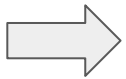
- overlinking



cmake-conan: calling Conan from CMake

“Canonical” flow

```
$ conan install …
$ cmake …
```



“cmake-conan” flow

```
$ cmake … # calls conan install internally
```

Conan 1.X cmake-conan integration

CMakeLists.txt

Download the file

include

Create conanfile.txt on the fly

Deduce profile from CMake config

Call conan install

```
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
  file(DOWNLOAD "<url>/0.18.1/conan.cmake"
                "${CMAKE_BINARY_DIR}/conan.cmake"
                TLS_VERIFY ON)
endif()
include(${CMAKE_BINARY_DIR}/conan.cmake)

conan_cmake_configure(REQUIRES fmt/6.1.2
                      GENERATORS cmake_find_package)

conan_cmake_autodetect(settings)

conan_cmake_install(PATH_OR_REFERENCE . BUILD missing
                    REMOTE conancenter SETTINGS ${settings})

find_package(fmt)
```

Conan 1.X cmake-conan integration

```
$ cmake ...
-- Conan executing: C:/Users/Diego/Envs/conan36/Scripts/conan.exe install .
--remote conancenter --build missing --settings arch=x86_6
Release --settings compiler=Visual Studio --settings compiler.version=17
--settings compiler.runtime=MD --settings os=Windows

Requirements
    fmt/6.1.2 from 'conancenter' - Downloaded

Installing (downloading, building) binaries...
fmt/6.1.2: Package '5a61a86bb3e07ce4262c80e1510f9c05e9b6d48b' created

conanfile.txt: Generator cmake_find_package created Findfmt.cmake

-- Found fmt: 6.1.2 (found version "6.1.2")
-- Build files have been written to: C:/Users/Diego/conanws/cmake-conan
```

cmake-conan 1.X problems

Extremely intrusive:

- Download + include + generate_conanfile + detect-profile + conan install
- Too much boilerplate in the CMakeLists.txt

```
conan_cmake_configure(REQUIRES fmt/6.1.2
                      GENERATORS cmake_find_package)
```

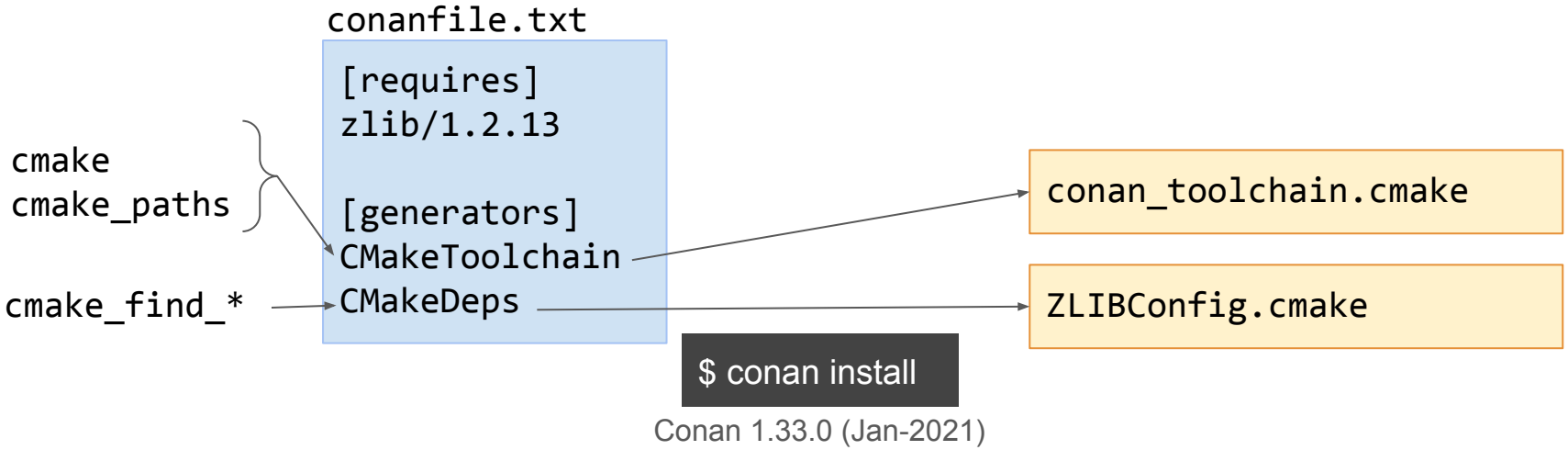
Need to be protected to avoid recursion when building packages

Confusing experience when something goes wrong, difficult to isolate problems

CMake and Conan: past, **present** and future.

MeetingC++23

Modern: CMakeToolchain and CMakeDeps



Conan 1.33.0 (Jan-2021)

CMakeLists.txt

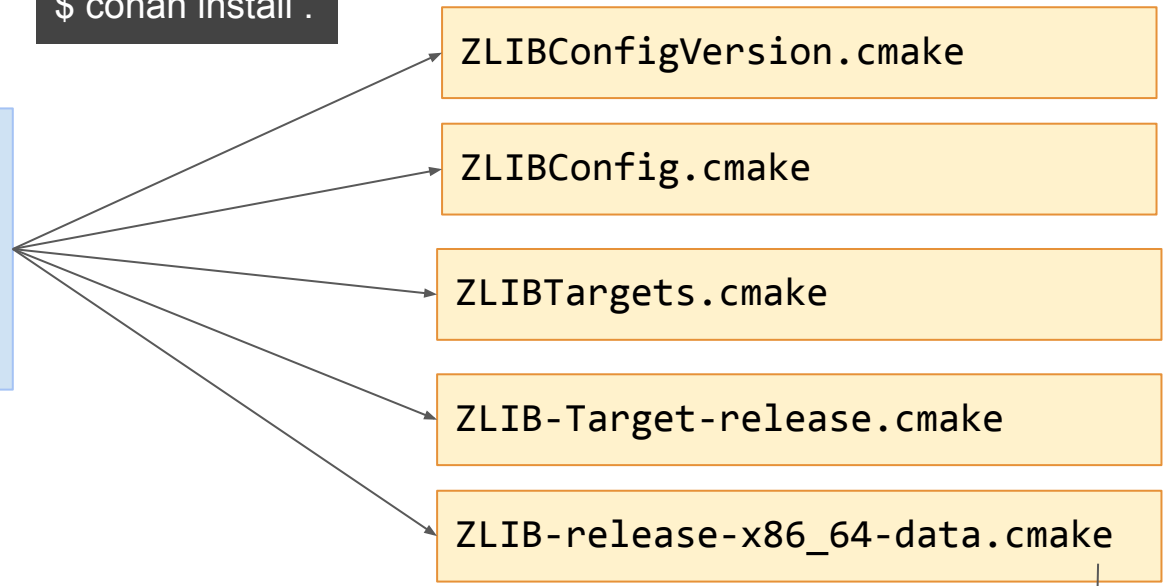
```
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

target_link_libraries(helloworld ZLIB::ZLIB)
```

CMakeDeps

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
CMakeDeps
```

$ conan install .




```
set(zlib_PACKAGE_FOLDER_RELEASE "C:/Users/Diego/.conan2/p/zlib0539bd5d6476e/p")
set(zlib_INCLUDE_DIRS_RELEASE "${zlib_PACKAGE_FOLDER_RELEASE}/include")
set(zlib_LIB_DIRS_RELEASE "${zlib_PACKAGE_FOLDER_RELEASE}/lib")
set(zlib_LIBS_RELEASE zlib)
```

CMakeDeps

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
CMakeDeps
```

$ conan install .

ZLIBConfigVersion.cmake

ZLIBConfig.cmake

ZLIBTargets.cmake

ZLIB-Target-release.cmake

ZLIB-release-x86_64-data.cmake

ZLIBTargets.cmake

```
if(NOT TARGET ZLIB::ZLIB)
    add_library(ZLIB::ZLIB INTERFACE IMPORTED)
    message(${ZLIB_MESSAGE_MODE} "Conan: Target declared 'ZLIB::ZLIB'")
endif()
```

CMakeDeps

$ conan install .

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
CMakeDeps
```

ZLIBConfigVersion.cmake

ZLIBConfig.cmake

ZLIBTargets.cmake

ZLIB-Target-release.cmake

ZLIB-release-x86_64-data.cmake

ZLIB-Target-release.cmake

```
set_property(TARGET ZLIB::ZLIB
             PROPERTY INTERFACE_INCLUDE_DIRECTORIES
             $<$<CONFIG:Release>:${zlib_INCLUDE_DIRS_RELEASE}> APPEND)
```

Configuring CMakeDeps

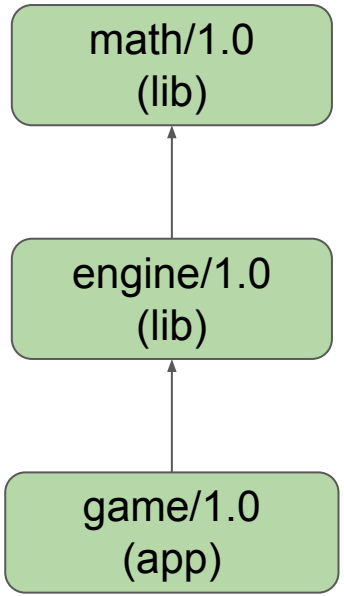
conanfile.py (zlib/1.2.13)

```
name = "zlib"
version = "1.2.13"

def package_info(self):
    self.cpp_info.set_property("cmake_target_name", "ZLIB")
```

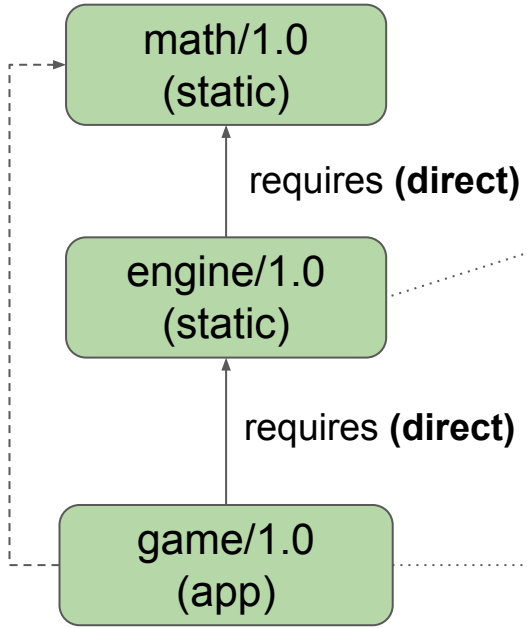
cmake_file_name
cmake_target_name
cmake_target_aliases
cmake_find_mode
cmake_module_file_name
cmake_module_target_name
cmake_build_modules
cmake_set_interface_link_directories
nosoname
cmake_config_version_compat

Propagation of traits in CMakeDeps

- Transitive dependencies not included or overlinking



Linkage requirements propagation (app->static->static)



requires
(transitive)

```
class Engine(ConanFile):
    def requirements(self):
        self.requires("math/1.0",
            headers=True, libs=True)
```

```
class Game(ConanFile):
    def requirements(self):
        self.requires("engine/1.0",
            headers=True, libs=True)
```

```
self.requires("math/1.0",
              headers=False,
              libs=True)
```

math-config.cmake

```
set_property(TARGET math::math PROPERTY INTERFACE_LINK_LIBRARIES ...)
set_property(TARGET math::math PROPERTY INTERFACE_INCLUDE_DIRECTORIES ...)
```

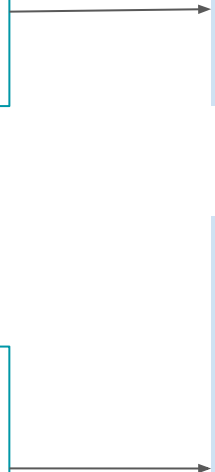
Shared library linking static

math2.cpp

```
int add(int a, int b){
  return a + b;
}
```

engine.cpp

```
#include "math2.h"
int move3d(int x, int y, int z){
    return add(x, add(y, z));
}
```



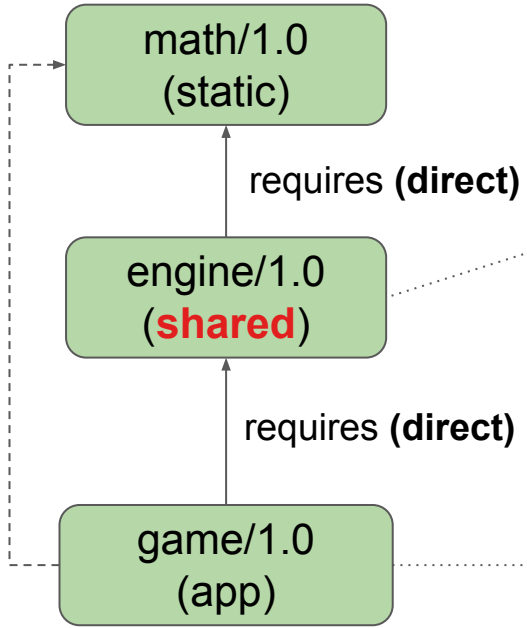
math2.lib

```
?add@@YAHHH@Z (int __cdecl add(int,int)):
  ...
  ...00011: 03 C8              add         ecx,eax
  ...
```

engine.dll

```
?move3d@@YAHHHH@Z (int __cdecl move3d(int,int,int)):
  ...
  ...00021: 8B 4C 24 30        mov         ecx,dword ptr [rsp+30h]
  ...00025: 8B 54 24 40        mov         edx,dword ptr [rsp+40h]
  ...00029: 8B 4C 24 38        mov         ecx,dword ptr [rsp+38h]
  ...0002D: E8 00 00 00 00     call        ?add@@YAHHH@Z
  ...00032: 8B D0              mov         edx,eax
  ...00034: 8B 4C 24 30        mov         ecx,dword ptr [rsp+30h]
  ...00038: E8 00 00 00 00     call        ?add@@YAHHH@Z
?add@@YAHHH@Z (int __cdecl add(int,int)):
  ...
  ...00011: 03 C8              add         ecx,eax
```

Linkage requirements propagation (app->shared->static)



requires
(transitive)

```
class Engine(ConanFile):
    def requirements(self):
        self.requires("math/1.0",
            headers=True, libs=True)
```

```
class Game(ConanFile):
    def requirements(self):
        self.requires("engine/1.0",
            headers=True, libs=True)
```

```
self.requires("math/1.0",
              headers=False,
              libs=False)
```

math-config.cmake

```
set_property(TARGET math::math PROPERTY INTERFACE_LINK_LIBRARIES ...)
set_property(TARGET math::math PROPERTY INTERFACE_INCLUDE_DIRECTORIES ...)
```

CMakeToolchain

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
CMakeToolchain
```

conan_toolchain.cmake

$ conan install

Conan 1.33.0 (Jan-2021)

```cmake
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(ZLIB)

target_link_libraries(helloworld ZLIB::ZLIB)
```

CMakeToolchain

Extra user toolchain files

Platform and toolset

Runtime

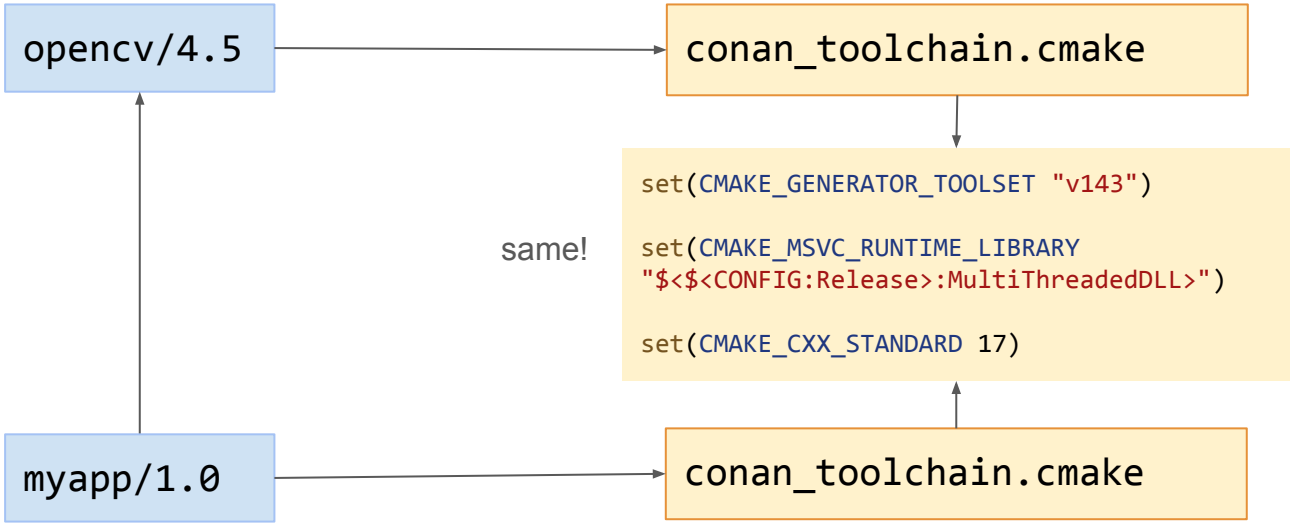
cppstd

Extra flags

Locating dependencies config.cmake

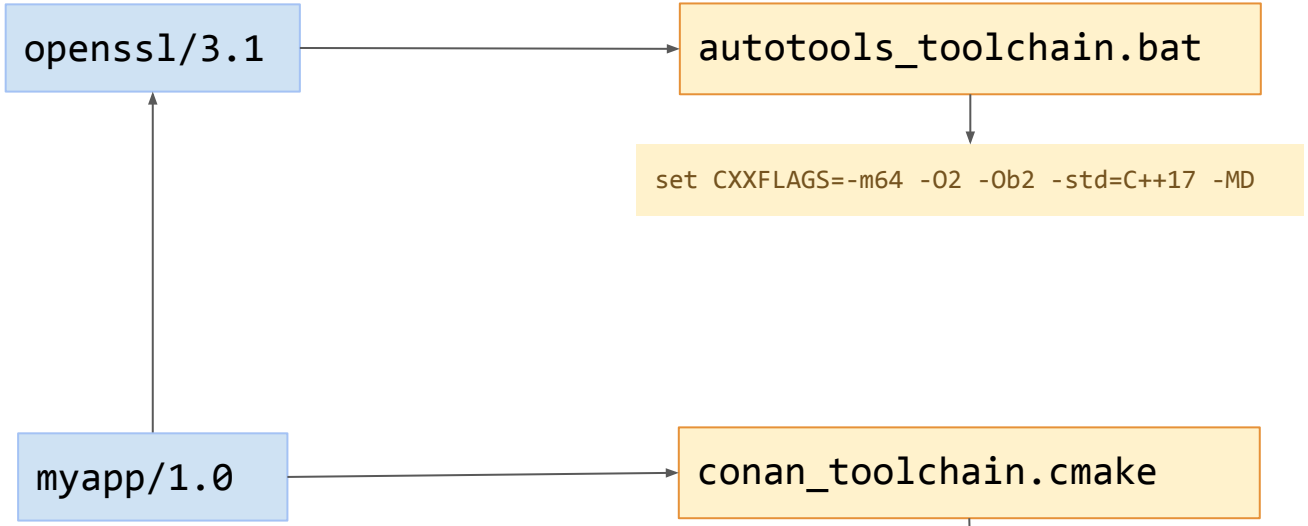
```
include("${CMAKE_CURRENT_LIST_DIR}/../../mytoolchain.cmake")

set(CMAKE_GENERATOR_PLATFORM "x64" CACHE STRING "" FORCE)
set(CMAKE_GENERATOR_TOOLSET "v143" CACHE STRING "" FORCE)

set(CMAKE_MSVC_RUNTIME_LIBRARY "$<$<CONFIG:Release>:MultiThreadedDLL>")

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_EXTENSIONS OFF)

string(APPEND CONAN_CXX_FLAGS " /MP8")
string(APPEND CONAN_CXX_FLAGS " /O1")

set(CMAKE_FIND_PACKAGE_PREFER_CONFIG ON)

list(PREPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR})
list(PREPEND CMAKE_PREFIX_PATH ${CMAKE_CURRENT_LIST_DIR} )
```

Why the toolchain is important for binary compatibility




```
$ conan create . --build=missing -s compiler=msvc -s
compiler.version=193 -s compiler.cppstd=17 -s
compiler.runtime=dynamic
```

Why the toolchain is important for binary compatibility




```
$ conan create . --build=missing -s compiler=msvc -s compiler.version=193 -s compiler.cppstd=17 -s compiler.runtime=dynamic
```

Configuring CMakeToolchain

- **tools.cmake.cmaketoolchain:toolchain_file**
- **tools.cmake.cmaketoolchain:user_toolchain**
- tools.android:ndk_path value
- tools.android:cmake_legacy_toolchain
- tools.cmake.cmaketoolchain:system_name
- tools.cmake.cmaketoolchain:system_version
- tools.cmake.cmaketoolchain:system_processor
- tools.cmake.cmaketoolchain:toolset_arch
- tools.cmake.cmake_layout:build_folder_vars
- tools.build:cxxflags
- tools.build:cflags
- tools.build:sharedlinkflags
- tools.build:exelinkflags
- tools.build:defines list of preprocessor definitions that will be used by add_definitions().
- tools.build:tools.apple:enable_bitcode
- tools.build:tools.apple:enable_arc
- tools.build:tools.apple:enable_visibility
- tools.build:sysroot
- tools.build:compiler_executables

How to use the toolchain

```
$ conan install . -pr=myprofile
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
  -DCMAKE_BUILD_TYPE=Release
```

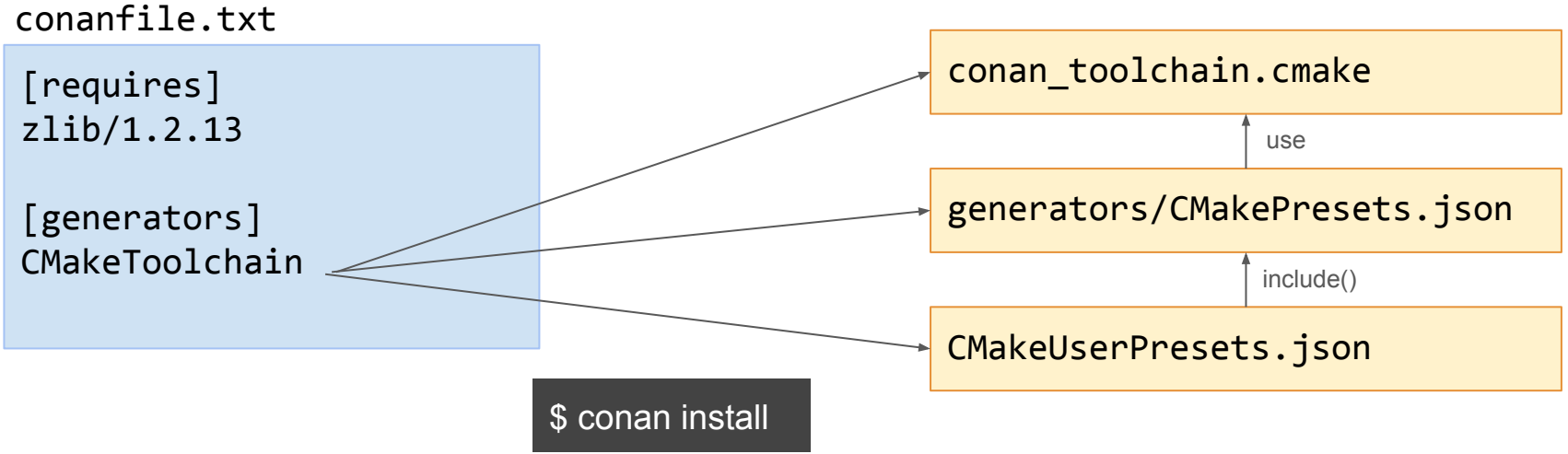
Welcome CMake presets

- Introduced in CMake 3.19, better CMake 3.24 for prod
- CMakePresets.json (or CMakeUserPresets.json) at the **root** of the project

```
$ cmake --preset default
```

```
"version": 3,
"cmakeMinimumRequired": { "major": 3,...},
"configurePresets": [
   {
      "name": "default",
      "displayName": "'default VS' config",
      "description": "my VS config",
      "generator": "Visual Studio 17 2022",
      "cacheVariables": { # same as -DMYVAR=
          "MYVAR": "MYVALUE"
      },
      "binaryDir": "/path/to/my/bin/dir"
   }
 ],
 "buildPresets": [
   {
      "name": "release",
      "configurePreset": "default",
      "configuration": "Release"
   }
 ],
```

CMakePresets.json generation



Conan 1.47.0 (Mar-2022)

Conan generated CMakePresets.json

conanfile.txt

```
[requires]
zlib/1.2.13

[generators]
CMakeToolchain
```

$ conan install

CMakePresets.json

```
"version": 3,
"cmakeMinimumRequired": {"major": 3},
"configurePresets": [
  {
     "name": "conan-default",
     "displayName": "'conan-default' config",
     "generator": "Visual Studio 17 2022",
     "cacheVariables": {
         "CMAKE_POLICY_DEFAULT_CMP0091": "NEW"
     },
     "toolchainFile": "<path>/conan_toolchain.cmake",
     "binaryDir": "path/test"
  }
],
"buildPresets": [
  {
     "name": "conan-release",
     "configurePreset": "conan-default",
     "configuration": "Release"
  }
],
```

Demo

Problems

- CMakeDeps generated targets artificial
- Lack of information for Windows import libraries for shared libraries: missing IMPORTED LOCATION
- Not ergonomic usage of build_modules
- It is not “cmake”! We liked “cmake”!

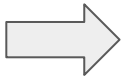
CMakeLists.txt

```
project(PackageTest CXX)

include(.../conanbuildinfo.cmake)
conan_basic_setup()

add_executable(app example.cpp)
target_link_libraries(app ${CONAN_LIBS})
```

cmake-conan: calling Conan from CMake (2.0)

“Canonical” flow

```
$ conan install …
$ cmake …
```



“cmake-conan” flow

```
$ cmake … # calls conan install internally
```

Meet CMake 3.24 “dependency providers”

CMakeLists.txt

```
project(helloworld)

find_package(ZLIB)
```

intercepts

setup.cmake

```
cmake_language(
  SET_DEPENDENCY_PROVIDER provider_method
  SUPPORTED_METHODS FIND_PACKAGE
)

macro(provider_method method package_name)
   ...
endmacro()
```

```
$ cmake -DCMAKE_PROJECT_TOP_LEVEL_INCLUDES=.../setup.cmake
```

cmake-conan 2.0 dependency provider

Call “install” only once

Check Conan installed

Map CMake config to Conan

Install deps

Call actual find_package()

```
macro(conan_provide_dependency method package_name)
    set_property(GLOBAL PROPERTY CONAN_PROVIDE_DEPENDENCY_INVOKED TRUE)
    get_property(CONAN_INSTALL_SUCCESS GLOBAL PROPERTY)
    if(NOT CONAN_INSTALL_SUCCESS)
        find_program(CONAN_COMMAND "conan" REQUIRED)
        conan_get_version(${CONAN_COMMAND} CONAN_CURRENT_VERSION)
        conan_version_check(MINIMUM ${CONAN_MINIMUM_VERSION} CURRENT ..)

        conan_profile_detect_default()
        detect_host_profile(${CMAKE_BINARY_DIR}/conan_host_profile)

        conan_install(${_host_profile_flags} ${_build_profile_flags}
                      --build=missing)
    endif()

   find_package(${package_name} ${_find_args} BYPASS_PROVIDER PATHS
                "${CONAN_GENERATORS_FOLDER}")
endmacro()
```

Two different developer flows

Generated toolchain

```
$ conan install . -pr=myprofile
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
```

Generated toolchain via generated presets

```
$ conan install . -pr=myprofile
$ cmake --preset conan-default # -DCMAKE_TOOLCHAIN_FILE=
```

Conan profile ⇒ toolchain ⇒ cmake

cmake-conan integration with provider

```
$ cmake -DCMAKE_PROJECT_TOP_LEVEL_INCLUDES=conan_provider
```

```
$ cmake --preset default # -DCMAKE_PROJECT_TOP_LEVE_INCLUDES
```

CMake config ⇒ profile

CMake and Conan: past, present and **future**.

MeetingC++23

Future

CMakeToolchain:

- Generate Environment variables (avoid conanbuild.bat|.sh script completely) in CMakePresets.json

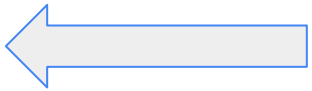
CMakeDeps:

- Full refactor to create better targets, representing static and shared libraries directly (with imported libs location), without artificial package interface targets
- Have something similar to Conan 1 “cmake” generator, that don’t need “find_package”

Common Package Specification (CPS)

...

CPS



ZLib

```
├──include
│       zconf.h
│       zlib.h
│
├──lib
│       zlib.lib
```

- Compiled **binary**
  - Any build system
  - Closed source
- **Single**-configuration
  - Valid for the current build
- **No version** information
- **No ABI** information

Related work

- CPS Matthew Woehlke et al https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1313r0.html
- ISO C++: https://github.com/isocpp/pkg-fmt
- Libman https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/libman/develop/data/spec.bs by Colby Pike (@vectorofbool)

Existing solutions

```
prefix=@CMAKE_INSTALL_PREFIX@
exec_prefix=@CMAKE_INSTALL_PREFIX@
libdir=@INSTALL_LIB_DIR@
sharedlibdir=@INSTALL_LIB_DIR@
includedir=@INSTALL_INC_DIR@

Name: zlib
Description: zlib compression library
Version: @VERSION@

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

```
set(_ZLIB_x86 "(x86)")
set(_ZLIB_SEARCH_NORMAL PATHS
"[HKEY_LOCAL_MACHINE\\SOFTWARE\\GnuWin32\\Zlib;InstallPath]"
list(APPEND _ZLIB_SEARCHES _ZLIB_SEARCH_NORMAL)

if(ZLIB_USE_STATIC_LIBS)
  set(ZLIB_NAMES zlibstatic zlibstat zlib z)
  set(ZLIB_NAMES_DEBUG zlibstaticd zlibstatd zlibd zd)
else()
  set(ZLIB_NAMES z zlib zdll zlib1 zlibstatic zlibwapi ..)
  set(ZLIB_NAMES_DEBUG zd zlibd zdlld zlibd1 zlib1d ..)
endif()

if(ZLIB_FOUND)
    set(ZLIB_INCLUDE_DIRS ${ZLIB_INCLUDE_DIR})

    if(NOT TARGET ZLIB::ZLIB)
      add_library(ZLIB::ZLIB UNKNOWN IMPORTED)
      set_target_properties(ZLIB::ZLIB PROPERTIES
                            INTERFACE_INCLUDE_DIRECTORIES
                           "${ZLIB_INCLUDE_DIRS}")
```

Searching for Convergence in C++ Package Management - Bret Brown & Daniel Ruoso - CppNow 2022

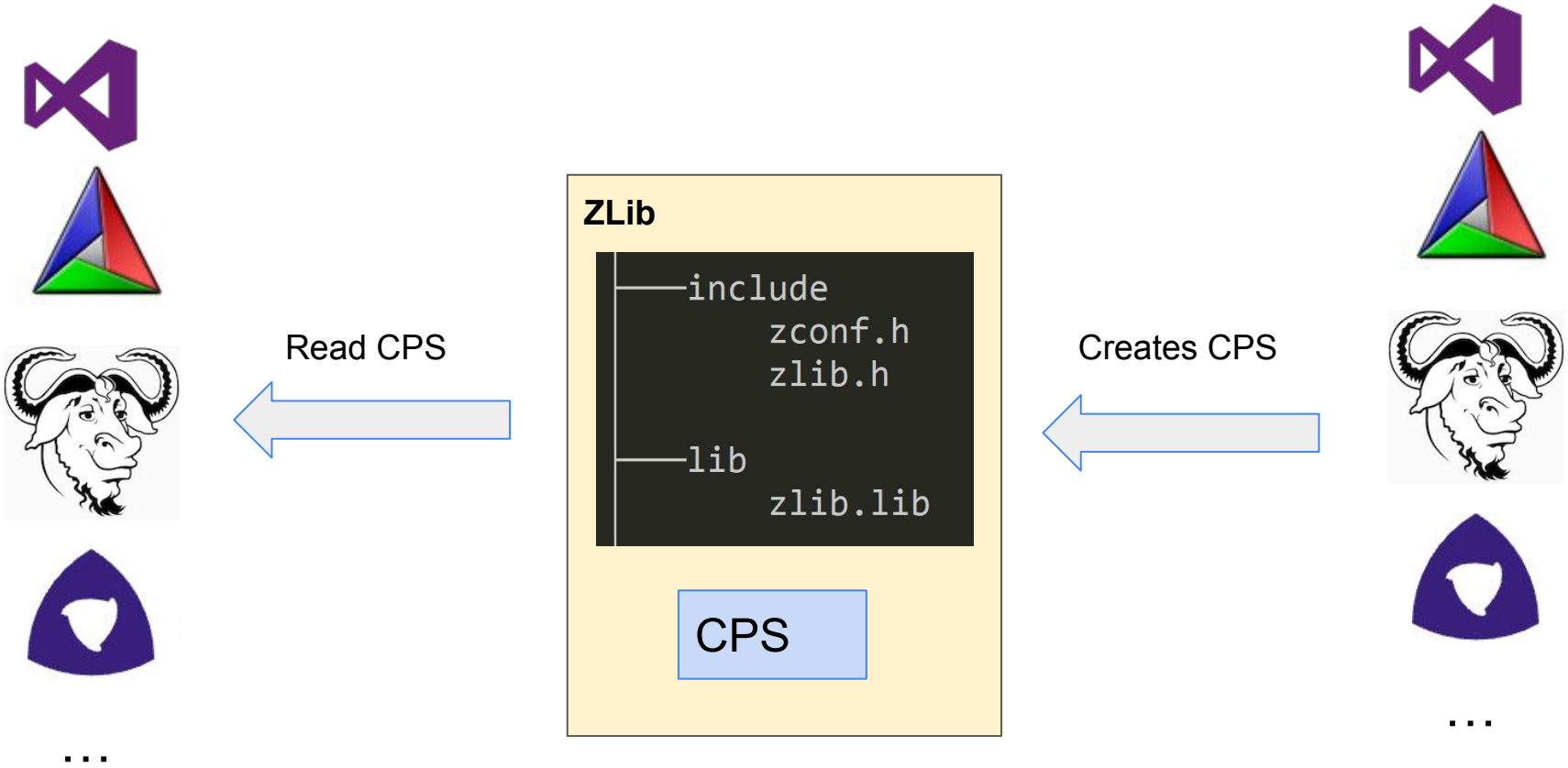
Case For a Standardized Package Description Format for External C++ Libraries by Luis Caro Campos - CppCon22

Existing solutions

conanfile.py

```
class ZLibConan(ConanFile):
    name = "zlib"
    version = "1.2.13"
    ...

    def package_info(self):
        self.cpp_info.includedirs = ["include"]
        self.cpp_info.libdirs = ["lib"]
        if self.settings.os == "Windows":
            libname = "zdll" if self.options.shared
                    else "zlib"
        else:
            libname = "z"
        self.cpp_info.libs = [libname]
```

Interoperability



ZLib

```
|--include
|    zconf.h
|    zlib.h
|
|--lib
|    zlib.lib
|
|--licenses
     LICENSE
```

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

* Just an instance, for Windows MSVC, static library

Include Directories

- Maps to `-I<folder>` or to `-Isystem<folder>`?
- Why it is a list of folders?
- Conventions?

zlib.cps

```
{
  "includedirs": ["include"],
  "libdirs": ["lib"],
  "libs": ["zlib"],
  "properties": {
    "cmake_find_mode": "both",
    "cmake_file_name": "ZLIB",
    "cmake_target_name": "ZLIB::ZLIB",
  }
}
```

Relative paths by default: package “relocatibility”

...

```
zlib_win
|--include
|    zconf.h
|    zlib.h
|
|--lib
|    zlib.lib
|--zlib.cps
```

$ mv zlib zlib_win

```
zlib
|--include
|    zconf.h
|    zlib.h
|
|--lib
|    zlib.lib
|--zlib.cps
```

Relative paths? “System” packages

- In non-standard (absolute) locations

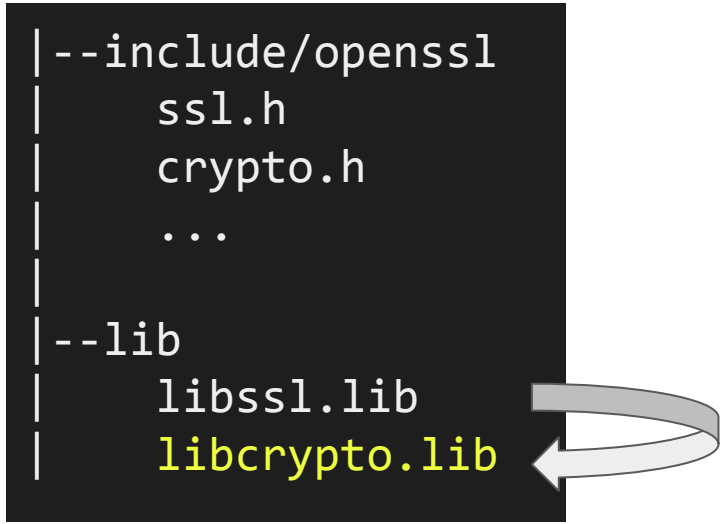
mypkg.cps

```
{
  "includedirs": ["/usr/nonstandardpath/headers/mylib/include"],
  "libdirs": ["/usr/othernonstandardpath/libs/mylib/lib"]
  "libs": ["mylib2", "mylib1"]
}
```

Litmus test:

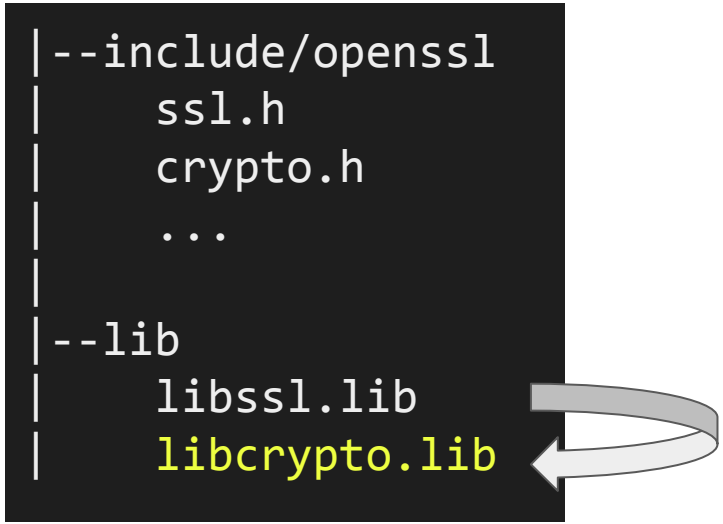
- Can it be the output of a “build/install” process?
- Can it be consumed by build systems?

OpenSSL



What “linking openssl” means?

- By default, most users want to use and link with “ssl” library
  - Link “crypto” transitively
- But, some users will want to use only “crypto”
  - What if we pass “-lssl -lcrypto”?
  - Some linkers can be smart and optimize away
  - Some linkers (embedded cross-toolchains) will not
    - Larger than necessary binaries



OpenSSL: Components

openssl.cps

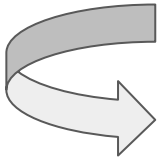
```
"root": {
   "properties": {"cmake_file_name": "OpenSSL"}
},
"ssl": {
  "includedirs": ["include"],
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  "includedirs": ["include"],
  "system_libs": ["crypt32", "ws2_32", "advapi32",...],
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

OpenSSL: Requirements

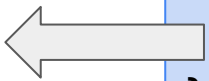
openssl.cps

```
"root": {
   "properties": {"cmake_file_name": "OpenSSL"}
},
"ssl": {
  ...
  "libs": ["libssl"],
  "requires": ["crypto"],
},
"crypto": {
  ...
  "system_libs": ["crypt32", "ws2_32", "advapi32",...],
  "libs": ["libcrypto"],
  "requires": ["zlib::zlib"],
}
```

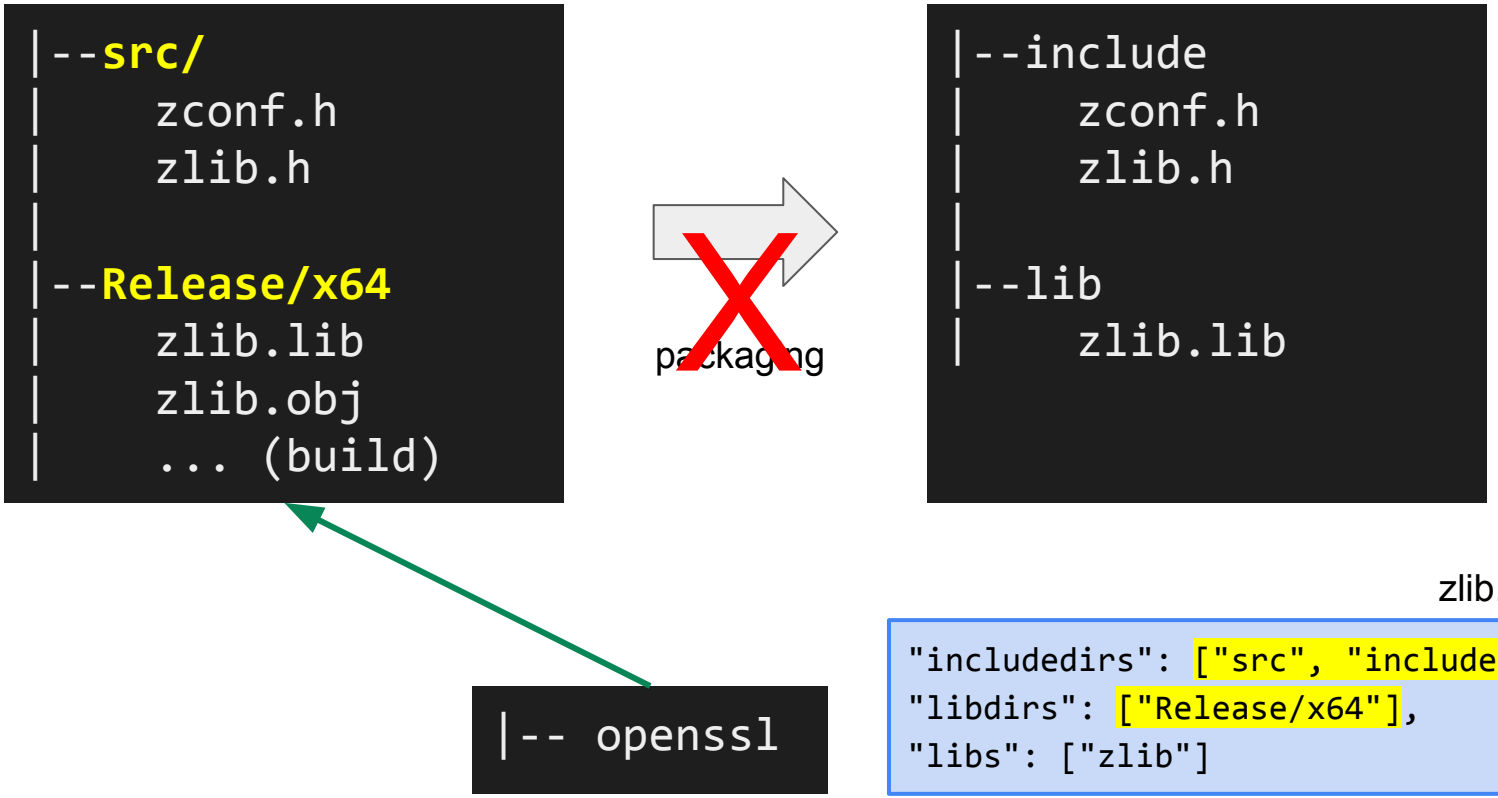
internal



external



“Editable” packages



Ongoing work

```
cmake_minimum_required(VERSION 3.27)

project(cpstest CXX)

load_package(${CPS_ROOT}/fmt.cps ${FMT_PREFIX})
load_package(${CPS_ROOT}/spdlog.cps ${SPDLOG_PREFIX})

add_executable(demo demo.cpp)
target_link_libraries(demo spdlog::spdlog)
```

*CppCon 2023 Bret Brown and Bill Hoffman: A First Step Toward Standard C++ Dependency Management

Ongoing work

```
def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "both")
    self.cpp_info.set_property("cmake_file_name", "ZLIB")
    self.cpp_info.set_property("cmake_target_name", "ZLIB::ZLIB")
    if self.settings.os == "Windows" and not self._is_mingw:
        libname = "zdll" if self.options.shared else "zlib"
    else:
        libname = "z"
    self.cpp_info.libs = [libname]
    import json
    print(json.dumps(self.cpp_info.serialize(), indent=2))
```

```
{
  ...
  "libs": ["zlib"],
}
```

* CppCon23. D. Rodriguez-Losada. A Common Package Specification: Getting Build Tools to Talk to Each Other: Lessons Learned From Making Thousands of Binaries Consumable by Any Build System

Ongoing work

- CppLang **#ecosystem** slack channel
- CPS repo: https://cps-org.github.io/cps
- Ecosystem mailing list: https://groups.google.com/g/cxx-ecosystem-evolution/about
- Monthly meetings

Conclusions

- CMake and Conan integration is live and evolving.
  - Based on users feedback
- CMake and Conan in 2023 can provide a fully transparent integration
  - Even commanded from ``cmake …`` command
  - Can manage the installation of tons of dependencies in seconds or few minutes, same flow in all OSs
  - Using a C, C++ package manager in 2023 has more advantages than disadvantages
- Future is looking even better with CPS standardization work
  - We are actively working in it

Thanks! Questions?