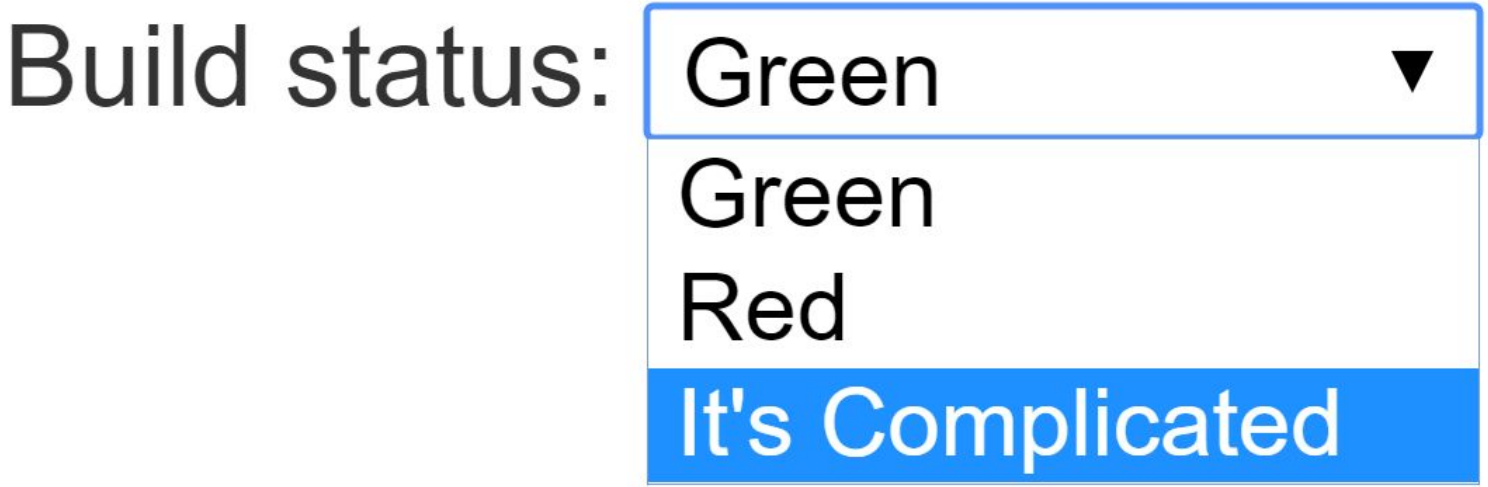
Build for Everyday Programmers

(and Why Should They Care)


Build status:
Green
Green
Red
It's Complicated

What if build wasn’t hard?

Hello!

I am Mathieu Ropert

I'm a C++ developer at Paradox Development Studio where I make Europa Universalis.

You can reach me at:

 mro@puchiko.net

 @MatRopert

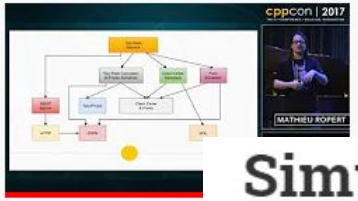
 https://mropert.github.io

About this talk

- Programmers and build
- Overview of “modern” C++ builds
- Toolchains, package managers and modern CMake
- Simplicity

1 Let’s talk about build

Or why it concerns us programmers



CppCon 2017: Mathieu Ropert “Using Modern CMake Patterns to Enforce a Good Modular Design”

CppCon • 34K views • 2 years ago

http://CppCon.org — Presentation Slides, PDFs, Source Code and other presenter materials are available

Simplifying build in C++ (part 1)

19 Oct 2017 on **Build systems**, **C++**

Package management and build systems are one of the next big challenges C++ is going to face. In this article series, I offer some thoughts and ideas to solve that. This post is the first part, where I give an overview and try to tackle the first issue: interaction between package managers and build systems.

http://CppCon.org — Discussion & Comments: https://www.reddit.com/r/cpp/ — Presentation Slides, PDFs, Source Code and other ...

Confession time

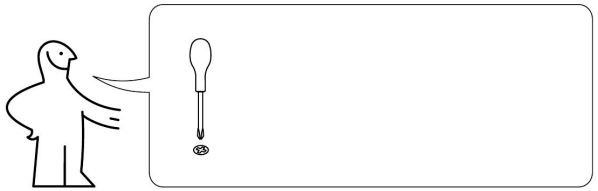
Confession time

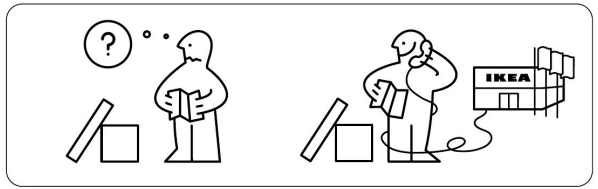
- I’m not a build engineer
- Most of my daily work is spent programming
- I had no idea how CMake worked 5 years ago
- And yet here I am…

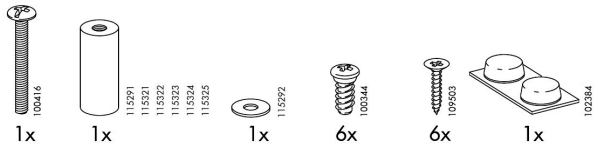
A *build metaphor*

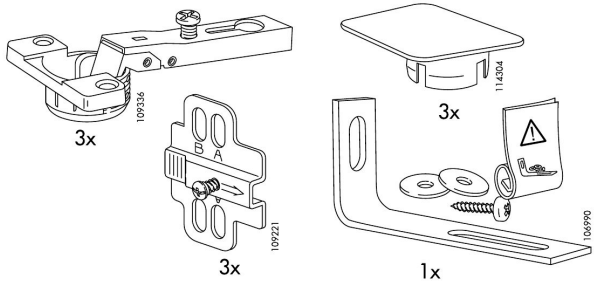
A *build metaphor*

A *build metaphor*



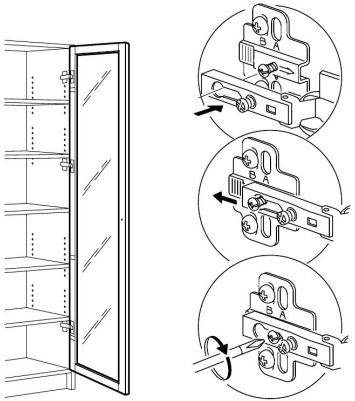




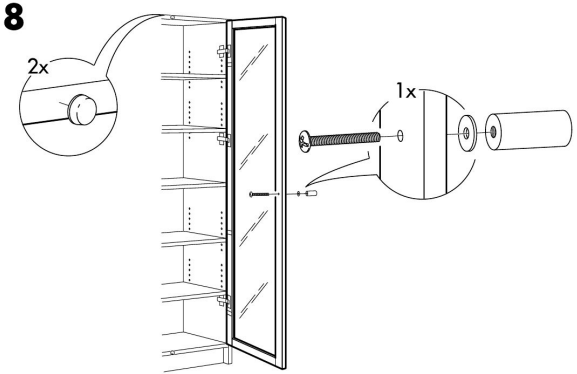


AA-169268-3

7

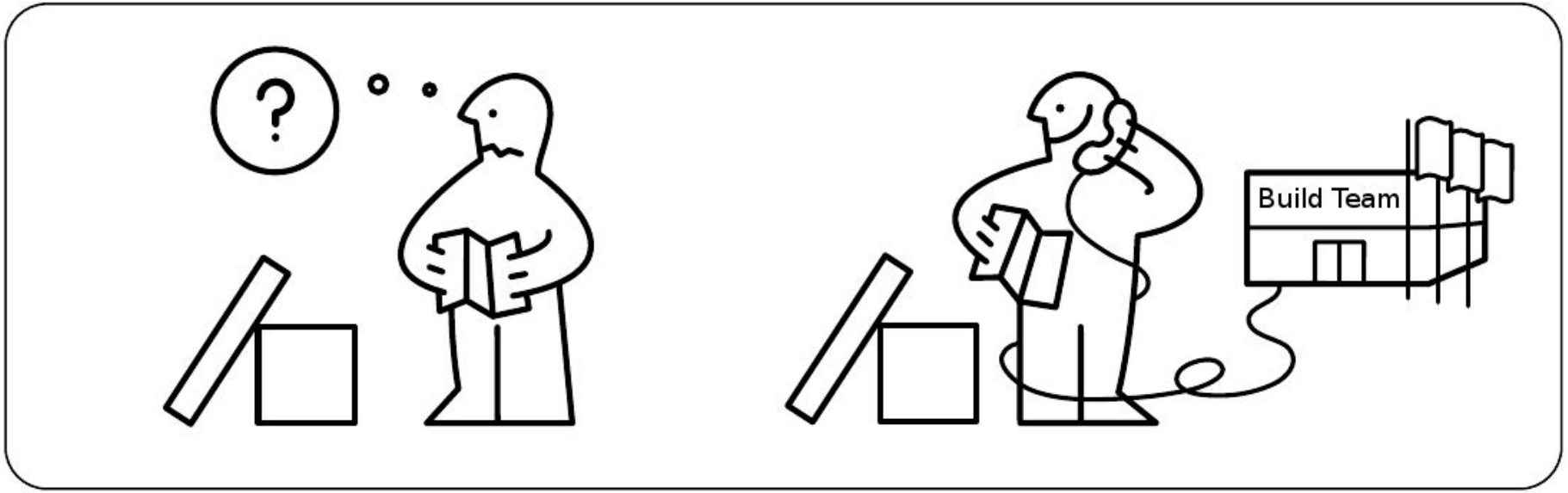


8



A build metaphor

A *build metaphor*



A *build metaphor*

A build metaphor

- Without build, software is just a bunch of loose source files
- Header-only-no-dependencies approach is the wrong way of solving the problem
- Programmers should take ownership of their build

Who takes care of the build

- Let’s ask ourselves: who makes the most changes to build files in a project?
- The build engineer
- Programmers who keep adding/removing sources and dependencies

Who takes care of the build

- Build reflects relationships between the various components in a project
- Ownership of software architecture implies ownership of the build

Build has changed

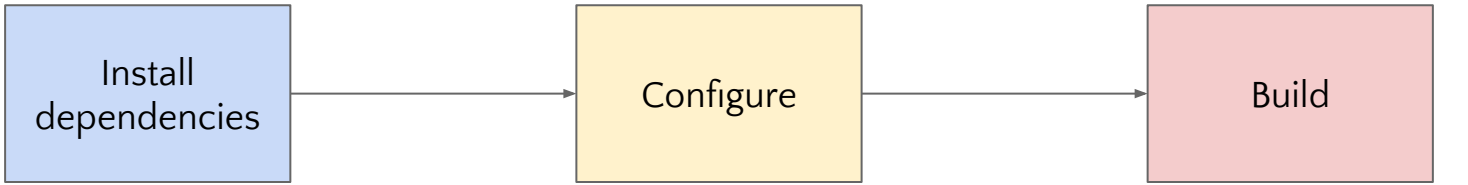
- Build tools have made tremendous progress in the past years
- CMake got more “Modern”
- Two good package managers are available on major platforms

Build has changed

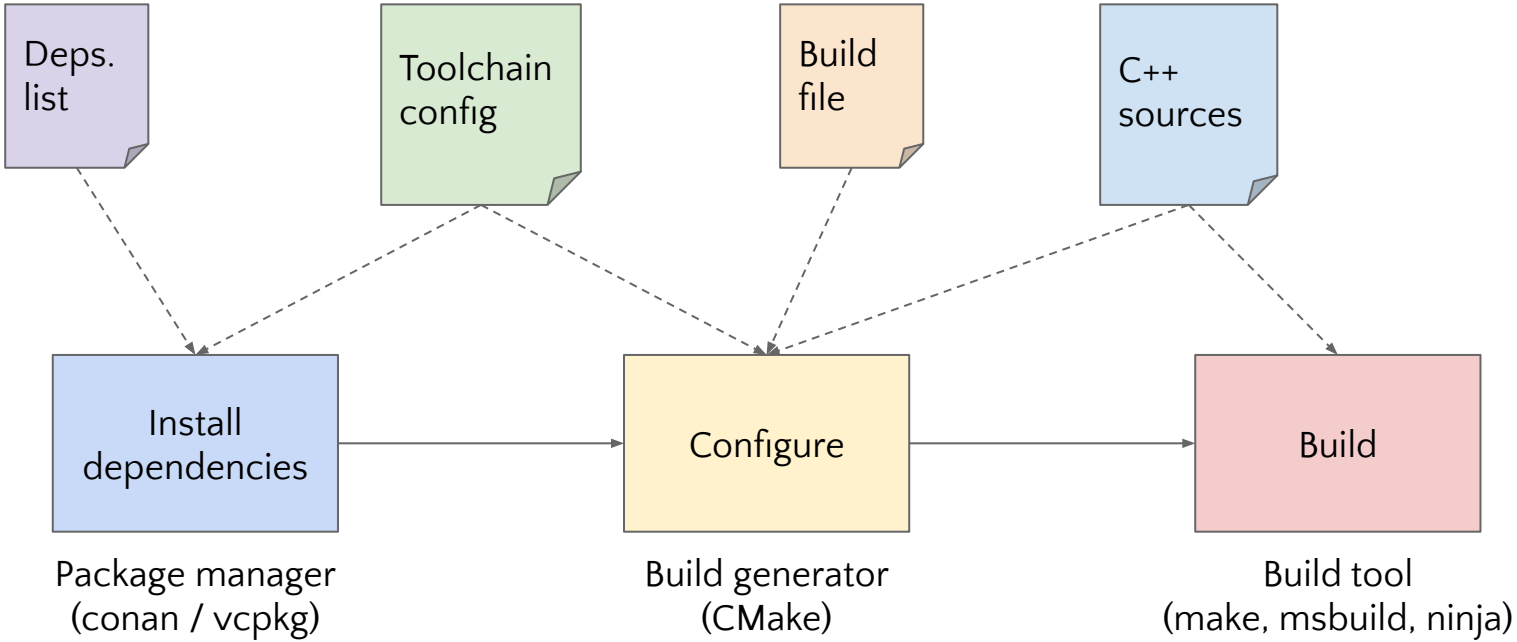
- Modern builds are much simpler to maintain
- They make it easy to add dependencies, migrate compiler and define new targets
- Cost of transition from legacy build isn’t as high as it may seem

2 Anatomy of a build

An overview of a “modern” C++ builds



Build workflow



Build workflow

Package manager

- Fetches dependencies required for the build
- Ensures binaries are compatible with toolchain
- May either download sources and build or reuse available pre-builts

Build generator

- Checks that all build pre-requisites are met
- Should fail if something is wrong
- Writes the actual build script (make, vcproj, ninja...)

Build tool

- Runs the actual compilation/linking commands
- Handles incremental rebuilds
- Restarts the whole process if build script has changed (*)

Modern build workflow

Then

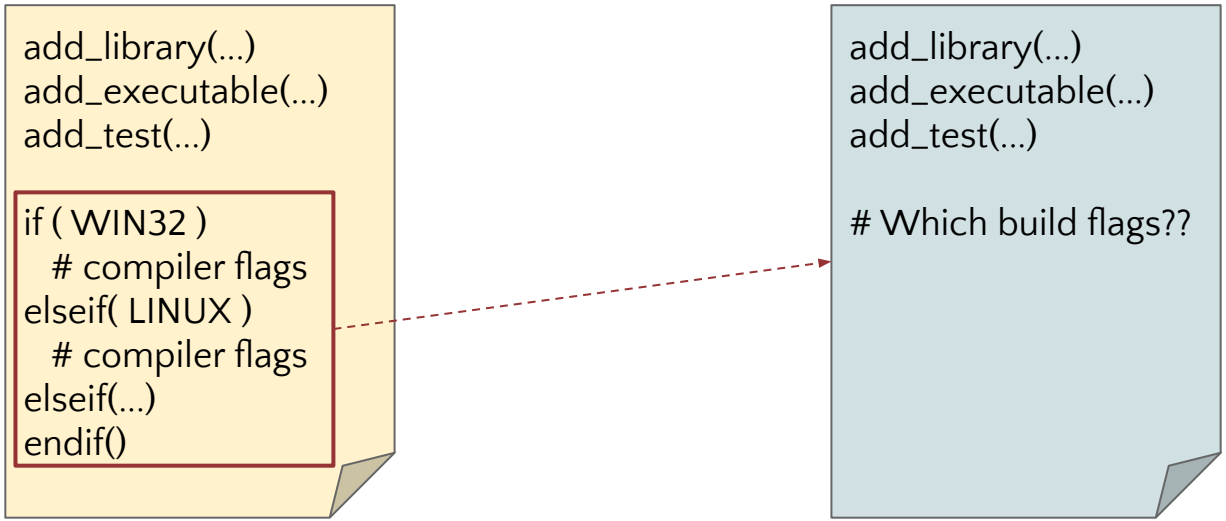
- cmake -G ...
- make / ninja / IDE

Now

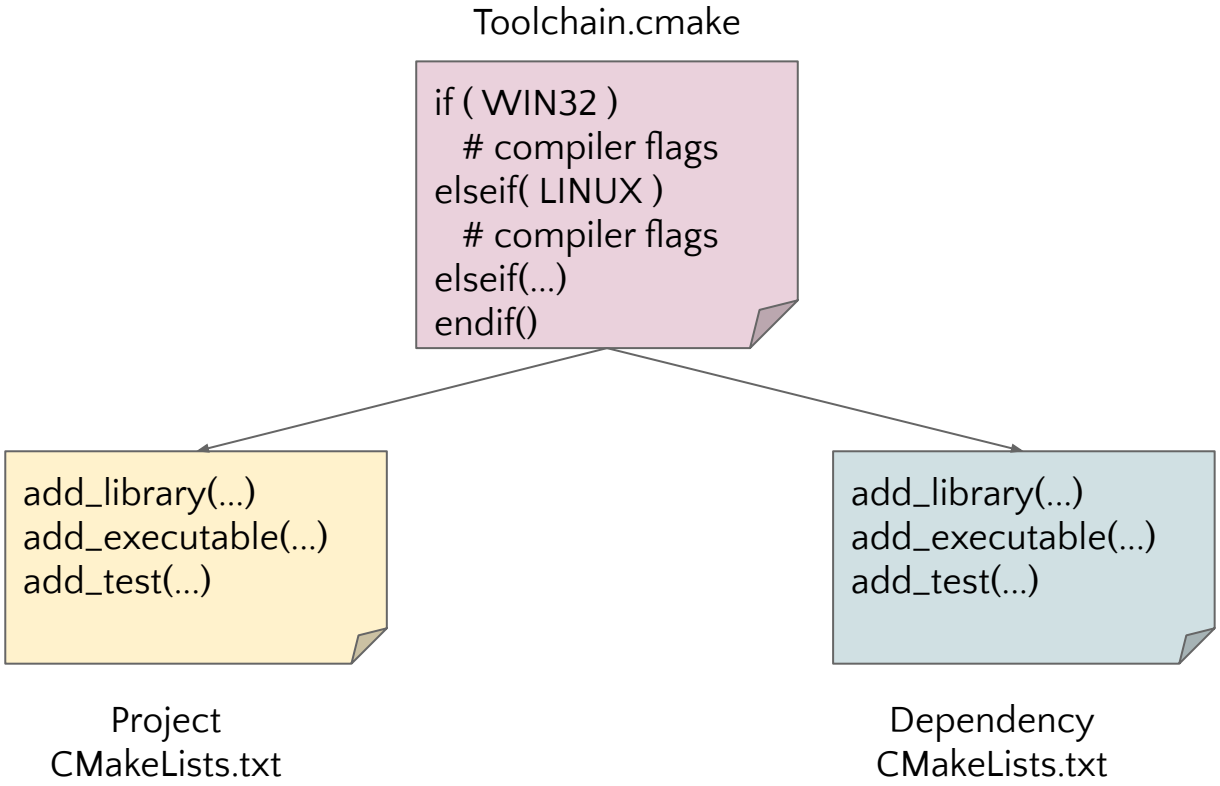
- conan / vcpkg
- cmake -DMAGIC=XXX -G ...
- make / ninja / IDE

3 Toolchain definitions

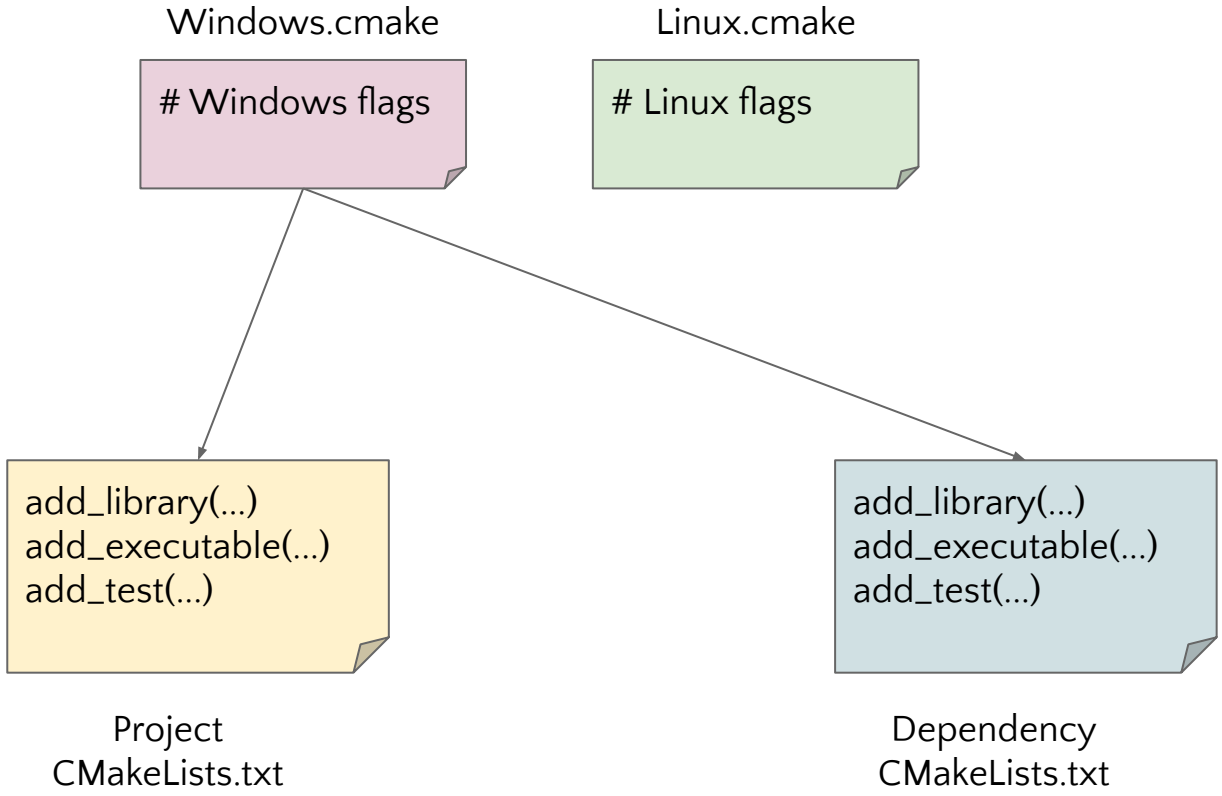
How to split concerns



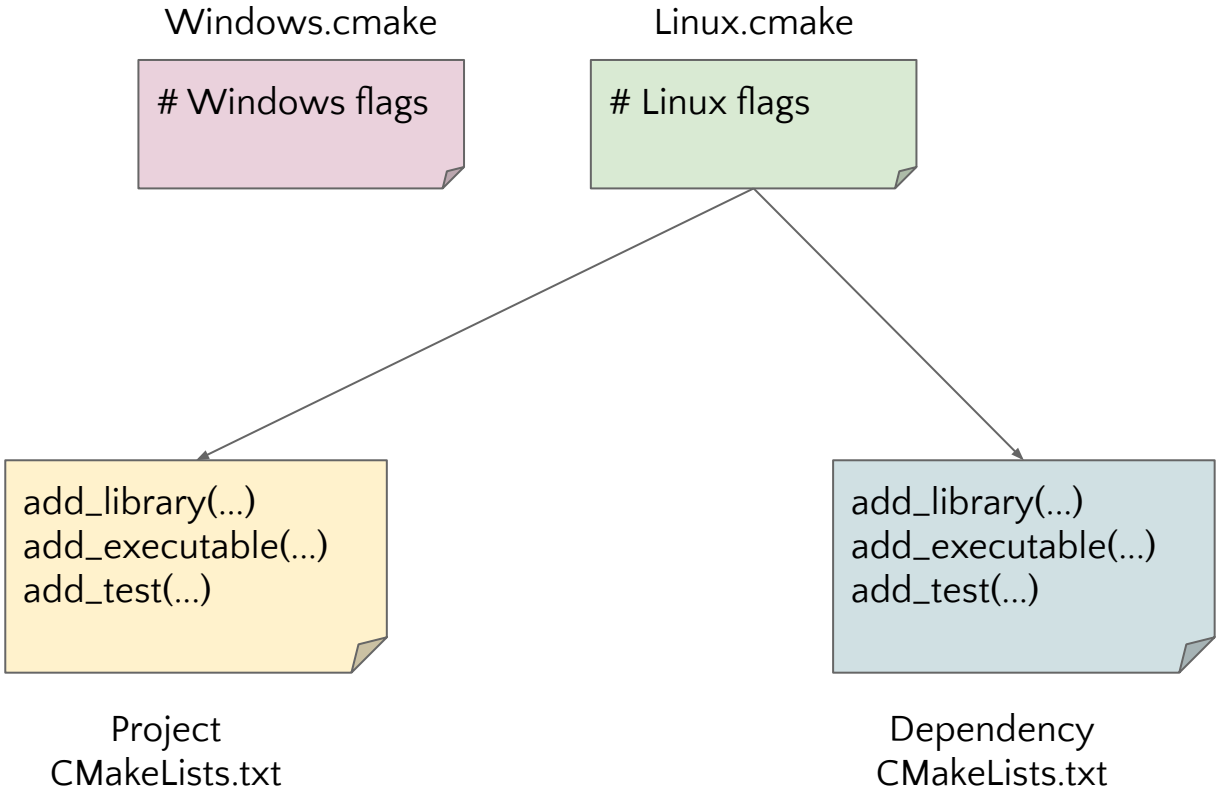
Toolchain description



Toolchain description



Toolchain description



Toolchain description

Toolchain description

- Abstract away compiler settings outside of project build files
- Easier to propagate between dependencies
- The default for cross compilation

Toolchain description

- Defines which compiler to use
- Sets the common build flags
  - Architecture (-m64, -march=haswell, ...)
  - Optimization (-O2, /Ob2, ...)
  - Debug info (-g, /Zi, ...)

Toolchain description

- May define one or more build targets
- For make/ninja targets, one toolchain descriptor per target
- For MSVC, all targets are likely to be defined in the same toolchain descriptor for ease of use

Toolchain usage

- `cmake -DCMAKE_TOOLCHAIN_FILE=x.cmake`
- Vcpkg: `VCPKG_CHAINLOAD_TOOLCHAIN_FILE`
- Conan uses “profiles” config files that can be translated to CMake and other build systems

Where settings live

Toolchain file

- Compiler, linker, assembler…
- Architecture and ABI flags
- Optimization flags

Project build

- Warnings
- Can’t think of another one

4 Package managers

One of them is a cool barbarian

Installing packages 101

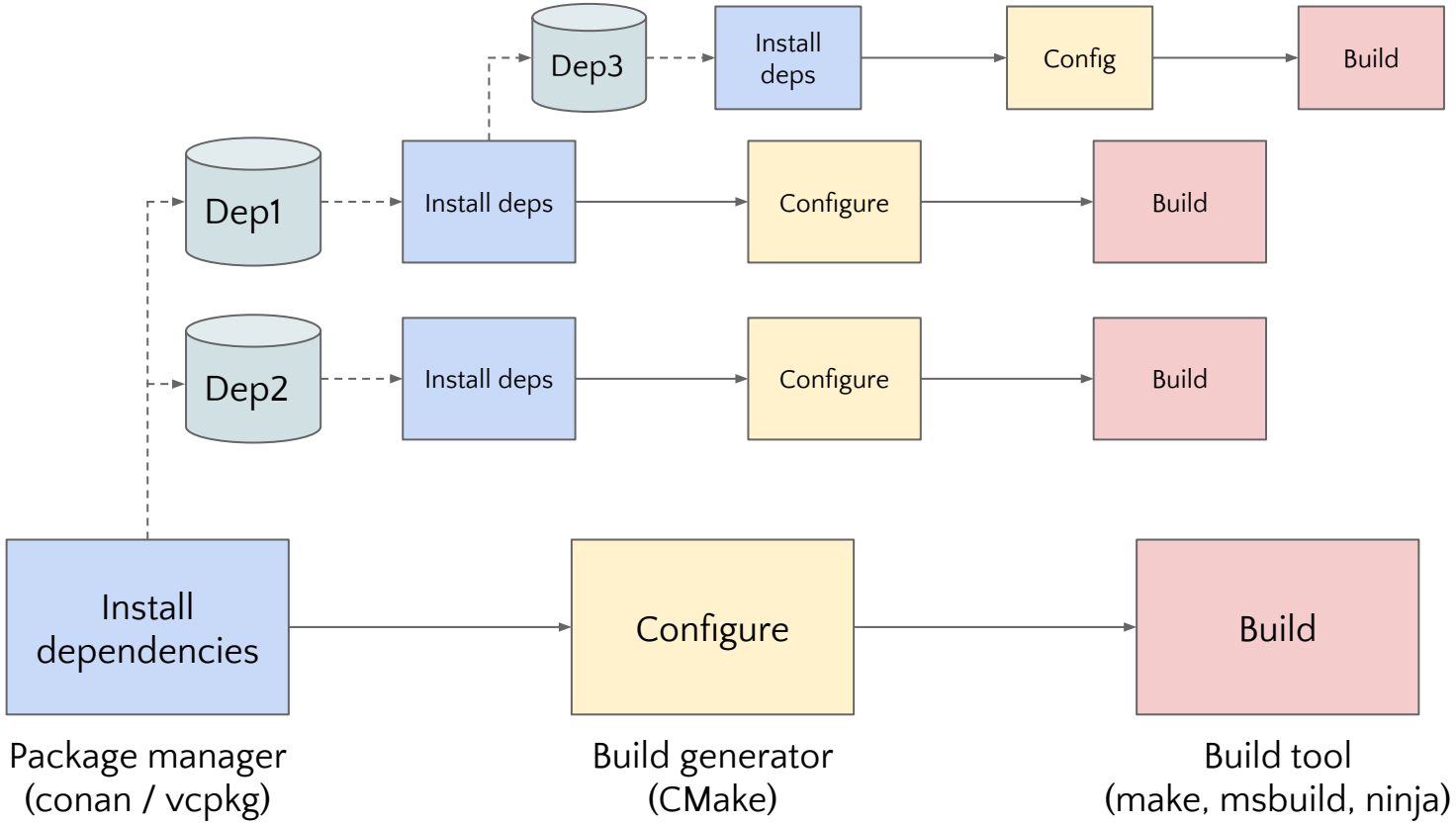
- Install dependencies
- Download sources
- (Patch)
- Configure / Build
- Copy to install directory

Installing packages 101

- Install dependencies
- Download binaries
- Copy to install directory

Package manager jobs

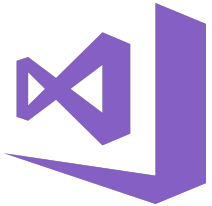
- Install dependencies recursively
- Make them available for the build system
- Make our own project available to others



Installing deps

Which package manager?

- Business focused
- Less packaged libraries
- Artifactory / Bintray integration



- Open source focused
- Lots of packaged libraries
- Harder to setup for internal usage

Conan

```
$ conan install ../

$ cmake ../ -DCMAKE_TOOLCHAIN_FILE=conan_paths.cmake
```

vcpkg

```
$ vcpkg install googletest

$ cmake ../ -DCMAKE_TOOLCHAIN_FILE=/.../vcpkg.cmake
```

Describing dependencies

- With vcpkg a simple text file with a list of dependencies can be piped to vcpkg install
- With Conan either a short config manifest or a python script
- Conan python option make your project a reusable package that can be published

The ultimate showdown

- If you quickly want to try out a new 3rd party, vcpkg is your best option
- For education and personal projects, vcpkg is also recommended
- Conan really shines in corporate environments

5 Build files

Who loves CMakeLists.txt?

Quick recap

- With toolchains we moved away compiler definition and build flags
- With package managers we extracted dependency build and install
- What should remain in our CMakeLists.txt?

Not *much* 🤷‍♀️

Simplicity

- Build files should be simple and stupid
- A lot of existing complexity is a workaround to a lack of toolchain or package manager
- Most content should be about describing the project sources architecture

Simplicity

- Define libraries and executables
- Define dependencies between them
- Define tests
- Done!

CMake Simplicity

- `cmake_minimum_required()`
- `project()`
- `find_package()`
- `enable_testing()`

CMake Simplicity

- `add_library()`
- `add_executable()`
- `add_test()`

CMake Simplicity

- `target_include_directories()`
- `target_add_definitions()`
- `target_link_libraries()`

Extra best practices

- Check and fail, do not attempt to fix build environment from within
- Adding warning flags is perfectly OK, other flags should be treated with suspicion
- Script constructs are likely signs of a need for abstraction

6 Wrapping up

Recap and suggestions for further study

In conclusion

- A project build should be its maintainer's concerns
- Build doesn't have to be hard, use toolchains and package managers
- Keep your CMakeLists dumb so you can dump them when the next best thing finally comes

Do you want to know more?

- *Using Modern CMake Patterns to Enforce a Good Modular Design*, CppCon 2017
- *The State of Package Management in C++*, ACCU 2019
- *Unannounced Toolchains Talk*, ???

Furthermore, I think your build should be destroyed

Thanks!

Any questions ?

You can reach me at

- mro@puchiko.net
- @MatRopert
- @mropert
- https://mropert.github.io