# The State of Package Management in C++

*“We need a better package/build system”*

*Bjarne Stroustrup*

*CppCon 2017*

## What we want

- Bjarne pitched a simple workflow
  - `> download gui_xyz`
  - `> install gui_xyz`
- Done!
- Then you can just write(*)
  - `import gui_xyz;`

(*) When Modules are adopted

## What do we have today?

## Hello!

### *I am Mathieu Ropert*

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

You can reach me at:

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

## About this talk

- Why package management?
- Today’s packager managers for C++
- Making your library packageable
- Looking at the future

1

## Why package management?

The great challenge to come

## Getting stuff done

- ISO C++17 Standard Library currently offers:
  - File I/O
  - Filesystem operations
  - Console output
  - Command line arguments
  - System environment variables
- That's it 😢

## Getting stuff done

- Out of the box you can't
  - Access HTTP resources
    - … or any network resource at all
  - Display any 2D or 3D GUI
  - Play sounds
  - Access a SQL (or NoSQL) database
  - Read a well defined format (ZIP, JPEG, JSON…)
  - Handle Unicode

## Getting stuff done?

- Not every software is purely about computation and console UI
- Makes it hard to kickstart development
- Especially harmful to education

## Getting stuff done

- Push it into the standard!
  - Networking TS
  - 2D graphics proposal
  - SG11: Databases
  - SG16: Unicode
- Use a 3rd party library

## 3rd party libraries?

- C++ doesn’t lack in quantity or quality
  - Boost
  - Catch2
  - CURL
  - FFMpeg
  - FreeImage
  - OpenSSL
  - SQLite

## Here’s my new cool library!

- “It’s header-only”
- “It has no dependencies”

## Why do we do this?

A. We don’t trust code made by others while implicitly asking them to trust ours

B. We are afraid that the hassle of package management will drive potential users away

## Why do we do this?

A. We don’t trust code made by others while implicitly asking them to trust ours

B. We are afraid that the hassle of package management will drive potential users away

## Why do we do this?

- Using external libraries has historically been painful in C++
- Dependencies of dependencies quickly turned into a nightmare
- How to redistribute them with the final product?

## Package management

- Leverage on code made by others
- Regardless of the platform or environment
- At a low cost
- Don’t reinvent the wheel!

## Package management

- Not a new topic
  - Unix distributions have been doing it for decades
  - A lot of languages offer a package manager
- But native cross-platform software has always been hard
  - ABI concerns
  - Different compilers and build systems

## Package management

- C++ is more than 30 years old, and sometimes uses even older C software
- Can't suddenly invent a standard and magically port all existing software to it
- Have to work with the existing ecosystem

## Use cases

### Open environment

- Open source development
- Education
- Unlimited number of build configurations

### Close environment

- Private or corporate projects
- Binary distributions
- Manageable number of build configurations

2

## Today’s package managers for C++

Many options, few solutions

## Installing packages 101

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

open source

## Installing packages 101

- Install dependencies
- Download binaries
- Copy to install directory

## Using installed packages

- Depends on your build system
- Quite straightforward for CMake
- Others may or may not be supported
- Fallback to include/lib search path

## A few good choices

- There's a surprisingly large number of attempts at solving the problem
- Featuring different approaches
- Only a handful really stand out

## A few good choices

- Constraint #1: support the 3 majors OS: Linux, OSX and Windows
- Even if not all users target the big 3, there will be a sensible share targeting each
- Eliminates: NuGet, Nix, apt-get, yum, ...

## A few good choices

- Constraint #2: must work with the existing ecosystem
- Do not expect maintainers to switch to a new build system, work with the existing
- Eliminates: Bazel, build2, meson

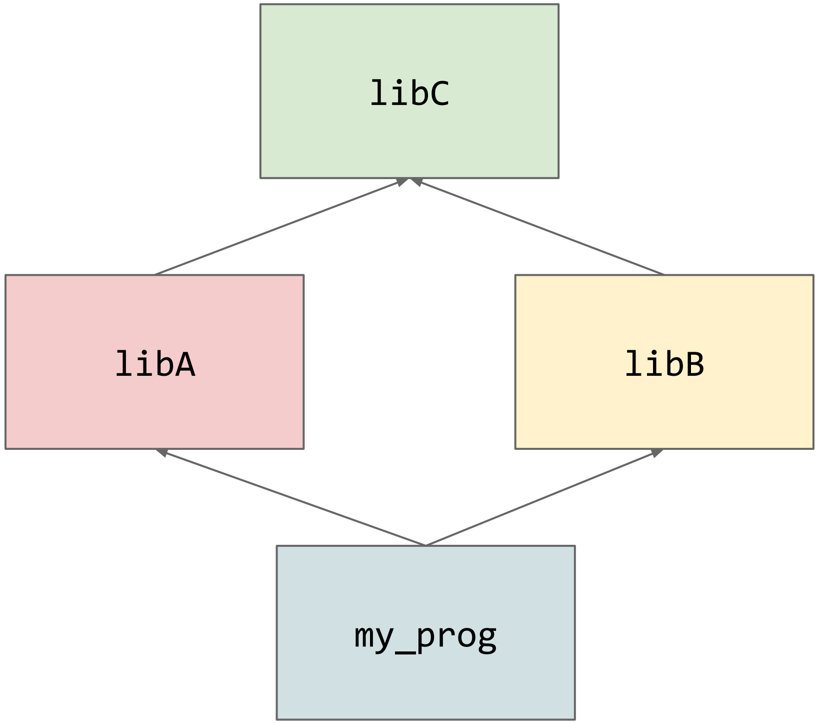
## A few good choices

- Constraint #3: respect encapsulation
- Don’t be intrusive and force package management intrinsics inside build files
- Eliminates: hunter

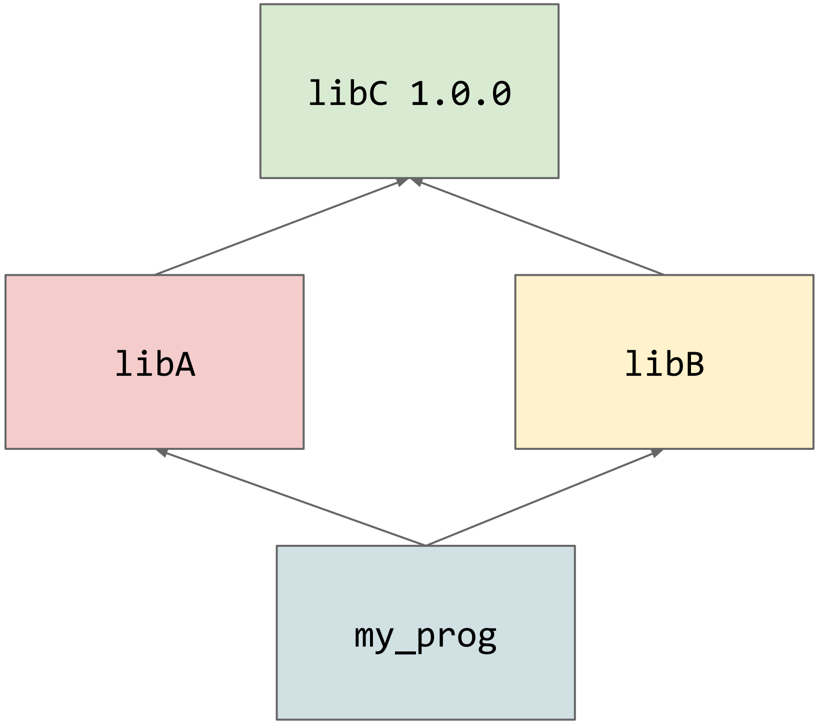
## A few good choices

- Constraint #4: handle the diamond problem

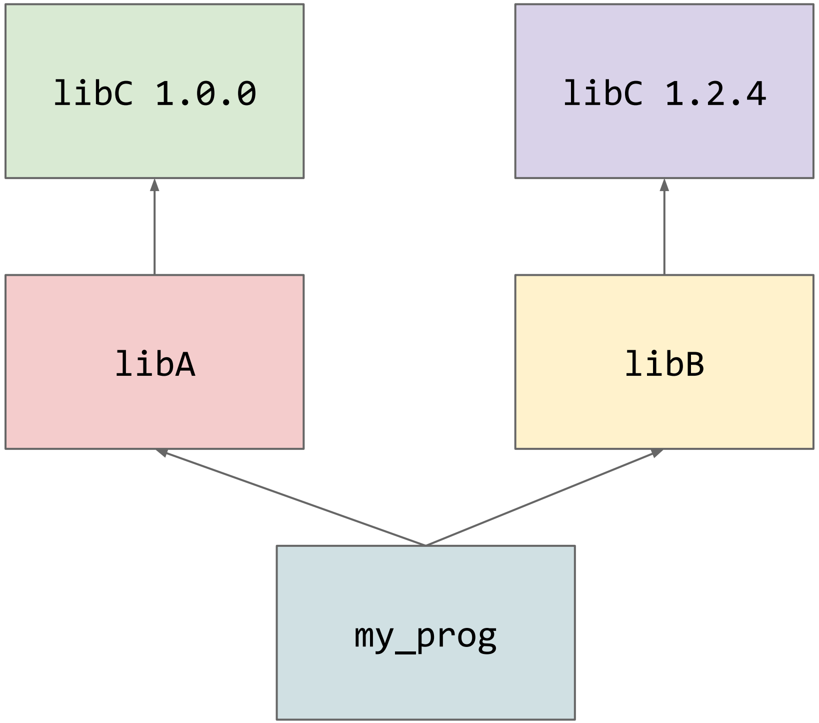
## Diamond problem?



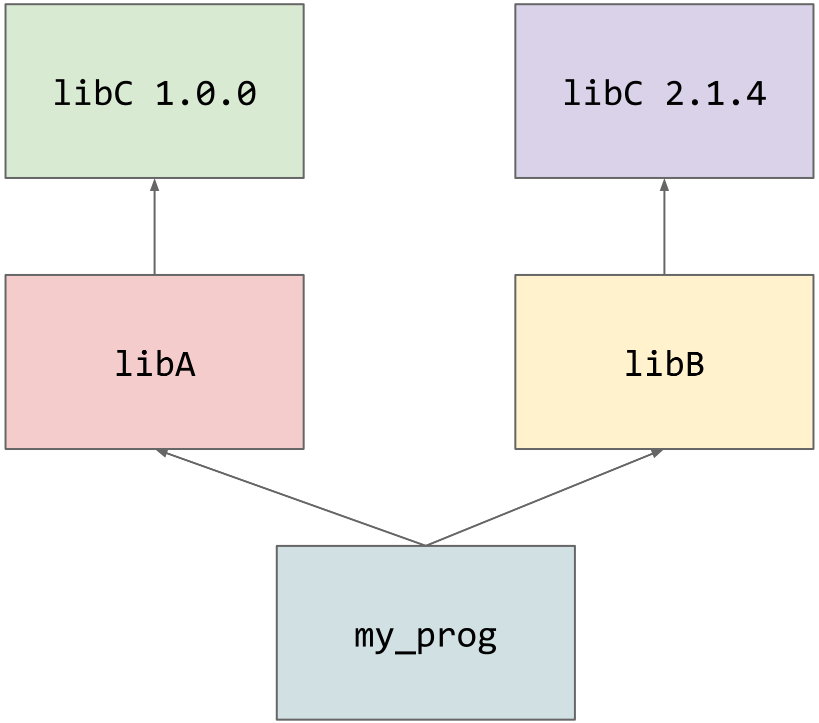
## Diamond problem?



## Diamond problem?



## Diamond problem?



## A few good choices

- Constraint #4: handle the diamond problem

## A few good choices

- Constraint #4: handle the diamond problem
- Incompatible versions of the same dependency in the tree are extremely painful

## A few good choices

- Constraint #4: handle the diamond problem
- Incompatible versions of the same dependency in the tree are extremely painful
- Eliminates: hunter

## A few good choices

- Constraint #5: be known
- I can’t put your package manager in this talk if I never heard about it
- Eliminates: ???

## A few good choices

- Conan (JFrog)
- vcpkg (Microsoft)
- cget (Paul Fultz II)

## A few good choices

- Conan (JFrog)
- vcpkg (Microsoft)
- cget (Paul Fultz II)

## The barbarian packager

- Started in 2015
- Today owned by JFrog
- Written in Python
- Around 300 packages
- Supports ARM and x86 on most platforms

CONAN.io
C/C++ Package Manager

## The barbarian packager

### conanfile.txt

```
[requires]
gtest/1.8.1@bincrafters/stable

[generators]
cmake_paths
```

## The barbarian packager

```
$ conan install ../

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

## The barbarian packager

### CMakeLists.txt

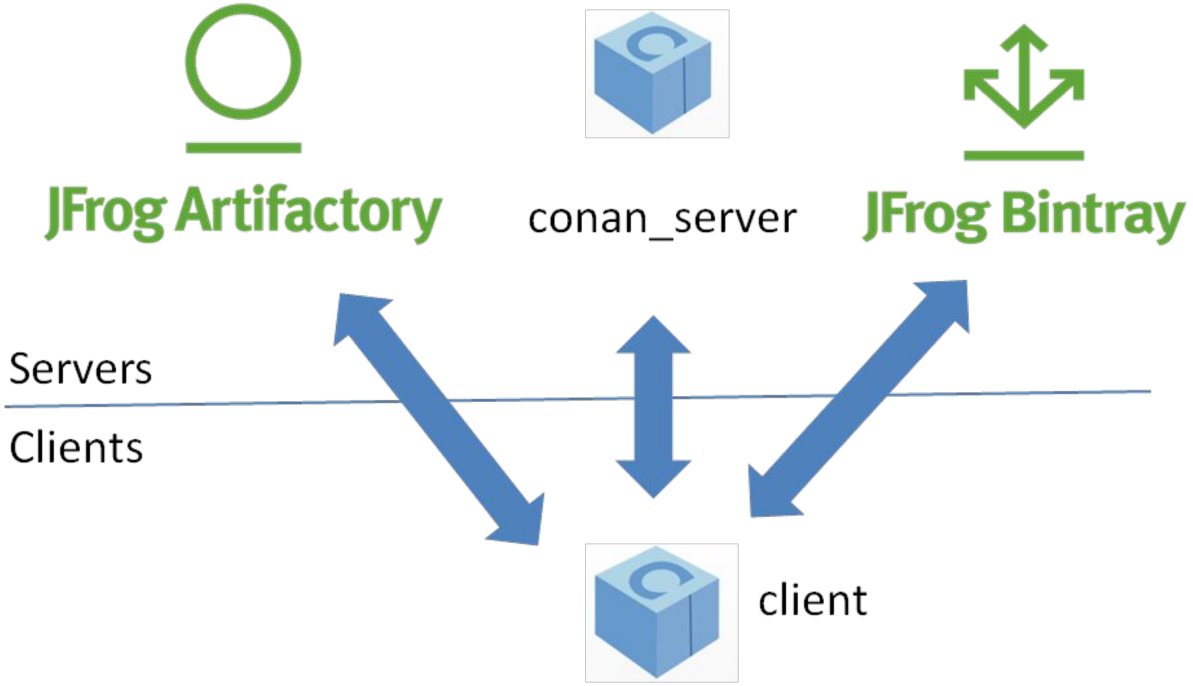
```
find_package(GTest REQUIRED)

enable_testing()
add_executable(foo foo_test.cpp)
target_link_libraries(foo PRIVATE GTest::GTest GTest::Main)
add_test(AllTestsInFoo foo)
```

## The barbarian packager

- Decentralized
- Select the remotes you want to use
- Offers a default repo of curated packages
- Companies can set up their own

## The barbarian packager



## The barbarian packager

- Uses binary caching by default
- Remotes can store artifacts with recipes
- Saves up compilation time immediately
- Better suited for closed environments

## The barbarian packager

- Default integration method can be intrusive
- Curated package repo is growing slowly
- Allows multiple versions of the same library
- Multi target generator is still experimental

## vcpkg

- Started in 2016
- Maintained by Microsoft
- Written in C++ and CMake
- Around 800 packages
- Supports ARM and x86 on Windows, Linux and OSX

## vcpkg

```
$ vcpkg install googletest

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

## vcpkg

### CMakeLists.txt

```
find_package(GTest REQUIRED)

enable_testing()
add_executable(foo foo_test.cpp)
target_link_libraries(foo PRIVATE GTest::GTest GTest::Main)
add_test(AllTestsInFoo foo)
```

## vcpkg

- Centralized versioned repository
- Fast growing list of OSS packages
- High quality curation
- Builds and handles Debug/Release by default

## vcpkg

- No binary caching out of the box
- Linux support still a bit behind
- Workflow is quite different for users and maintainers

## 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

3

## Making your library packageable

Help us poor maintainers

## *Keep It Simple Stupid*

## Tried and true solutions

- Don’t try to be creative!
- All package maintainers know CMake
- All clients will have it installed
- Anything else will require more work

## The Big Three

- Expect your users to be on Windows, Linux and OSX
- Stick to what's available on all three
- It's fine to have Win32 and POSIX toggles
- MinGW and Cygwin are not Windows support

## Assembly vs portability

- If you have to use Assembly
- Don't (*)
- Remember Windows has MASM, Linux has GAS, OSX has no default.
- 3rd parties introduce build dependencies

## Assembly vs portability

- Even with a portable syntax, ASM is still not portable
- Calling conventions and other ABI things vary between systems
- Simpler to have one source per target and use the system toolchain

## Build Dependencies

- Code generators, extra assemblers, exotic build systems…
- Avoid them if possible
- Remember they need to be built for the host platform, not the target

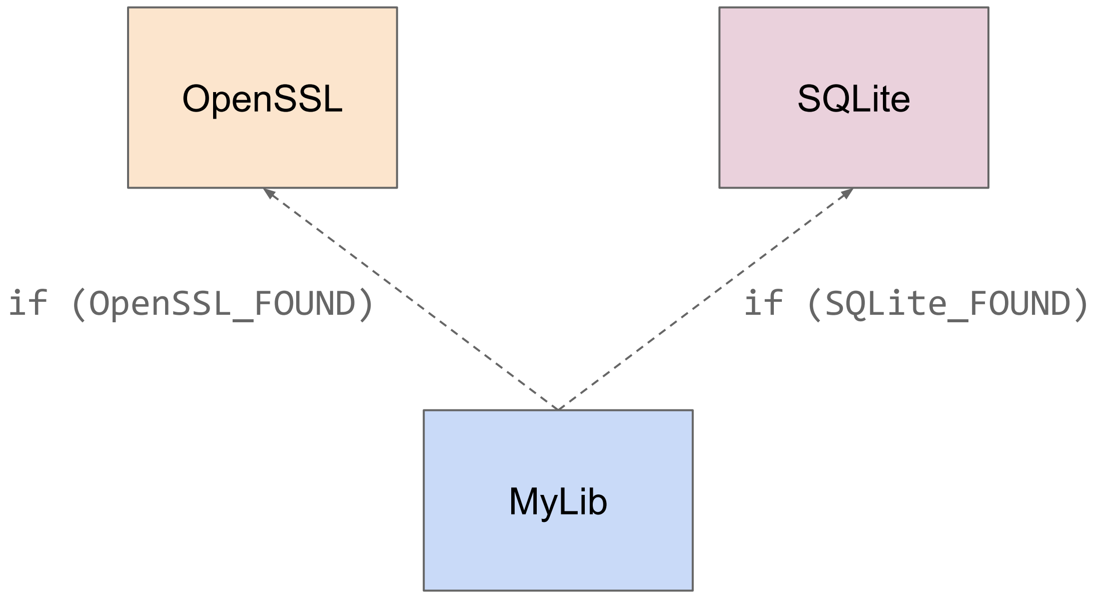
## Don’t hide dependencies

- Tell us which dependencies you require!
- Use `find_package(XXX REQUIRED)`
- Don’t try to install missing dependencies
- Don’t disable features and continue

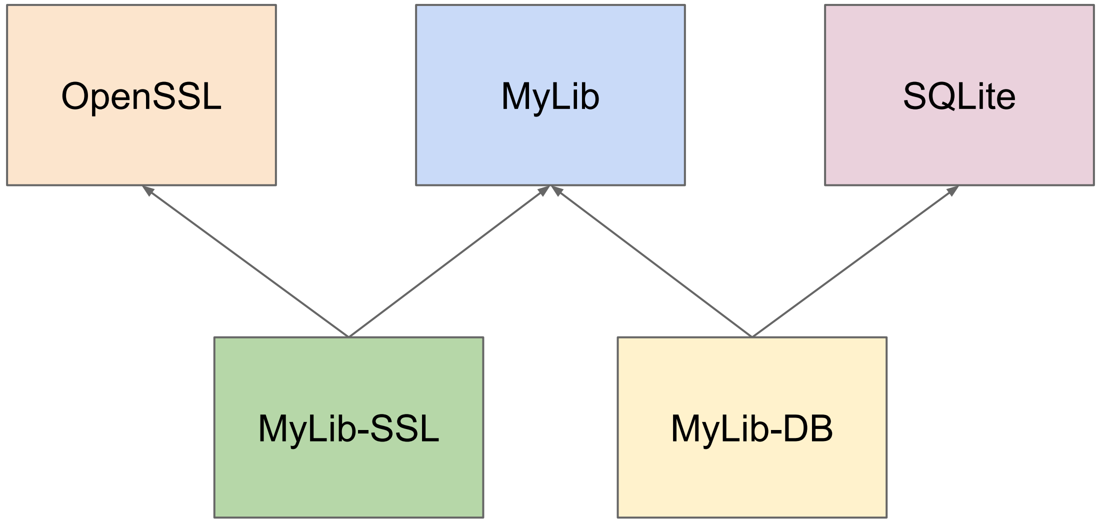
## About feature toggles

- Avoid them!
- Make additional libraries that can be packaged separately
- If you have a toggle, disable it by default and fail it can’t be built when enabled

## About feature toggles



## About feature toggles



## Preserving ABI

- Your library has to be ABI compatible with anything built with the same toolchain
- Change CFLAGS or CXXFLAGS only if you’re sure it doesn’t break ABI
- Checking and failing is safer than patching

## Preserving ABI

**Safe**

- Warning flags (-W)
- Optimization flags (-O)
- Debug flags (-g)
- C++ Standard flags (-std)

**Unsafe**

- Architecture flags (-m)
- Runtime flags (-stdlib, /MT, /MD)
- Sanitizer flags (-asan)

## Beware of ABI defines

- Some #defines can also break ABI
- `_ITERATOR_DEBUG_LEVEL`
- `_GLIBCXX_USE_CXX11_ABI`
- Don’t touch them!

## Package me if you can!

- `cmake -DCMAKE_TOOLCHAIN_FILE=...`
- `make`
- `make install`

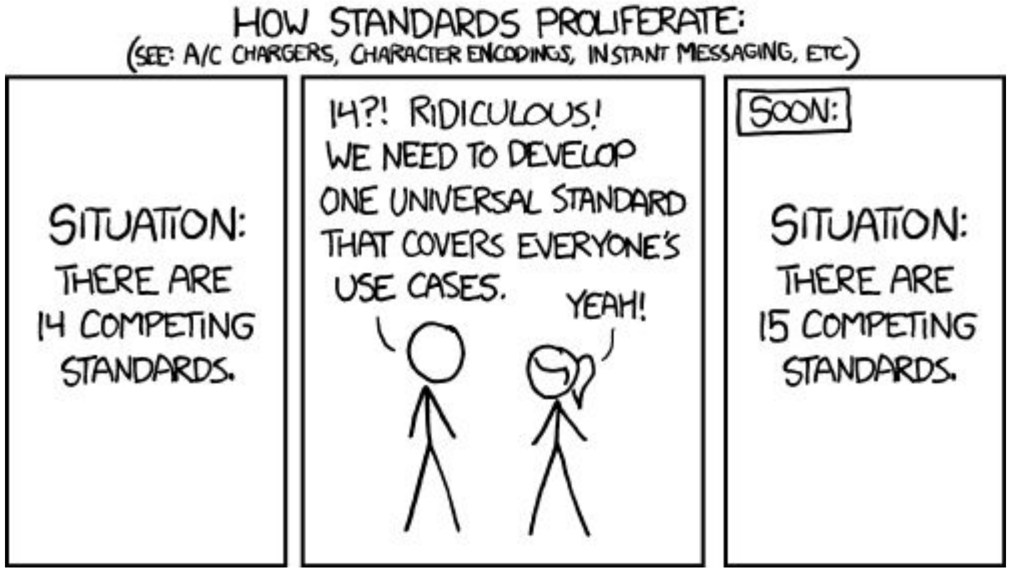
4

## What's next?

Are we there yet?

## Slow **progress**?

- C++ isn't a new language
- Build is not part of the standard
- We have to harmonize 30 years of diverging practices


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

*Convergence is easy!*

## A build standard

- We can’t rewrite the build of all existing libraries
- But we can package and expose them in a standard way
- New projects should be held to a higher standard

## A build standard

- CMake isn’t the best build system ever but...
- Going solo today will only isolate your library from the rest of the ecosystem
- Declarative CMakeLists are easy to migrate once we agree on a better system

## A build standard today

- Write a simple CMakeLists
- Run checks, fail if they aren’t met
- Rely on a toolchain file for build environment
- Describe requirements in README

## Challenges for tomorrow

- More standard!
- Describing requirements
- Producing a package manifest upon install

## Challenges for tomorrow

- Lower the cost of entry
- Generate toolchain files when installing development kit
- Or provide a wizard to setup one

## Challenges for tomorrow

- Get support from the build system
- Offer a strict “packaging” mode
- Report incompatible patterns in build files

## How can I help?

- Try out a package manager
- Make your library packageable
- Submit a recipe for Conan and vcpkg
- Tell your friends!

## Package management today

- Package managers are already out there
- Write packageable libraries
- Document your requirements
- Use a toolchain file

# Thanks!

## Any questions ?

You can reach me at

mro@puchiko.net

@MatRopert

@mropert

https://mropert.github.io

## Resources

- Don't package your libraries, write packagable libraries! (R. Schumacher, CppCon 2018)
- How To Make Package Managers Cry (K. Hoste, FOSDEM 2018)
- Why Not Conan 1, 2 and 3 (D. Rodriguez-Losada, CppCon ‘16, 17 and ‘18)