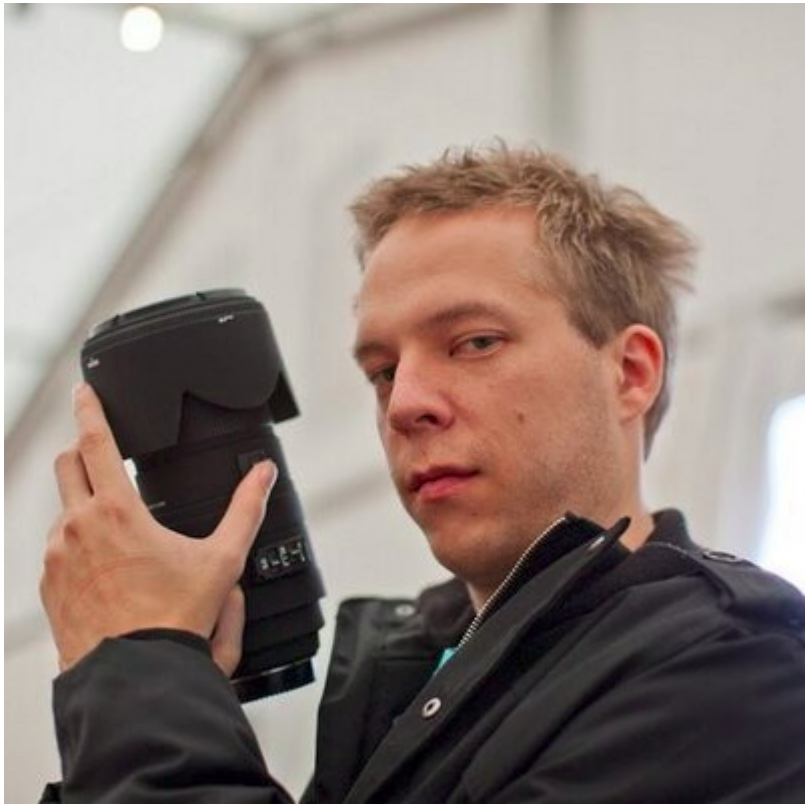
Build systems


Bulldozer
vtech

Who are we doing this for

The creator of autotools?

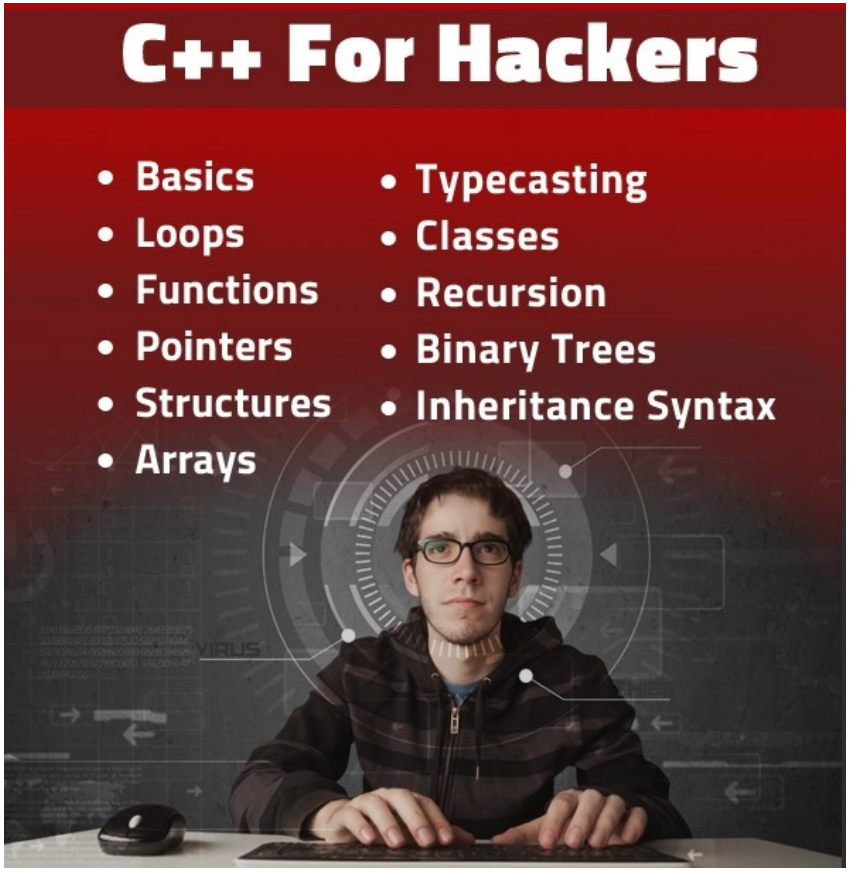
Build-tool specific experts?



Source:
CppCast,
Jussi Pakkanen,
Meson author

People that attend conferences?

Familiar with the language?



Source: http://www.hackinsight.org/free-content,33.html

New to the language

Source: Pixnio.com

New to programming entirely

Students seeing a compiler for the first time in their life

Children learning new skills

TOMTOM

My target

- You can read and write
- You can understand English keywords
  - Somewhat
- You have a computer
  - But a 10-year old computer should suffice

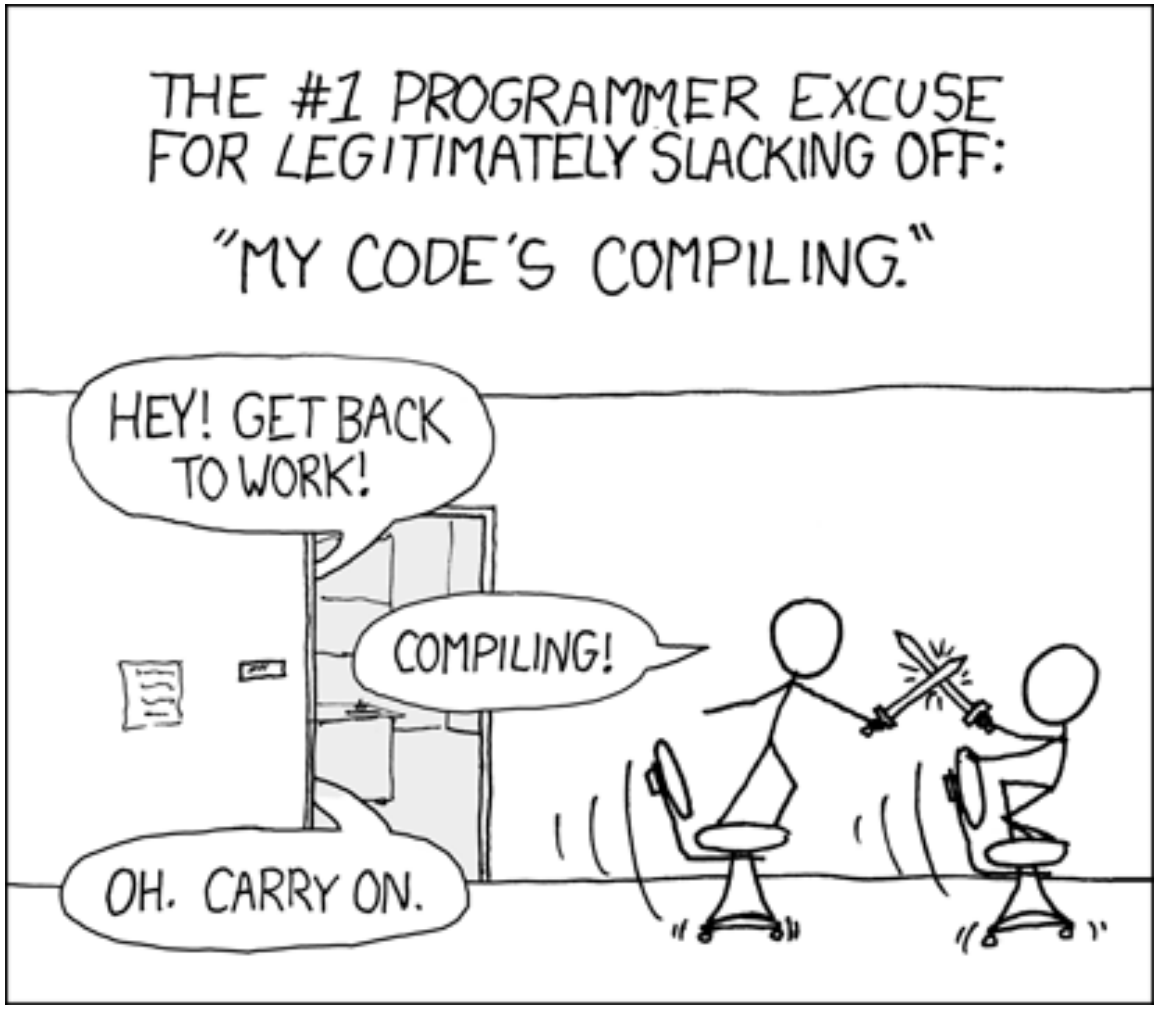
My target

- Make it really simple to start
- Allow the users to learn beyond
- Offer a cheap and smooth upgrade path

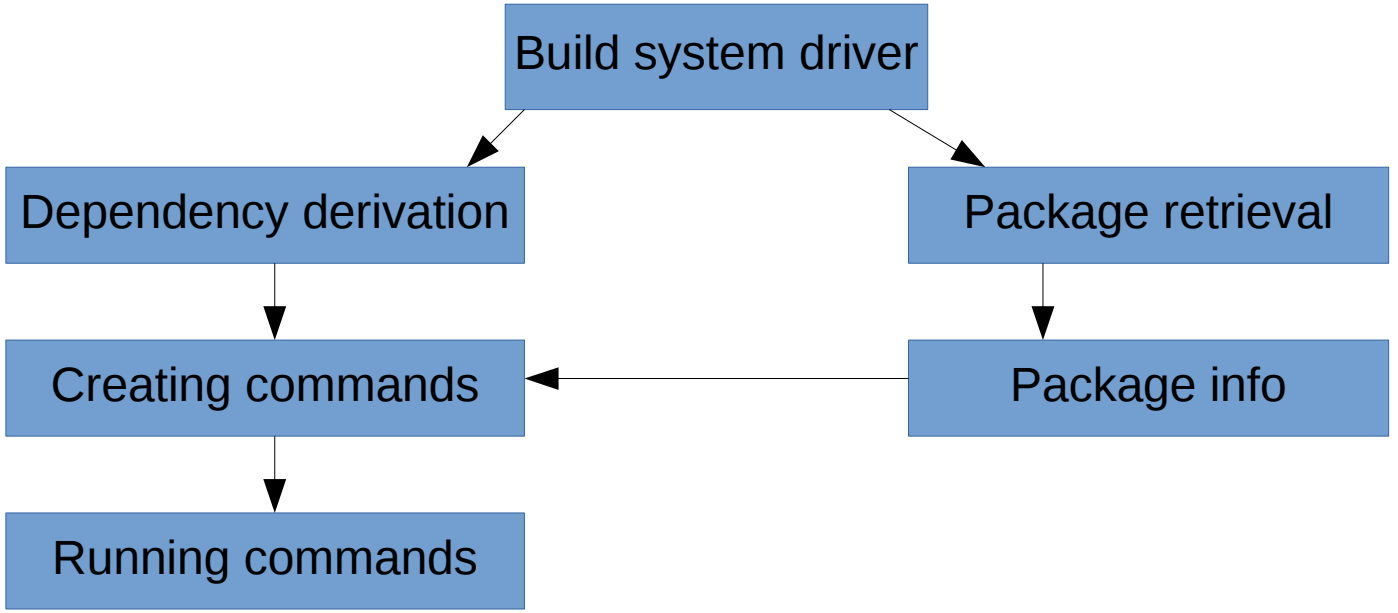
Simplicity

- You saw Kate’s talk yesterday

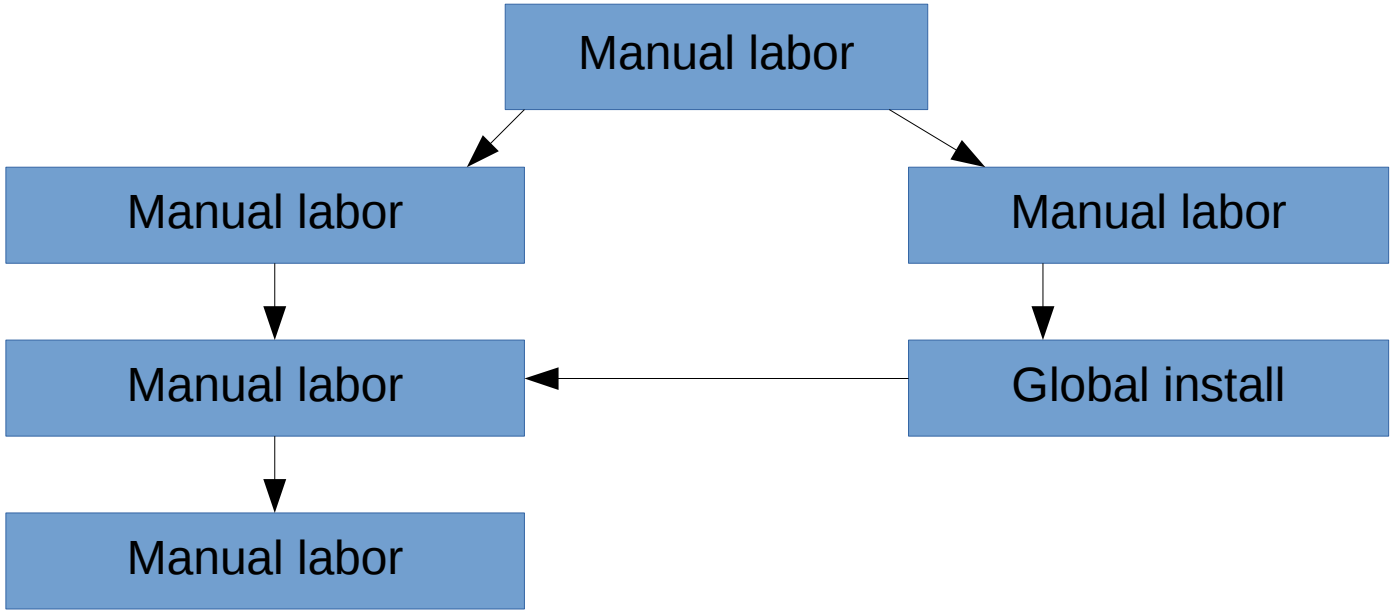
  “Is this really all I need to change?”



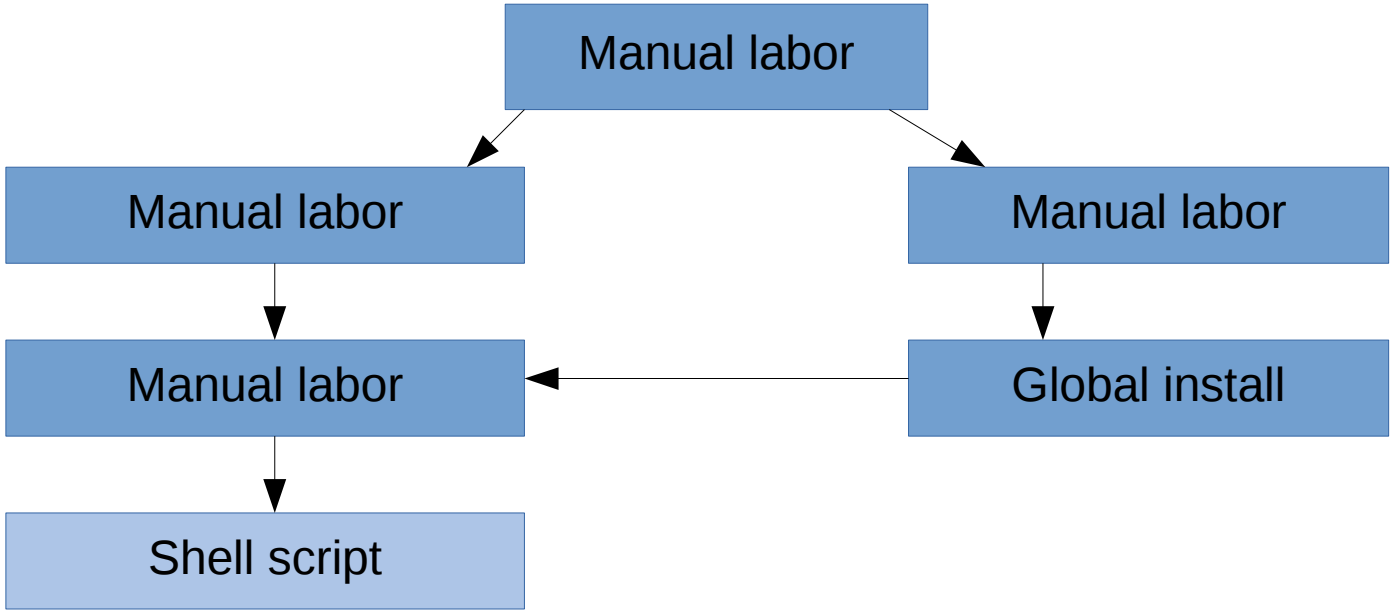
The parts of a build



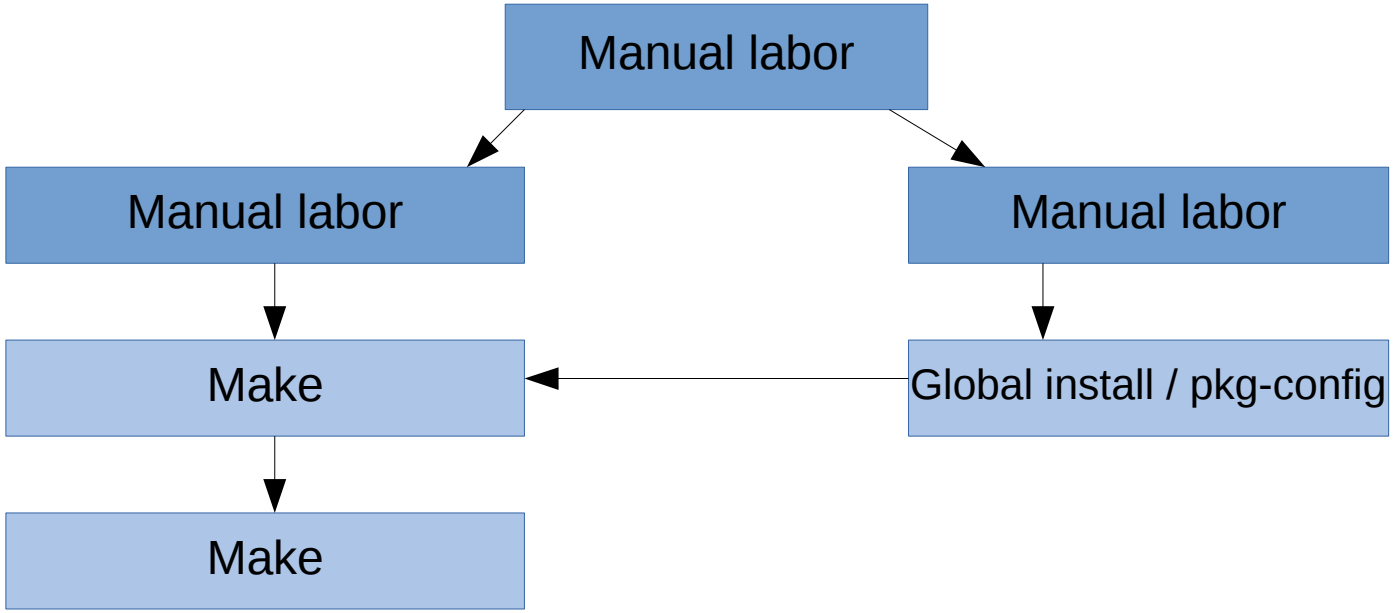
Way before time began

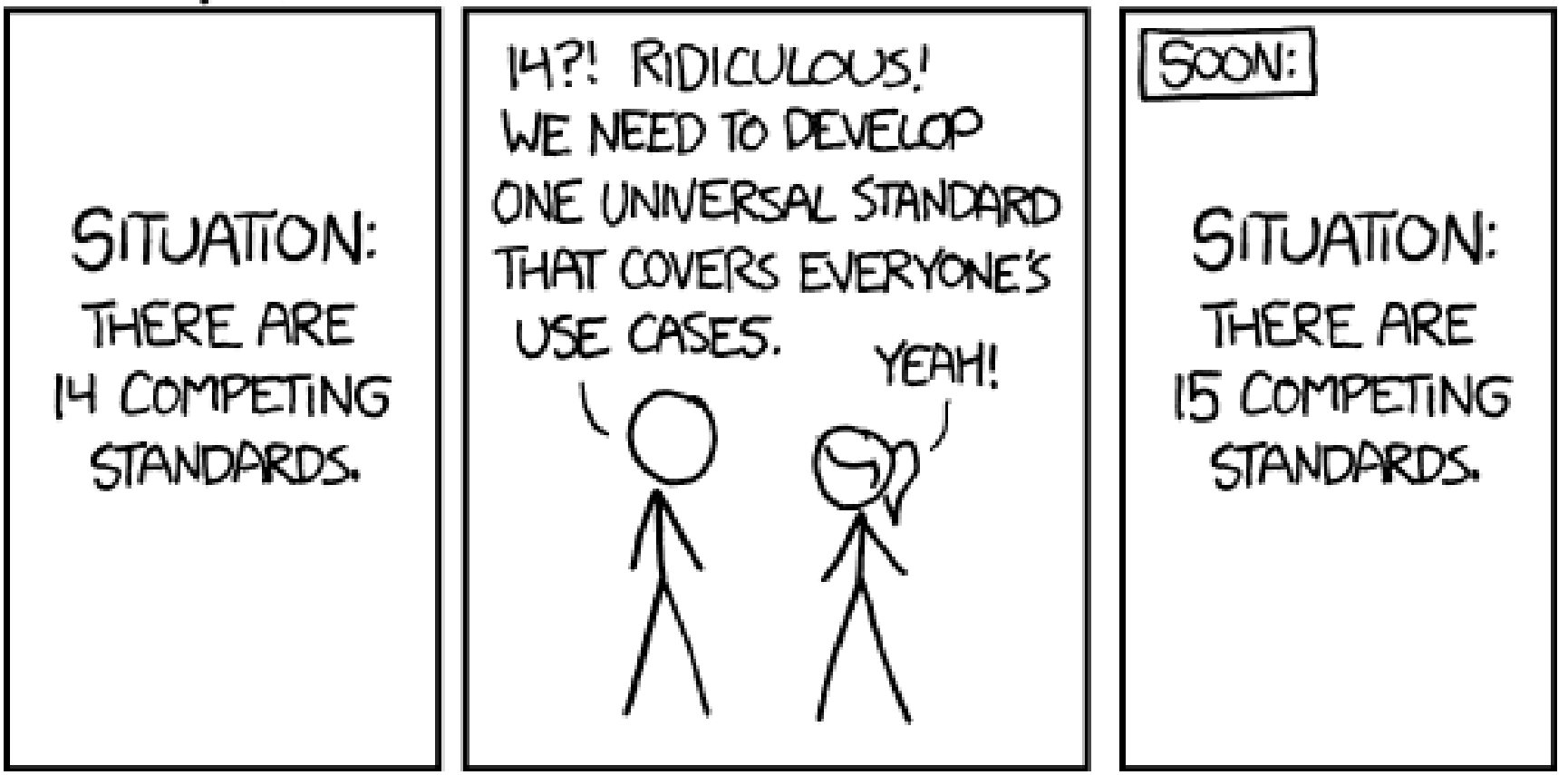


Way before time began



The Make era





All the makes

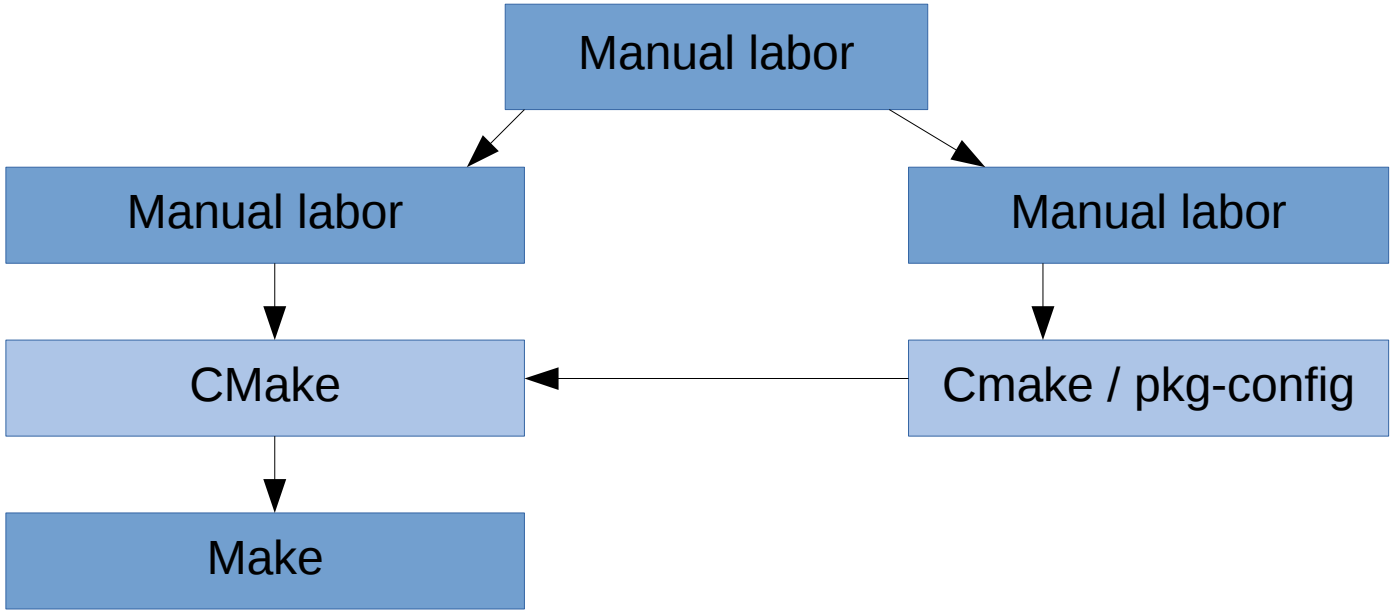
- bmake
- cmake
- dmake
- emake
- fmake
- gmake
- hmake
- imake
- jmake
- kmake
- mmake
- nmake
- omake
- pmake
- qmake
- rmake
- smake
- tmake
- umake
- vmake
- wmake
- xmake
- ymake
- zmake

If you want to make another make

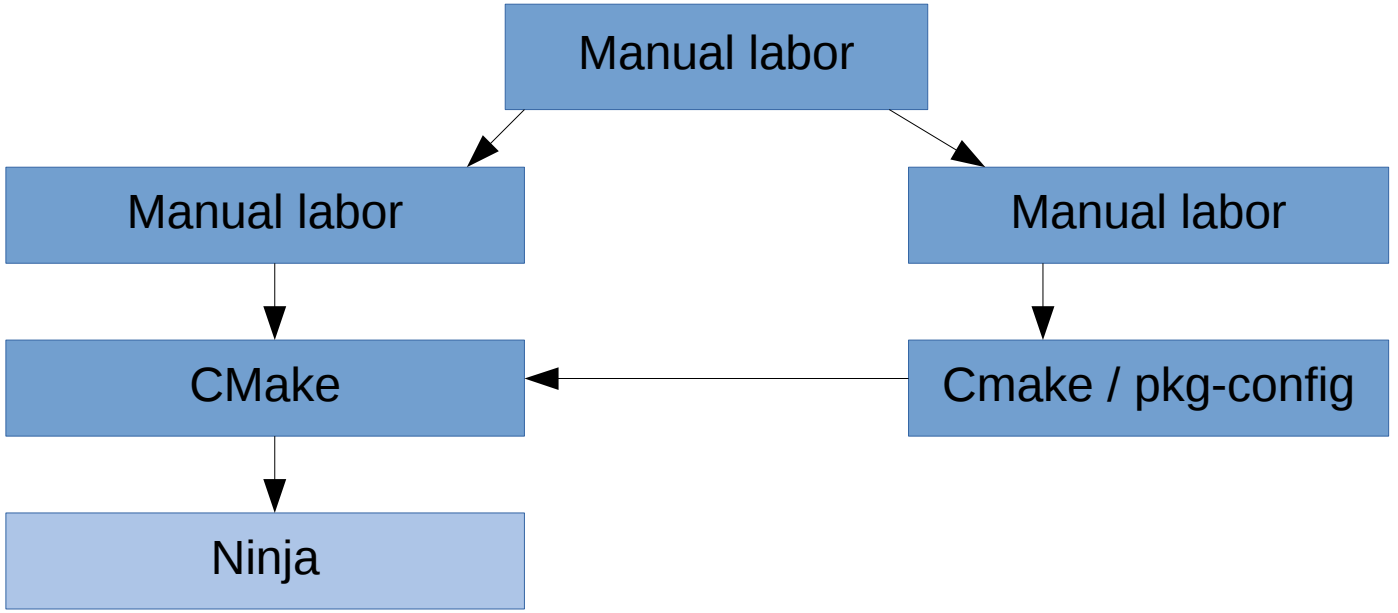
- amake
- lmake
  - Not to be confused with imake, of course

- But my list is from 2011, so maybe not
- What about 😀make ?

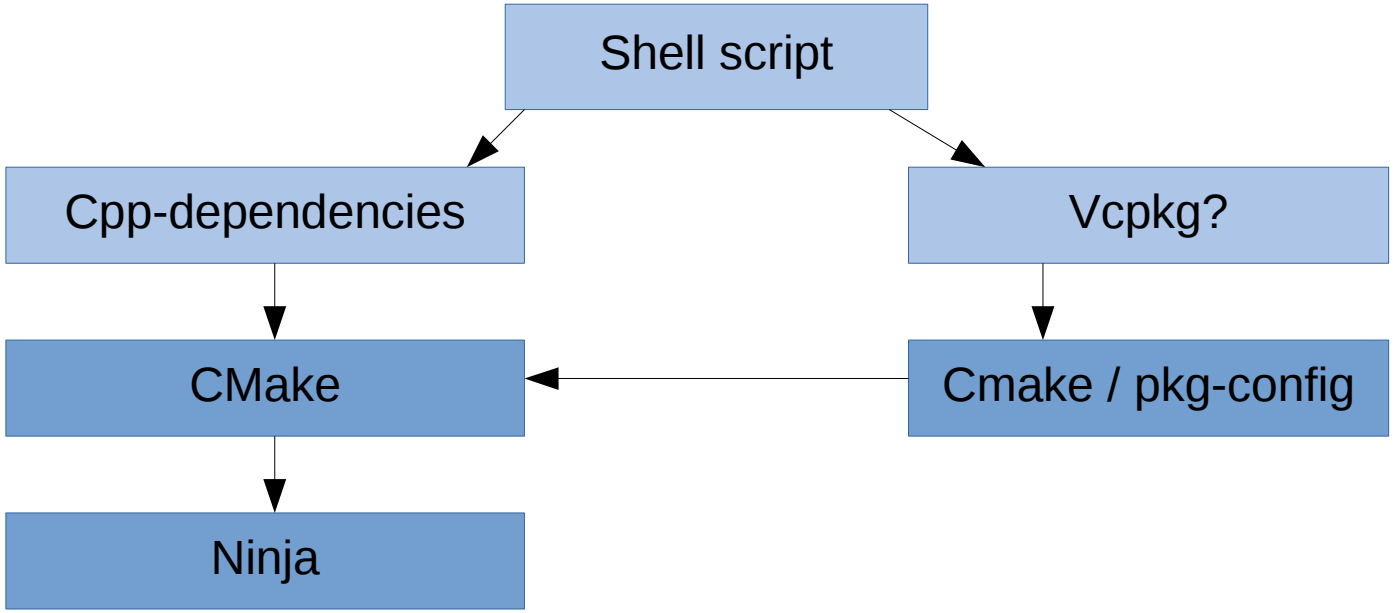
The Cmake era

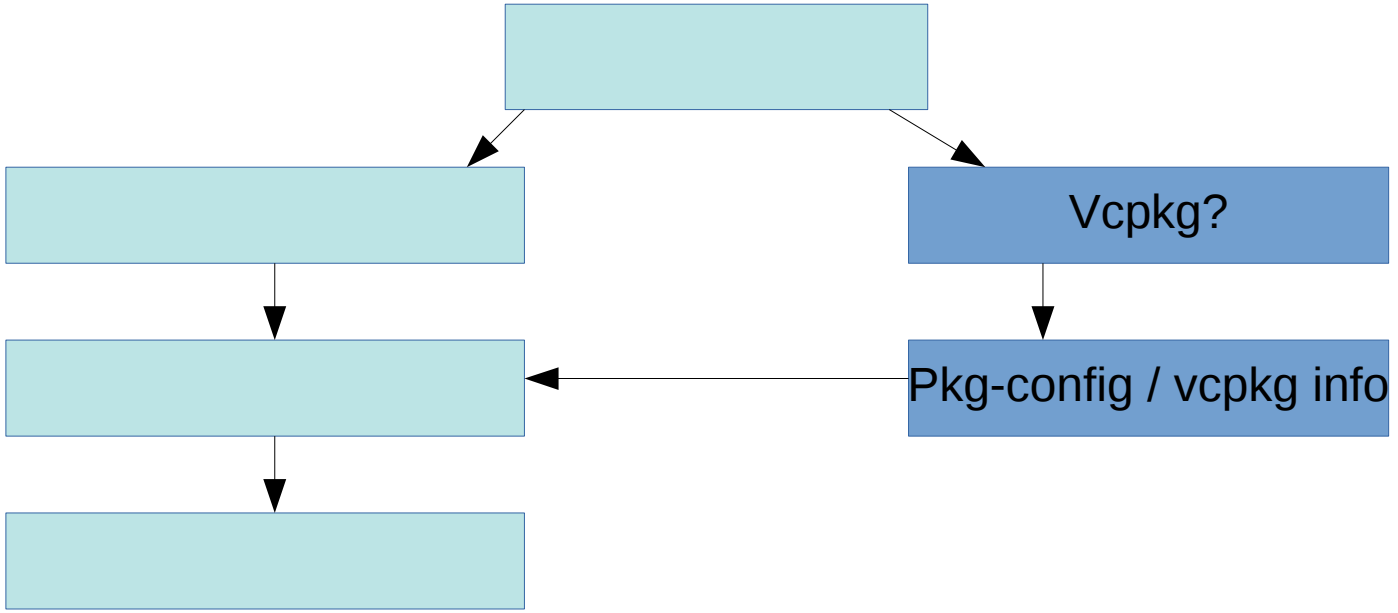


Beyond make



Recent history





Why do build systems always tend to complexity


All the corner cases

Total build sequence

- Build object files
- Pack static libraries
- Link executables
- Link shared libraries
- Run unit tests
- Run code formatter

Total build sequence

- Build object files
- Pack static libraries
- Link executables
- Link shared libraries
- Run unit tests
- Run code formatter
- Run static code analysis

Total build sequence

- Build object files
- Pack static libraries
- Link executables
- Link shared
- Run unit tes
- Run code fo
- Run static code analysis
- Upload to embedded target
- Send to binary package repository

- Download code from git
- Update version file & commit
- Git submodule init
- pdate
- ge of new
- Create release notes
- Announce on twitter
- Wget script | sudo bash -c

And more!

Modules

What are modules

- From build system point of view

Generated code

- Generated source files
- Having your source generator part of the same build
- That depends on some other generated file itself
- Recursively repeated.

Build system types

- Procedural
- Functional
- Declarative / descriptive
- Prescriptive / Generative

Procedural

- Your build steps
- In the order
- To run them

Procedural

- Simple to grasp
- Can be reliable

Procedural

- 100% manual scripting
  - Omissions
  - Bugs
  - Typos
  - Unintended deviations

Procedural

- Requires full knowledge of compile process
- No parallelism
- No incremental compilation
- **Very** easy to mess up

Functional

- Define how to derive the build steps
- Let the build system figure out what follows
- Excellent if compile steps and tools vary a lot

Functional

- Handles incremental builds
- Can be very reliable
  - But make doesn’t warn if you mess it up
- Portable
  - (arguably)

Functional

- Nowadays very unpopular
- Really hard to diagnose and fix
- Implementation incompatibility

Functional

- No good solution for dependencies
- Minor historical limitations

Declarative / Descriptive

- Most build systems
- About as verbose as the build script
- Allows arbitrary scripting

Who here has seen build scripts

- Special cases in some targets
- Custom hacks

- // TODO: remove this in 5 years
- // 2005-06-28 Temporary hack

Who knows *FOR SURE* that you don’t have this?

Remember that you can override - for example - add_library in cmake to do something else entirely.

This could be in somebody else’s scripts that you include

Prescriptive/Generative

- Instead of writing down what to build, derive what to build
- Allow minimal room for adjustments and exceptions
  - Nobody else can either

Prescriptive/Generative

- Complicated build tool
- Requires knowledge of what the tool expects

So how do we break the cycle of complexity?

Explicitly target beginners

- Very little configuration required
  - To compile your Hello World it should require absolutely no configuration.
  - As far as is reasonably possible, it should require no configuration.
  - If there is configurability, it should be restricted to acceptable minor adjustments.
    - Name and type of component

Explicitly target beginners

- Limit creativity
  - Make it ***not*** turing complete.

Explicitly target beginners

- Clearly delineate what the tool will accomodate, and what it will not.
- Limit deployment type proliferation
- Error for build-breaking conditions
- Warn for things that cause build-breaking conditions

Make it prescriptive
Make it generative

Why do prescriptive systems fail

- People hate changing habits
- People dislike being prescribed things

Works for Java

Works for Basic


```
**** COMMODORE 64 BASIC V2 ****
K RAM SYSTEM  38911 BASIC BYTES F
```

Works for Pascal

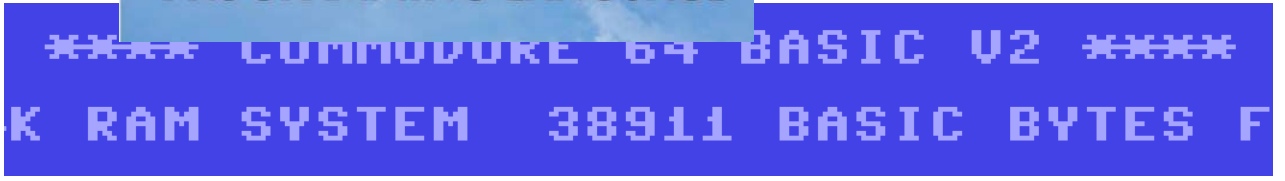
Pascal

Works for C#

Pascal

Why not C++?

cal



Inheritance from C

- Started before build systems
- Did not standardize one
- People forked and innovated

  => zoo of tools & inscrutible build solutions

- C++ just went with this

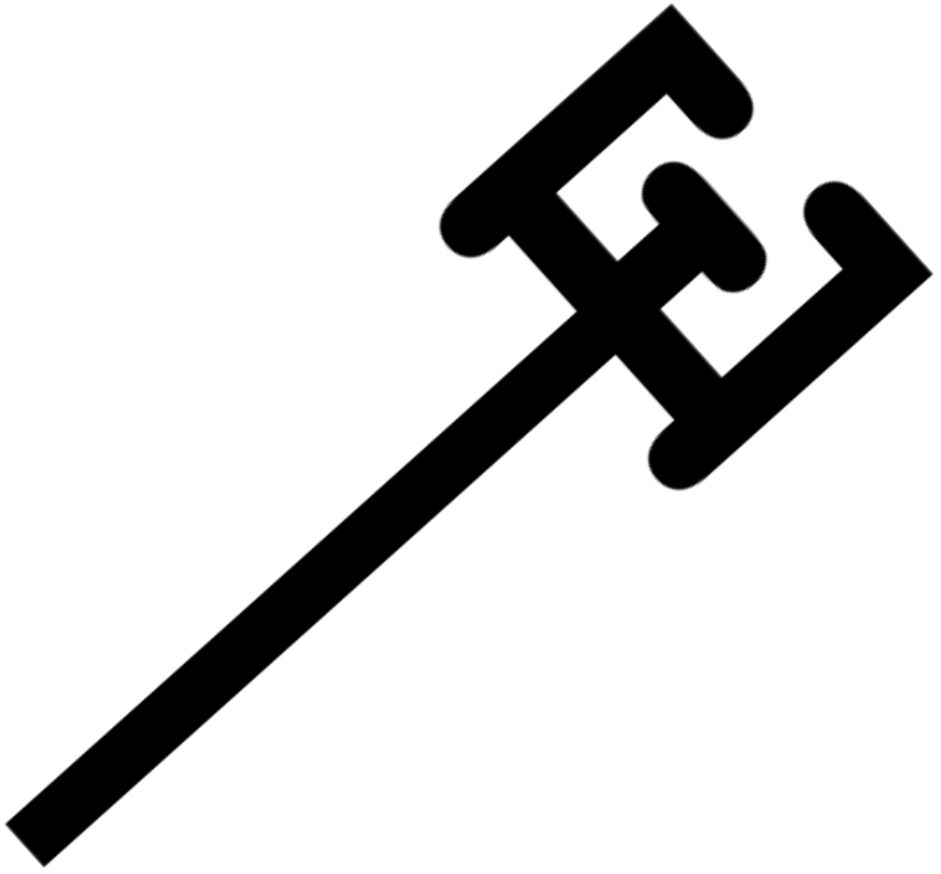
How do you make a prescriptive solution ***work***?

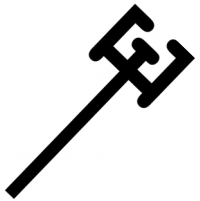
Prescribe the status quo

- Prescribe what we already do
  - in a structured way
  - Make everything derivable from the code
  - Allow the user to override
    - but only where we must allow variation

Standardized project layout

- Per component
  - src
  - include
  - test
  - doc
  - examples
  - ...





Pitchfork proposal by Colby Pike

- https://ol.reddit.com/r/cpp/comments/996q8o
- Take the current defacto standard
- Codify it
- Tweak it where you must
- https://github.com/vector-of-bool/pitchfork

What this offers to tools

- Warning people for relying on implementation details
- Implicit "at home" feeling in a new code base
- Automatically detecting dependencies

  → Zero-configuration project layout information

Dependencies

- Compile time
- Link time
- Run time

Compile time

- #include <pixel.hpp>
- import std.io;

Link time

- extern double d;
- void f();

Runtime

- Shared libraries
- Plugins
- Database links
- Operating system functions
- External hardware
- ...

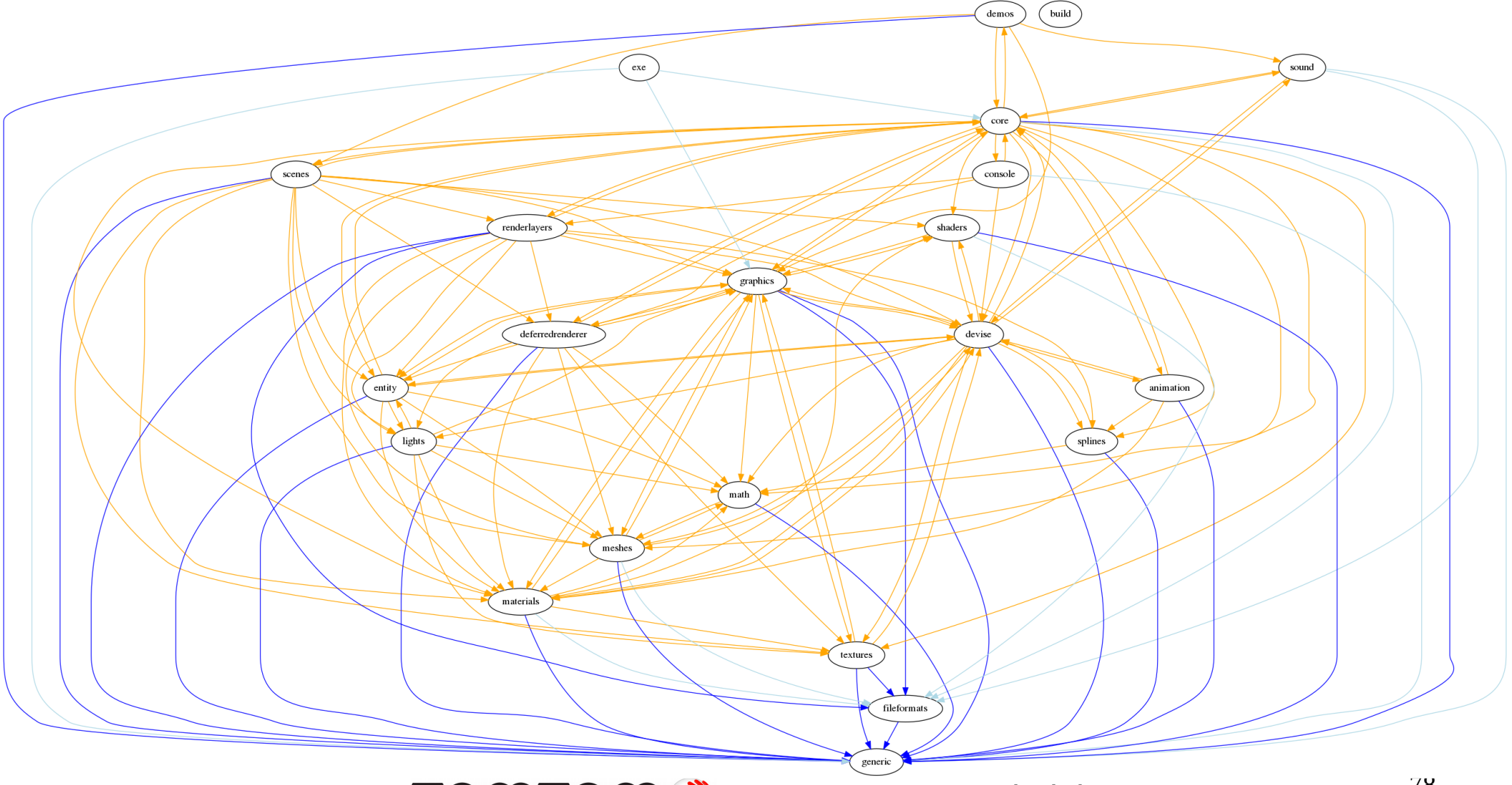
Dependencies

- Compile time is better than link time
  - Detectable dependencies are good
- Link time is better than run time
  - Breaking a build beats having complaining customers

Manually specified

- target_link_libraries

- Relative or absolute includes work around this
- Overspecified
- Underspecified



Humans

- Humans
  - Smart
  - Fallible

  (error rate >2%)
- Computers
  - consistently dumb
  - very consistent

  (error rate < 0.0000000001%)

Write Everything Twice

Not hardcoding inferred data

==

it's up to date

Cpp-dependencies

- Read all your code
  - Extract includes
  - Extract imports
- Derive what depends on what

Cpp-dependencies

- Discover transitive dependencies
- We know included files
  - Who includes them
  - What include path they need

Cpp-dependencies

- Component-wise thinking
  -  Gives us component roots
  - CMakeLists.txt location also works

Cpp-dependencies

- We know component locations
- We know all constituent files
- We know public and private dependencies
- We know public and private include paths

- This is 95% of what your build needs...

Cpp-dependencies at TomTom

- TomTom proprietary code base
  - 7M LoC
  - 500 components
  - Lot of test coverage
- 2012 4-core laptop with sufficient memory

- Expectancy: Should be able to match C preprocessor in speed

Cpp-dependencies at TomTom

- 2s run time with hot cache
- SSD speed limited with cold cache
- Interactive analysis
- Cycle detection & prevention
- Autogenerating cmakelists
  - Last time I checked we did about 60%

Cpp-dependencies at TomTom

- MeetingC++ talks in 2016 and 2017

- So how to make this into a build system?

Evoke

- A simple solution for the build system problem

- Evoke on Merriam-Webster:
  - 1 : to call forth (or up)
  - 2 : to re-create imaginatively

Basic idea

- Take
  - Cpp-dependencies
  - Cmake with fully generated build files
  - Ninja to build something
  - (Package manager to find needed packages)
- Roll that into one

Later goals

- Run continuously
- Cross-compile for other platforms

Bugs in all build systems now

- Race conditions
  - Save a file while it's being used for a compilation
    - the compile timestamp for object file will be *after* source file
    - Won't trigger rebuild if it was successful
    - Will cause very hairy bugs if you continue
  - Solution: When done with compiling, pre-date the file to the start of compilation

Bugs in all build systems now

- Incremental builds hide warnings
  - Files with warnings build successfully
  - Incremental build skips them
  - Warnings are hidden
- This is why -Werror is popular.
- Solution: Cache warnings & re-emit if not rebuilding
  - Breaks time-dependent commands

Design

- Take all includes and imports, derive file-level dependencies.
  - Ask package manager for packages corresponding to missing files?
  - If the package manager did something, rebuild the tree including those packages.

Design

- Translate file-level dependencies to component-level dependencies (like cpp-dependencies)
- Translate the components and dependencies into command lines for a given target with file-level dependencies (like CMake)
- Invoke the low-level build runner to handle these (like Ninja)
  - Is currently built-in, but could be externalized to Ninja, Make or something like it

Current limitations

- Derived dependencies from includes and imports
- Imperfect parser
  - >99.99% recognized though
- There is the 5% that cannot be done.

The ifdef problem

```
#ifdef X
#include "file1.h"
#else
#include "file2.h"
#endif
```

Current limitations

- Toolchain switching (Clang)
- Modules is completely untested.
  - Implemented as per current TS
  - Current Clang matches the design; I just haven't gotten around to it.
  - Does not include this weeks' changes

Current limitations

- Untested implementation
  - Android
- Unimplemented
  - OSX
  - Windows
  - iOS
  - No reason they can’t be, they just aren’t now

Current limitations

- Package manager links
  - Dependency import is not implemented properly
  - Demo uses a tiny stub. The stub conforms to the interface.
  - Tested with a self-written package system
    - Miles from release quality

- Open to contributors
- Come talk to me after the talk
- Reach out
- Even – *especially* – if you can't code

- **https://github.com/dascandy/evoke**

Demo

If we have time

Hello World from scratch

Pixel showcase with Evoke

Let’s make the **C++** world a

simpler

place

- Peter Bindels
- TomTom Intl
- @dascandy42
- dascandy at discord&slack

- https://github.com/tomtom-international/cpp-dependencies
- https://github.com/dascandy/evoke