accu 2023

C++ DEPENDENCIES DON'T HAVE TO BE PAINFUL

WHY YOU SHOULD USE A PACKAGE MANAGER

AUGUSTIN POPA

Which of these do you find frustrating about C++ development?

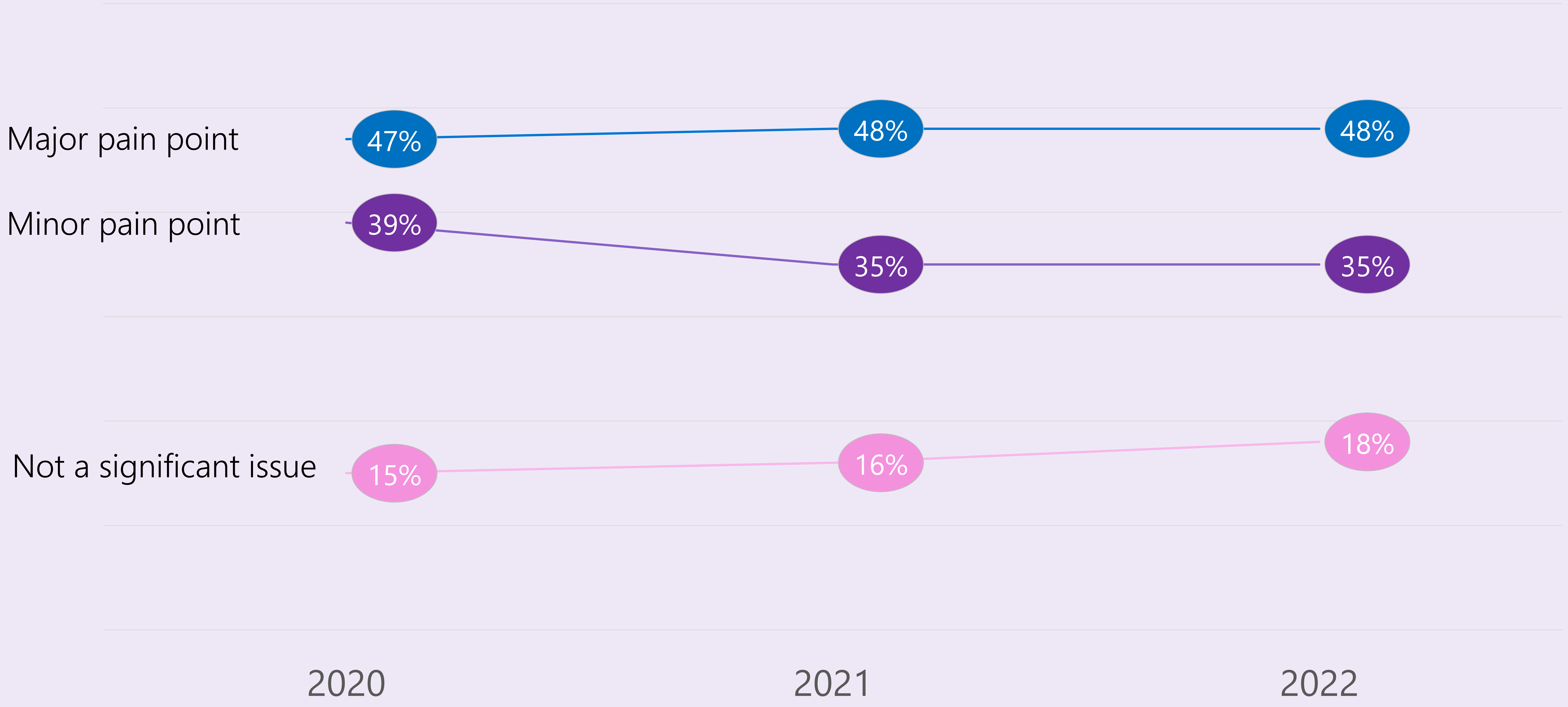
Question from recent ISO C++ surveys

- Build times
- Managing CMake projects
- Debugging issues in my code
- Parallelism support
- Memory safety
- Managing Makefiles
- Managing MSBuild projects
- Setting up a CI pipeline from scratch
- Security issues
- Type safety
- Managing libraries my application depends on
- Moving existing code to the latest language standard
- Setting up a development environment from scratch

Source: ISO C++ surveys (2020, 2021, 2022)

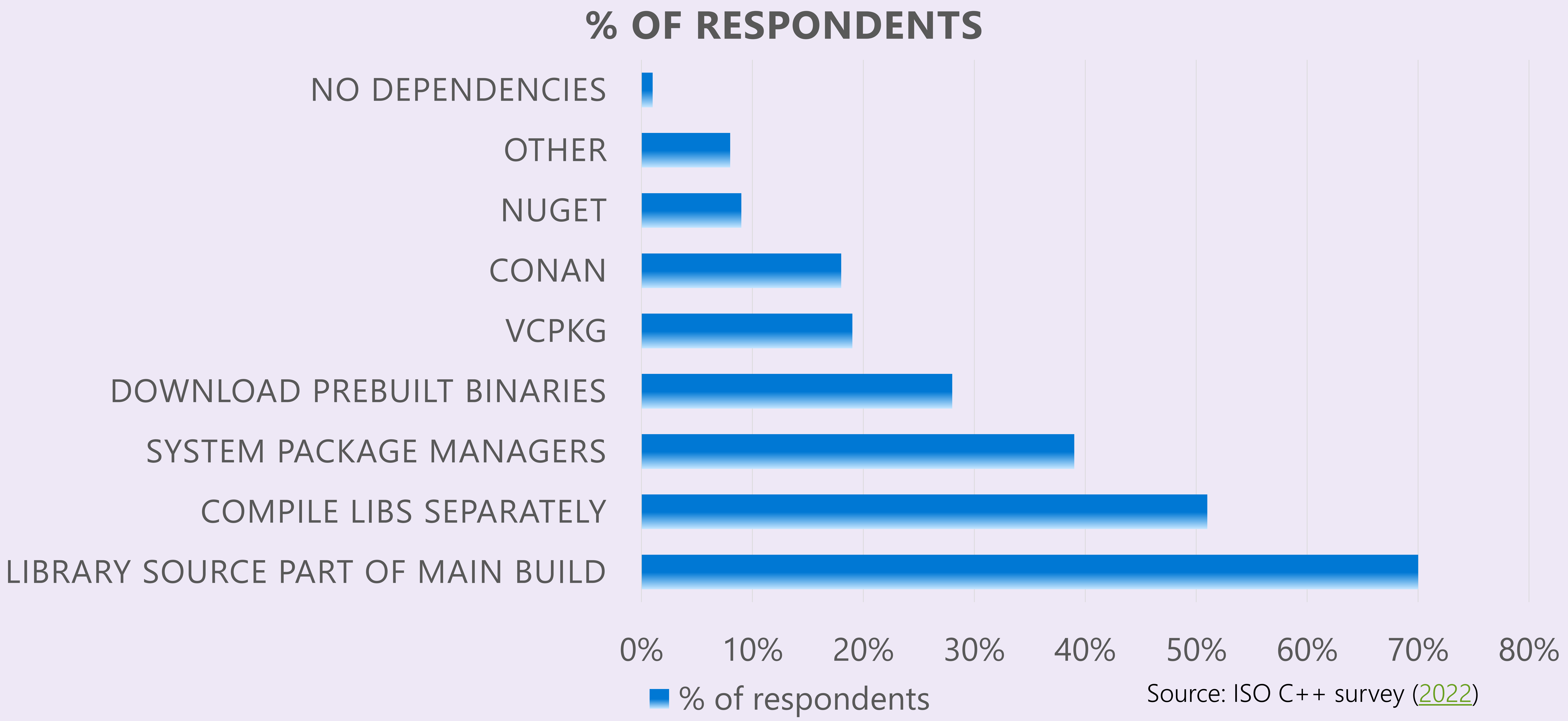
Which of these do you find frustrating about C++ development?

Answer: Managing libraries my application depends on



Source: ISO C++ surveys (2020, 2021, 2022)

How do you manage your C++ 1st and 3rd party libraries? (Check all that apply)



Source: ISO C++ survey (2022)

ISO C++ 2023 Survey Is Open!

Please take the survey to give feedback to the C++ community

https://isocpp.org/blog/2023/04/2023-annual-cpp-developer-survey-lite

Summary results will be publicly posted to isocpp.org.

Survey closes on April 25, 2023

ABI compatibility between libraries and consuming project

A common C++ problem

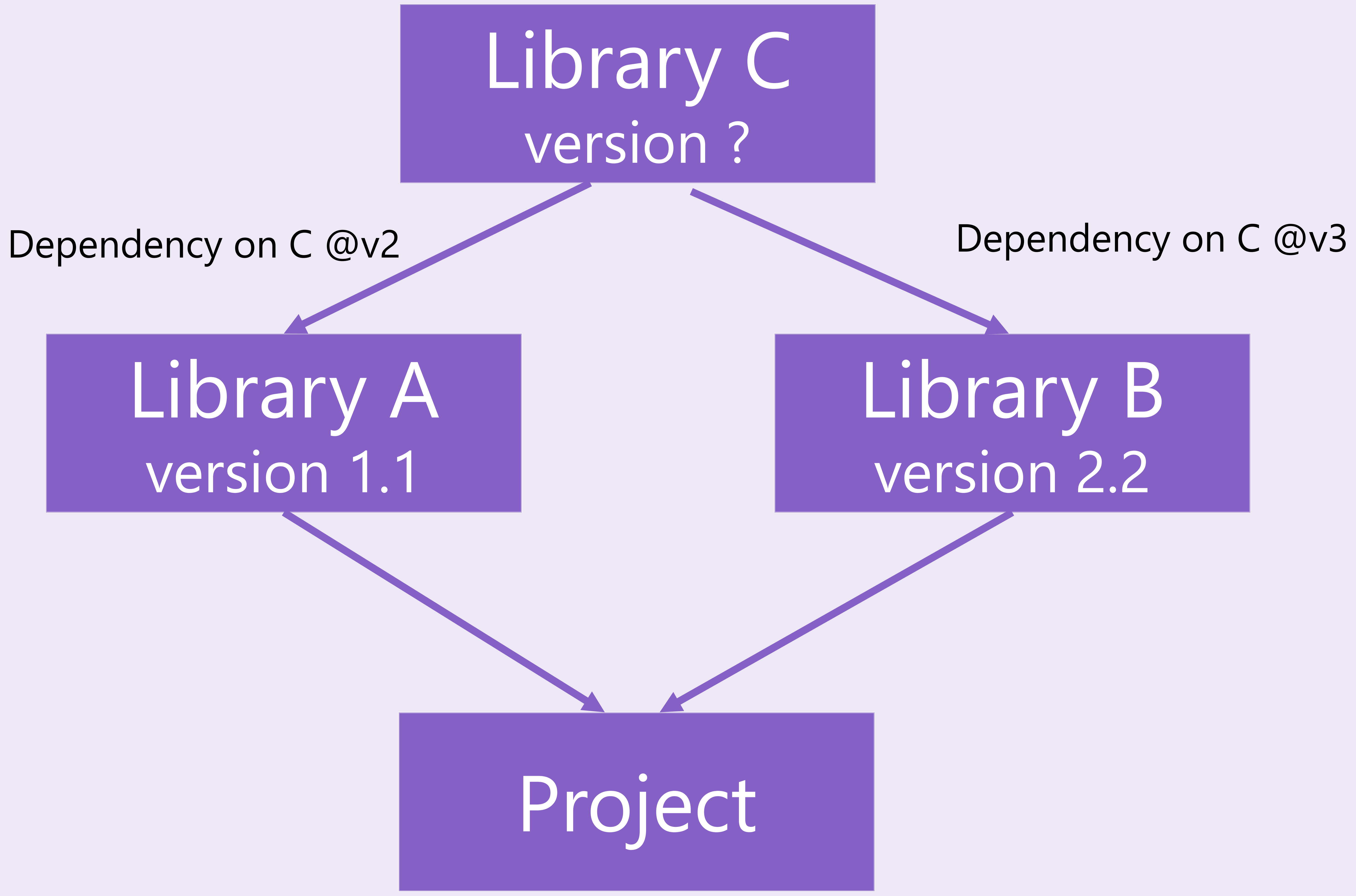
uncommon in other programming languages

How to break the ABI and your builds

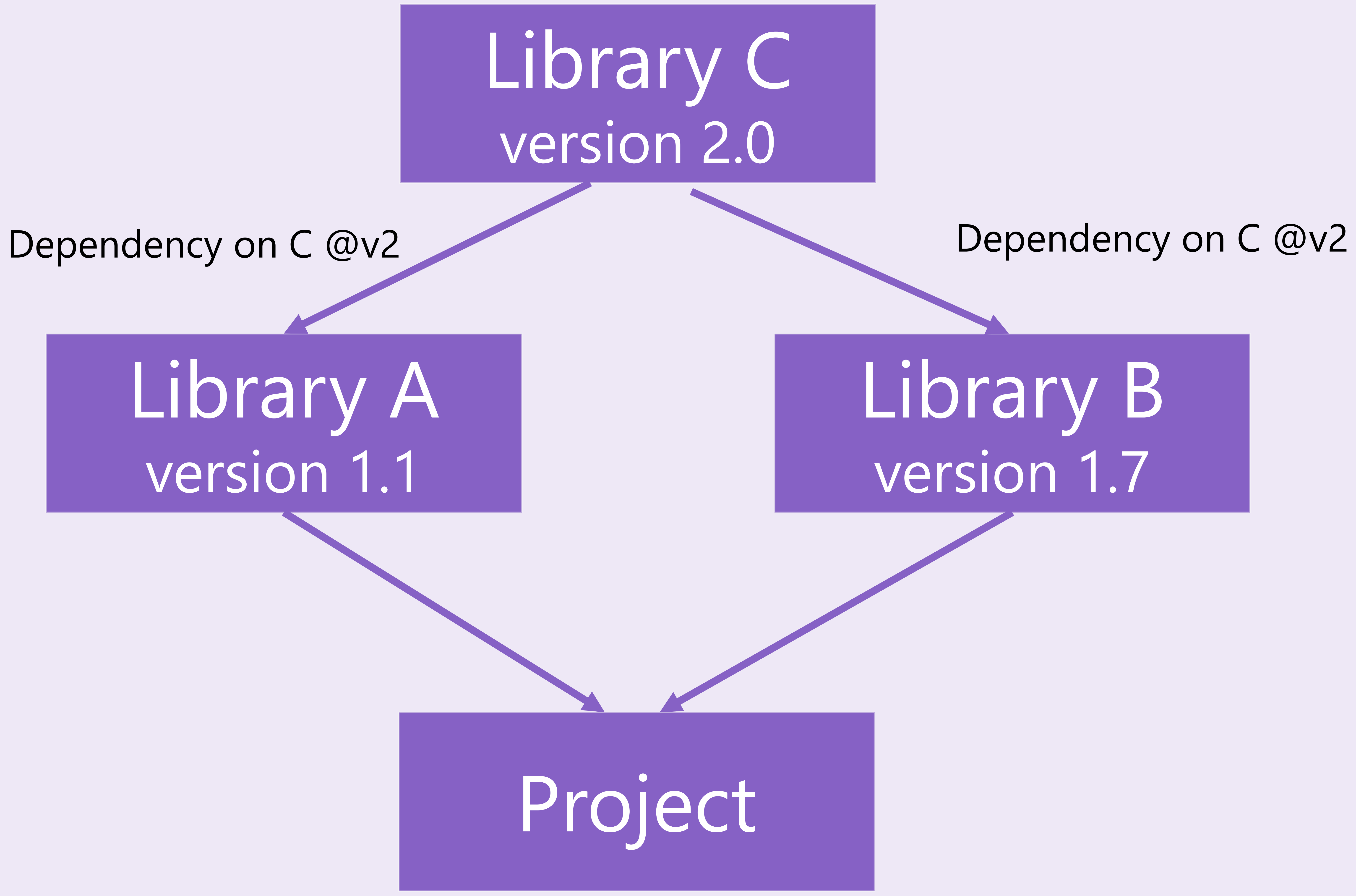
- Change the compiler
- Change the compiler version
- Change the target OS
- Change the target architecture
- Internal library changes

The “Diamond Problem”

The diamond problem of dependency management
Example

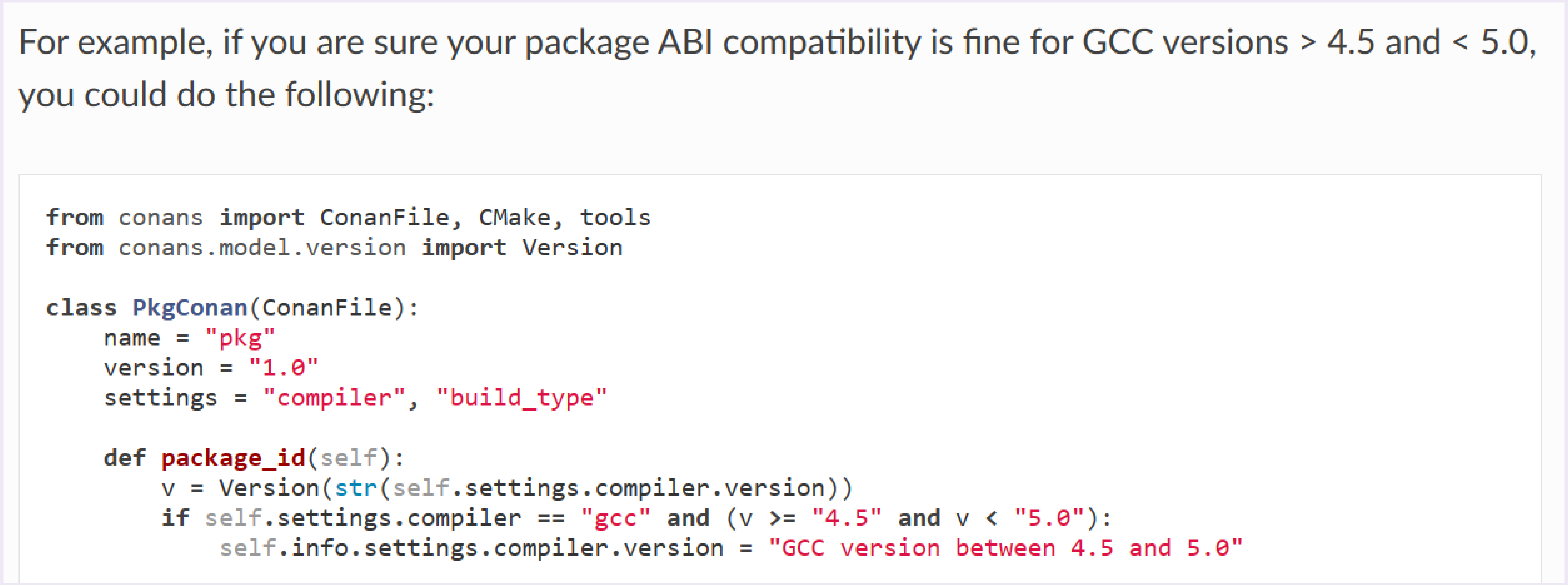


The diamond problem of dependency management
Example Resolved



Package managers can help here.

Packages can hint what they need
Conan example



For example, if you are sure your package ABI compatibility is fine for GCC versions > 4.5 and < 5.0, you could do the following:

```python
from conans import ConanFile, CMake, tools
from conans.model.version import Version

class PkgConan(ConanFile):
    name = "pkg"
    version = "1.0"
    settings = "compiler", "build_type"

    def package_id(self):
        v = Version(str(self.settings.compiler.version))
        if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
            self.info.settings.compiler.version = "GCC version between 4.5 and 5.0"
```

Semantic versioning hints

For libraries that respect semver

LIBRARY_NAME

version 1.2.3

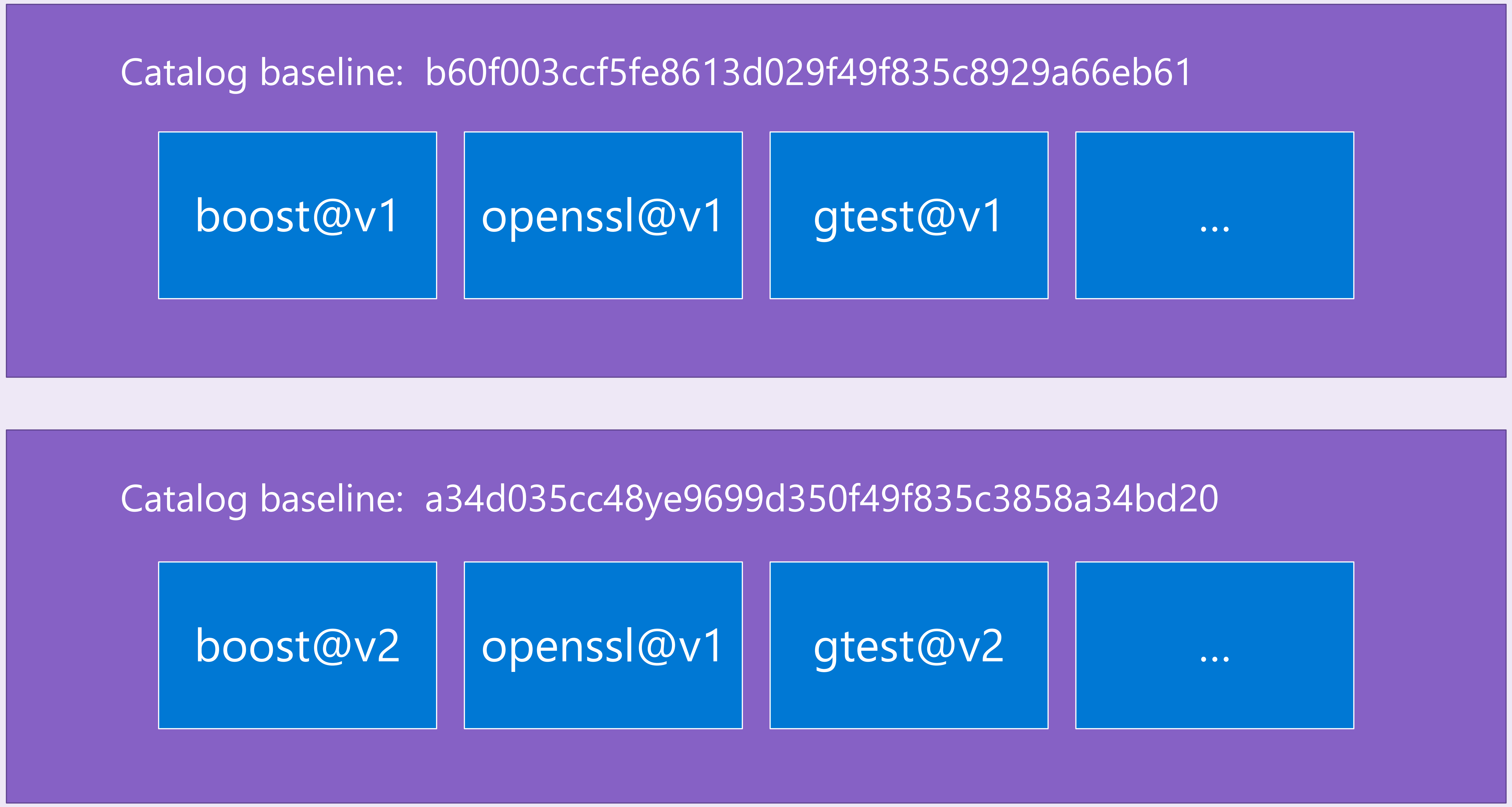
Major version
ABI definitely breaks

Minor version
ABI shouldn't break

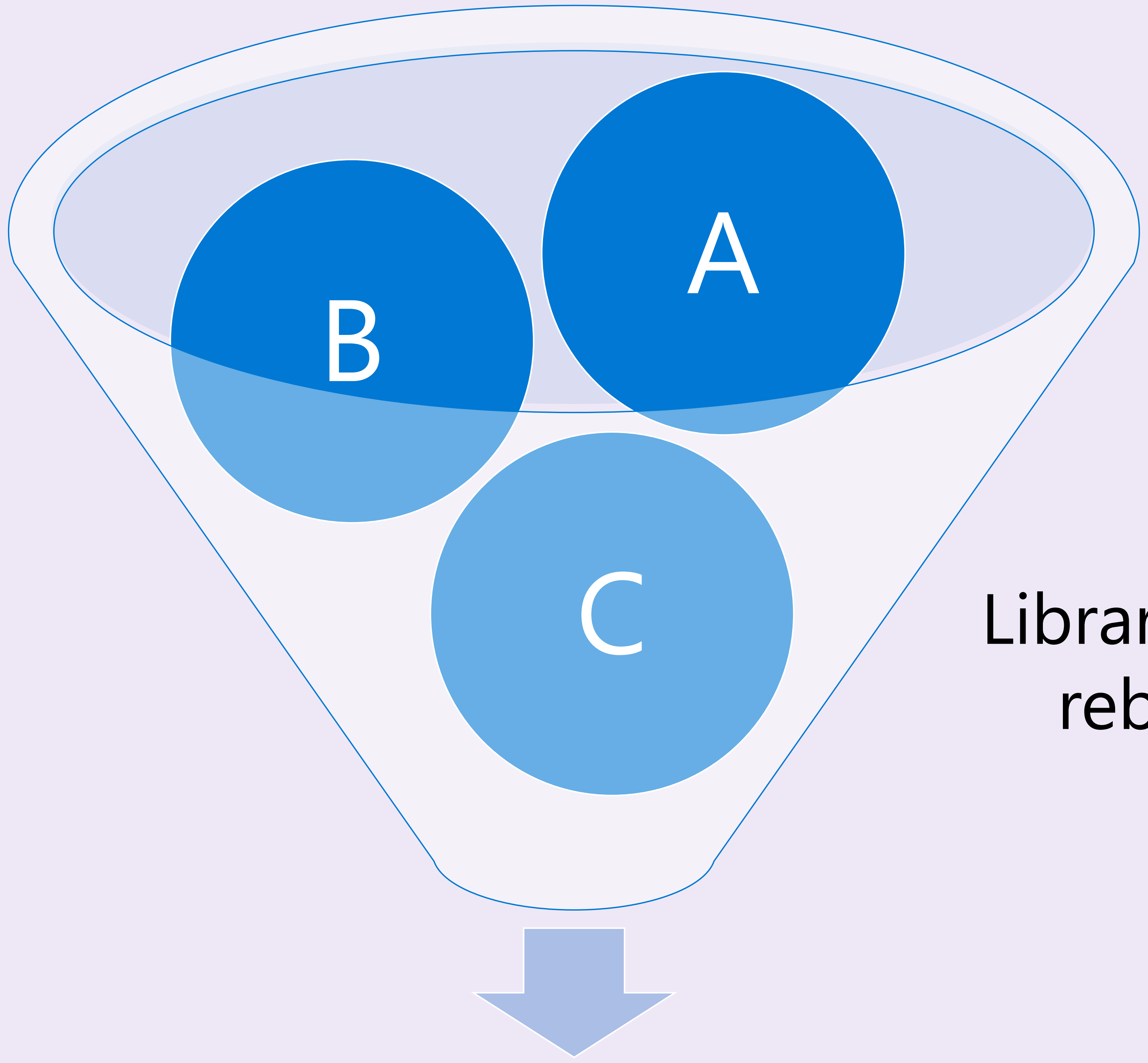
Patch
ABI is OK

Baselines in vcpkg

Making version matching an implementation detail

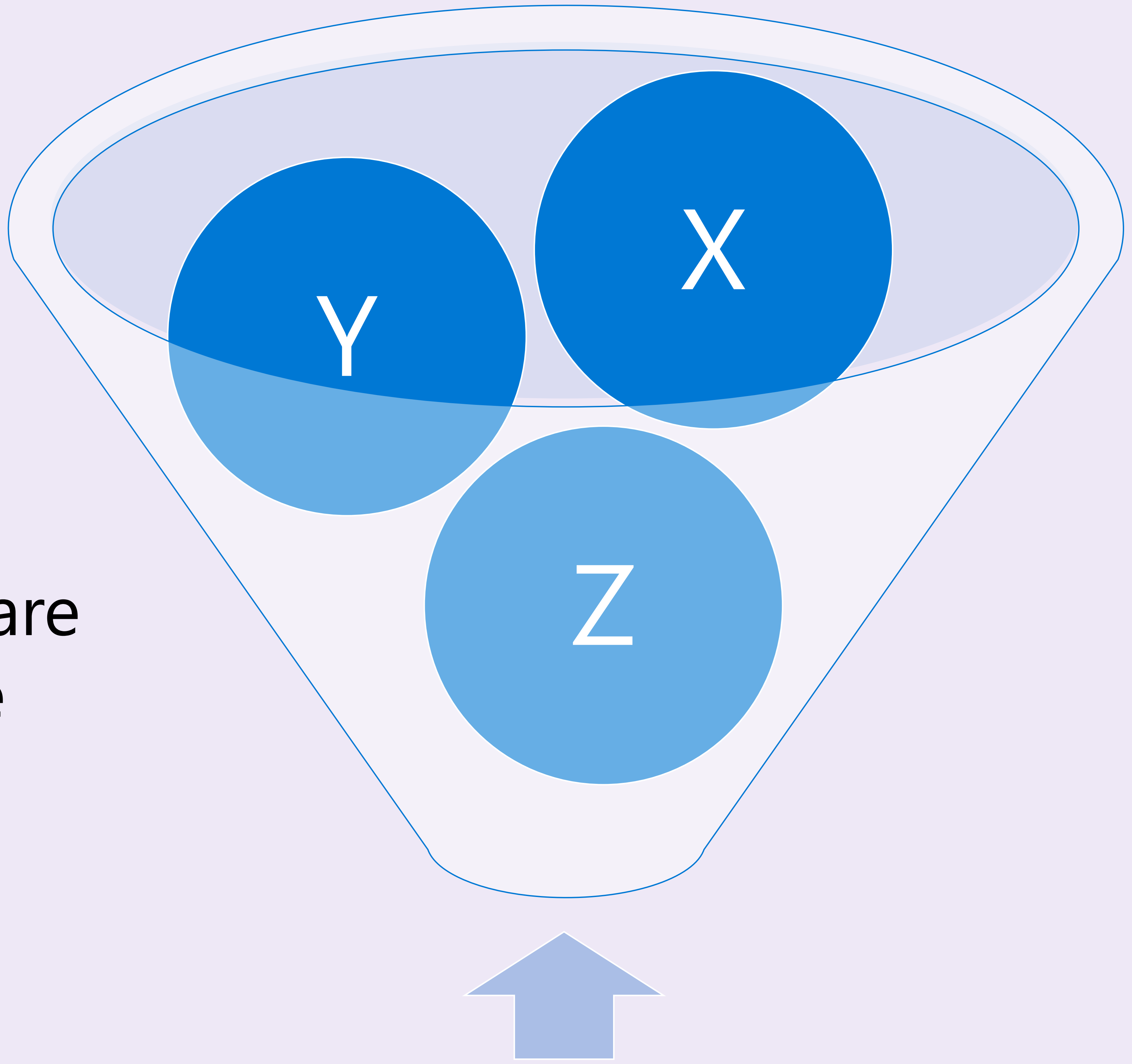

Catalog baseline: b60f003ccf5fe8613d029f49f835c8929a66eb61
boost@v1
openssl@v1
gtest@v1
...
Catalog baseline: a34d035cc48ye9699d350f49f835c3858a34bd20
boost@v2
openssl@v1
gtest@v2
...

The “cones of destruction” in vcpkg
Preventing diamond problems



Library being updated uses A, B, C

Libraries A, B, C, X, Y, Z are rebuilt along with the changed library



Library being updated is used by X, Y, Z

Minimum versioning (version>=) in vcpkg
Example

Dependency constraints:

Library A v1.0 depends on Library B v1.0

Library A v1.1 depends on Library B v1.0 and
Library C v3.0

Library A v1.2 depends on Library B v2.0 and
Library C v3.0

Library C has no dependencies

Consumer's vcpkg.json:

```
{
    "name": "example",
    "version": "1.0.0",
    "dependencies": [
        { "name": "A", "version>=": "1.1" },
        { "name": "C", "version>=": "2.0" }
    ],
    "builtin-baseline": "<some git commit with A's baseline at 1.0>"
}
```

Solution: A v1.1, B v1.0, C v3.0

Overrides in vcpkg
Example

Consumer's vcpkg.json:

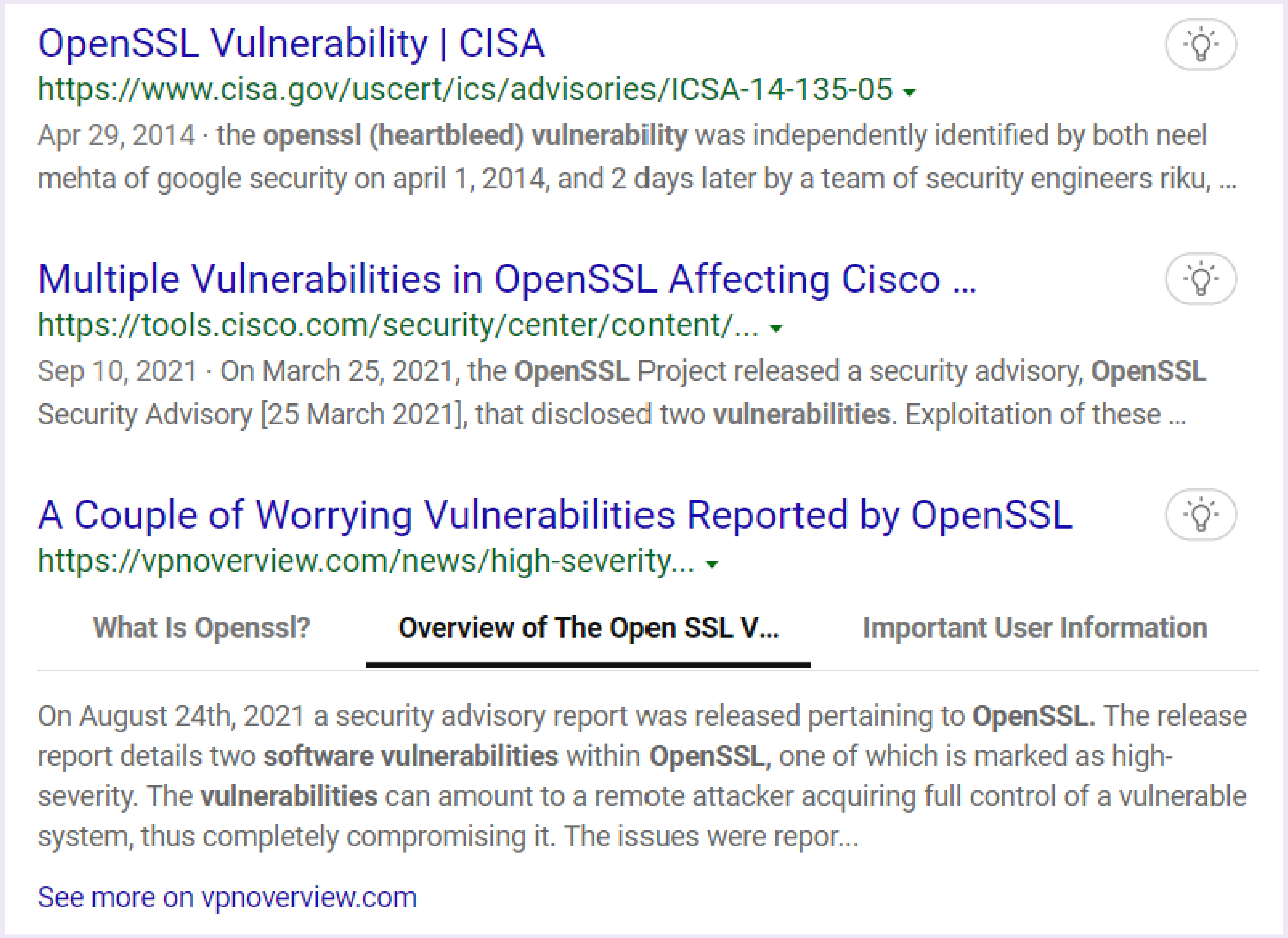
```
{
  "name": "project",
  "version-semver": "1.0.0",
  "dependencies": [
    { "name": "zlib", "version>=": "1.2.11#9" },
    "fmt"
  ],
  "builtin-baseline":"3426db05b996481ca31e95fff3734cf23e0f51bc",
  "overrides": [
    { "name": "fmt", "version": "6.0.0" }
  ]
}
```

fmt is acquired at version 6.0.0, regardless of any baselines or version>= constraints for it elsewhere

Overrides are only available to end consumers in the dependency graph

Keeping dependencies up to date is important
Performance, new features, bug fixes, security, …

Using a package manager makes it easier to update your libraries and keep them current



OpenSSL Vulnerability | CISA
https://www.cisa.gov/uscert/ics/advisories/ICSA-14-135-05
Apr 29, 2014 · the **openssl (heartbleed) vulnerability** was independently identified by both neel mehta of google security on april 1, 2014, and 2 days later by a team of security engineers riku, ...

Multiple Vulnerabilities in OpenSSL Affecting Cisco ...
https://tools.cisco.com/security/center/content/...
Sep 10, 2021 · On March 25, 2021, the **OpenSSL** Project released a security advisory, **OpenSSL** Security Advisory [25 March 2021], that disclosed two **vulnerabilities**. Exploitation of these ...

A Couple of Worrying Vulnerabilities Reported by OpenSSL
https://vpnoverview.com/news/high-severity...

On August 24th, 2021 a security advisory report was released pertaining to **OpenSSL.** The release report details two **software vulnerabilities** within **OpenSSL,** one of which is marked as high-severity. The **vulnerabilities** can amount to a remote attacker acquiring full control of a vulnerable system, thus completely compromising it. The issues were repor...

See more on vpnoverview.com

Other C++ package manager benefits

- Building packages from source and binary caching
- Control and flexibility over package versions
- Reproducible build environments (using manifests)
- Large, tested package catalogs
- Consistent open-source and closed-source package experience
- Support for offline builds
- Substantial number of open-source community contributions

When you should consider a package manager
Any one of these is enough

1. When your project has more than 1-2 dependencies, or you have dependencies of dependencies
2. When you have open-source dependencies
3. When your project has no dependencies, but you want to implement something that is already available in an open-source library
4. When you are thinking about making your library header-only because it will make it more portable
5. If you are concerned about maintenance time or security

What about modules?
Won’t they fix all our pain?

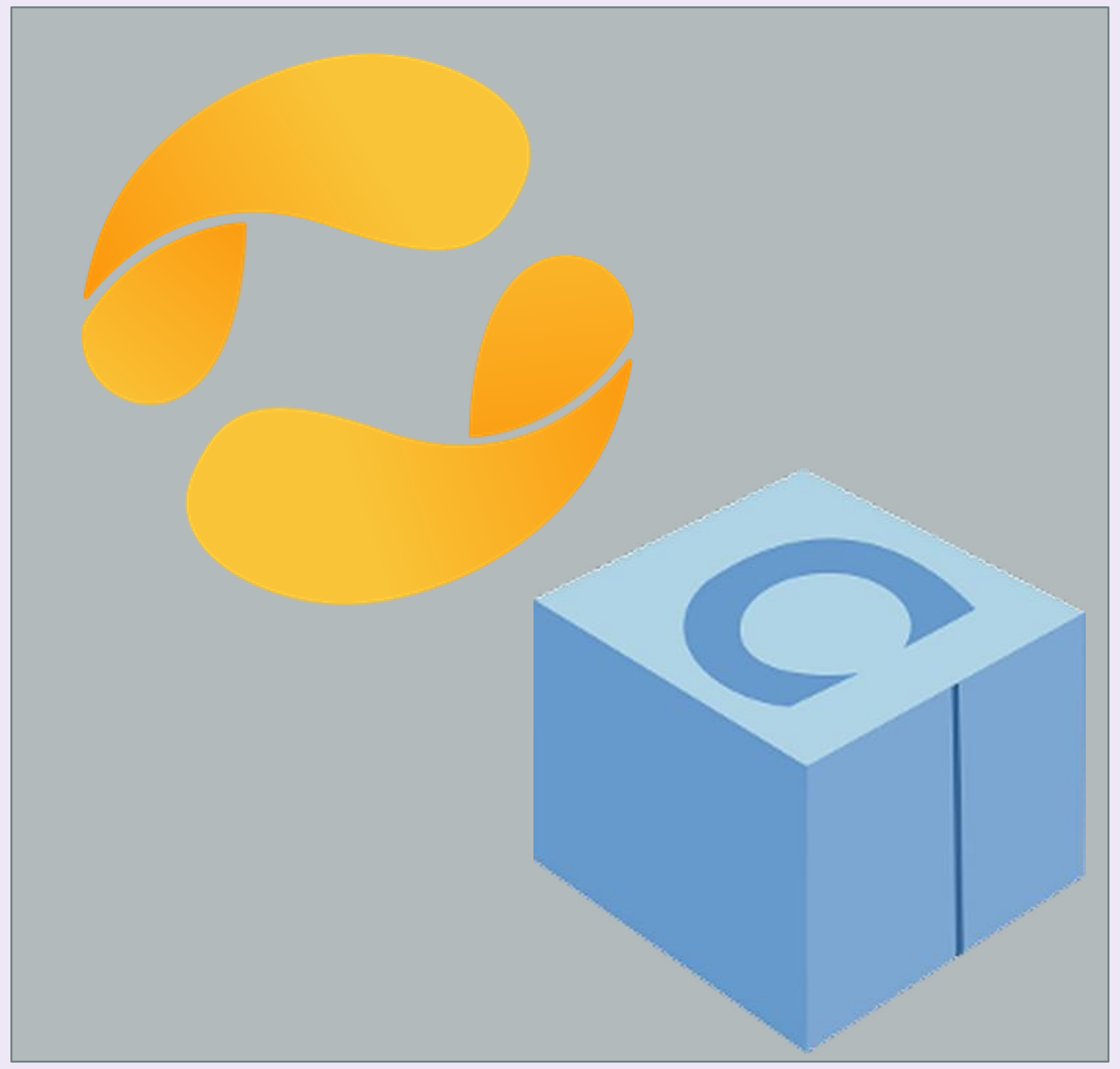
- Issues modules will address:
  - Over-use of headers during preprocessing
  - Macros and “using” declarations (by hiding them away from code outside the module)
  - Some ODR violations (separate translation units in separate modules)
  - Too many files in your repo (no need for source + headers as separate files)
  - Code architecture is clearer with separate logical components

- Issues modules won’t fix:
  - Maintaining ABI stability within a dependency graph
  - Diamond problems
  - Migrating thousands of existing open-source libraries to modules (and even more closed-source ones)

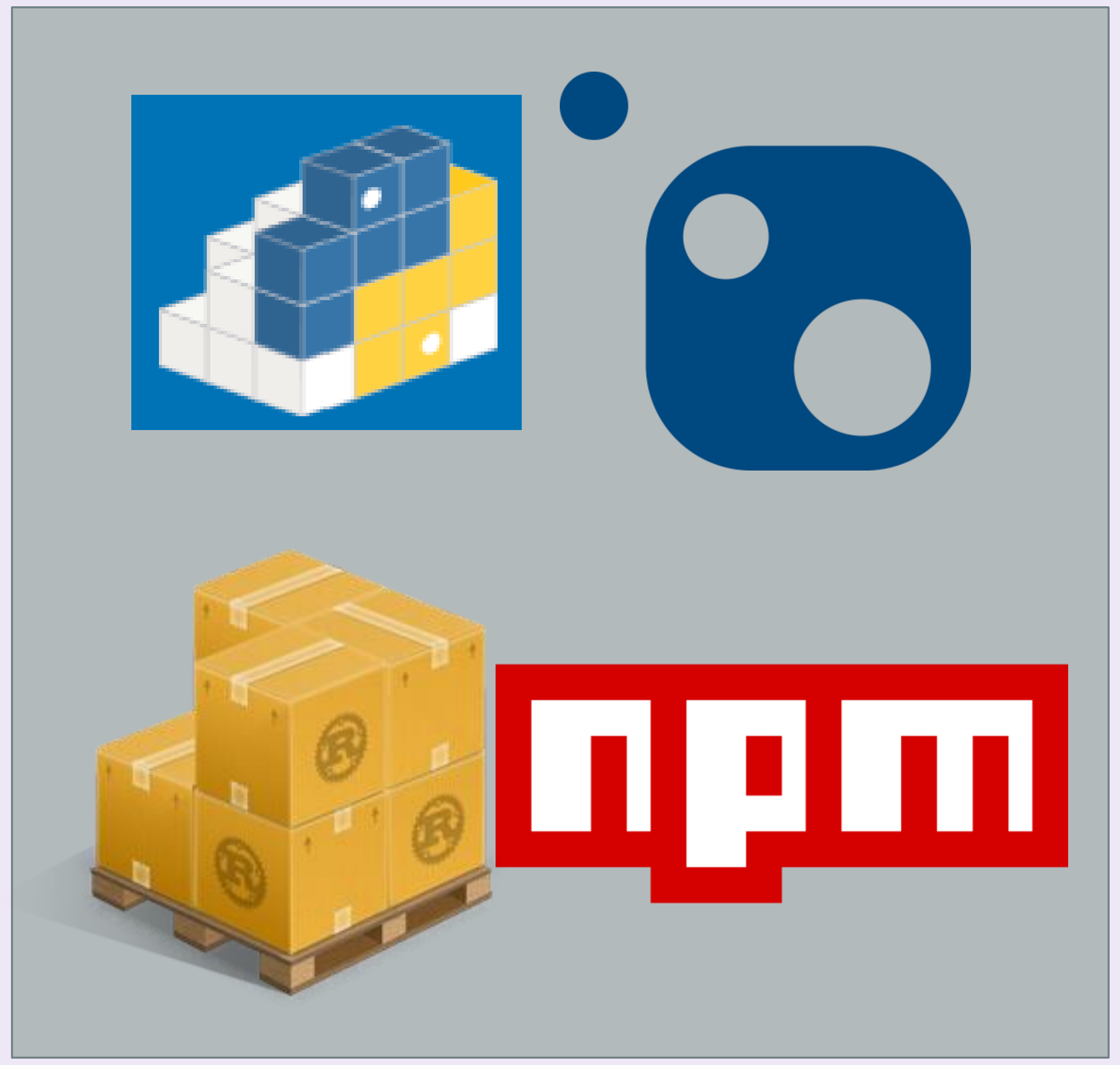
Types of package managers

As used by C++ developers

System package managers



C++ language package managers



Language package managers (non-C++)

System package managers

E.g. apt, yum, rpm, brew, winget, pacman, …

- Designed for a particular operating system (OS) distribution
- Cross-platform projects must have special build logic for different OSes if system package managers are needed
- Packages are installed system-wide (usually elevated)
- Packages are typically acquired as-is, though some system package managers support build from source (e.g. Pacman)
- Not exclusive to C++ packages or even software development; can install apps, tools, and libraries for any workflow
- Don't typically provide first-class integration with build systems – however, since install paths are known defaults, your build system may find packages anyway
- Most popular on Linux

C++ language package managers
E.g. vcpkg, Conan

- Tailored for C++ development with more advanced features
- Have ways to address diamond dependency problems
- Support building packages from source or downloading valid prebuilt binaries
- Support a large variety of open-source libraries out-of-the-box
- Also support private libraries
- Support acquisition of build tools, platform SDKs, debuggers, and other tools needed for a working C++ environment for cross-platform development
- Work across multiple platforms, architectures, and compilers

Non-C++ language package managers
E.g. NuGet, npm, Cargo, pip

- A single-language package manager being repurposed for use with other programming languages
- Useful in limited scenarios (e.g. when a developer primarily uses another programming language than C++, and doesn't want a new package acquisition workflow for C++)
- NuGet (a .NET package manager) is the most common example used by C++ developers (9% in 2022 ISO C++ survey).
- Scenarios NuGet does <u>not</u> address well:
  - ABI violations/diamond problems: no support for building from source, for different compilers, compiler versions, target architectures, target OS. Need a separate package for each configuration.
  - Build systems that are not MSBuild
  - Non-Windows operating systems (while technically possible under Mono or dotnet CLI, it's still not first-class support for C++)
- Since non-C++ language package managers do not address unique C++ requirements, not recommended for C++ except for developers touching C++ assets that have no plans to ever modify them

Which type of package manager should you use for C++ packages?

- If you work in or target primarily one system, and you do not update your dependencies frequently, use a **system package manager**
- If you need a system-specific asset (e.g. a Linux-only graphics library for your video game's Linux port), and that package is not easily available in a C++ language package manager, use a **system package manager**
- If you primarily work in another programming language, use that **language's primary package manager**
- Otherwise, use a C++ language package manager, which helps you resolve ABI issues, diamond problems, offers you access to much wider variety of C++ packages (and updated package versions). Also great for installing per-project dependencies so that other projects can have separate versions of the same dependency.

Reproducible development environments
Getting compilers, debuggers, build systems, graphical assets, …

- Package managers can be extended beyond just library dependencies
- You can package developer tools as well (build tools, debuggers, platform SDKs, static analysis tools, runtime analysis tools, build systems, …)
- System package managers have always offered a wide variety of packages (including dev tools)
- More recently, C++ language package managers are too (including vcpkg and Conan)
- It is important to be able to bootstrap a C++ development environment in an automated and reproducible way
- This ensures consistency between different dev machines and local dev environments and CI
- Use manifests to declare devtool and library dependencies

How to manage dependencies well
Principles to keep in mind

- Be able to build dependencies from source (when necessary)
- Keep your dependencies up-to-date
- Cross-platform should be first class experience
- Make your build environment reproducible
- Do download prebuilt binaries, if they are verified
- Simplify workflow for authoring and publishing dependencies
- Take advantage of existing open-source solutions
- Enforce ABI requirements across all packages, not one at a time
- Use more than one package manager if it improves your productivity

C++ package managers must be accessible and productive for ALL C++ developers

C++ Dependency Management at Microsoft

OLD WAYS

- Different dependencies integrated in different ways
  - git submodules
  - Custom, in-house package managers
  - Built separately, linked to consuming project
  - Open-source treated differently than 1st party dependencies
- Avoiding open-source dependencies altogether
- Every team picking their own solution
- Dependencies get months/years out of date

NEW WAY

- Many teams standardizing on one package manager (vcpkg)
- Many teams share the same dependencies
- Shared compliance process for open-source dependencies
- Dependencies are versioned and easy to upgrade
- 1st party and 3rd dependencies are treated the same way
- Dependencies are tracked in a single manifest file in the team repo

Consider contributing to an open-source package manager

Rather than maintaining an in-house solution

Q & A