USING VCPKG IN ANGER

Martin Hořeňovský

USING VCPKG IN ANGER

Martin Hořeňovský

WHY SHOULD YOU LISTEN TO ME TALK ABOUT VCPKG?

- I started moving us to vcpkg ~3 years ago

- I started moving us to vcpkg ~3 years ago
- We use vcpkg for complex scenarios

- I started moving us to vcpkg ~3 years ago
- We use vcpkg for complex scenarios
    - Multiple interdependent packages

- I started moving us to vcpkg ~3 years ago
- We use vcpkg for complex scenarios
  - Multiple interdependent packages
  - Private packages and vcpkg remote

- I started moving us to vcpkg ~3 years ago
- We use vcpkg for complex scenarios
  - Multiple interdependent packages
  - Private packages and vcpkg remote
  - Custom target triplets

- I started moving us to vcpkg ~3 years ago
- We use vcpkg for complex scenarios
  - Multiple interdependent packages
  - Private packages and vcpkg remote
  - Custom target triplets
  - Varied target platforms

ABOUT THIS TALK

WHAT THIS TALK ISN'T

WHAT THIS TALK ISN'T

- Tutorial for using vcpkg

WHAT THIS TALK ISN'T

- Tutorial for using vcpkg
- Advocacy for Conan

WHAT THIS TALK ISN'T

- Tutorial for using vcpkg
- Advocacy for Conan
- Advocacy against vcpkg

WHAT THIS TALK ISN'T

- Tutorial for using vcpkg
- Advocacy for Conan
- Advocacy against vcpkg
- Advocacy against package managers

WHAT THIS TALK IS

WHAT THIS TALK IS

- Sharing experience from 3 years of vcpkg

WHAT THIS TALK IS

- Sharing experience from 3 years of vcpkg
- More realistic look at using vcpkg in prod

WHAT THIS TALK IS

- Sharing experience from 3 years of vcpkg
- More realistic look at using vcpkg in prod
- Venting

USING VCPKG: QUICK TIPS

manifest mode or classic mode?

manifest mode or classic mode?

Always use vcpkg in manifest mode

Always specify version baseline

```
no-baseline$ ls
CMakeLists.txt vcpkg.json

no-baseline$ cat CMakeLists.txt
cmake_minimum_required(VERSION 3.20)
project(no-baseline LANGUAGES CXX)

no-baseline$ cat vcpkg.json
{
  "$schema": "https://raw.githubusercontent.com/microsoft/vcpkg/mast
  "name": "no-baseline",
  "dependencies": ["fmt"]
}
```

```
no-baseline $ cmake -B build -S . \
     -DCMAKE_TOOLCHAIN_FILE=~/vcpkg/scripts/buildsystems/vcpkg.cmake

-- Running vcpkg install
Detecting compiler hash for triplet x64-linux...
Compiler found: /usr/bin/c++
The following packages will be built and installed:
    fmt:x64-linux -> 10.1.1
  * vcpkg-cmake:x64-linux -> 2023-05-04
  * vcpkg-cmake-config:x64-linux -> 2022-02-06#1
Additional packages (*) will be modified to complete this operation.
...
```

```
no-baseline $ cmake -B build -S . \
     -DCMAKE_TOOLCHAIN_FILE=~/vcpkg-2/scripts/buildsystems/vcpkg.cma
-- Running vcpkg install
Detecting compiler hash for triplet x64-linux...
The following packages will be built and installed:
    fmt:x64-linux -> 11.0.2
  * vcpkg-cmake:x64-linux -> 2024-04-23
  * vcpkg-cmake-config:x64-linux -> 2024-05-23
Additional packages (*) will be modified to complete this operation.
...
```

Always specify version baseline

The vcpkg docs used to point towards `builtin-baseline` as the way to specify baseline version.

The vcpkg docs used to point towards `builtin-baseline` as the way to specify baseline version.

Don't use `builtin-baseline`

```json
{
  "name": "builtin-baseline",
  "builtin-baseline": "68d349964cb4e8da561fd849d9491e6ba11c5681",
  "dependencies": [
    "fmt"
  ]
}
```

```
builtin-baseline$ cmake -B build -S .
   -DCMAKE_TOOLCHAIN_FILE=~/vcpkg/scripts/buildsystems/vcpkg.cmake

-- Running vcpkg install
error: while checking out baseline from commit '68d34996',
       failed to `git show` versions/baseline.json. This may
       be fixed by fetching commits with `git fetch`.
error: git failed with exit code: (128).
fatal: path 'versions/baseline.json' exists on disk, but not
       in '68d34996'
while checking out baseline 68d34996
while loading baseline version for fmt
```

Don't use `builtin-baseline`

```
$ cat vcpkg-configuration.json
{
  "default-registry": {
    "kind": "git",
    "repository": "https://github.com/Microsoft/vcpkg",
    "baseline": "68d349964cb4e8da561fd849d9491e6ba11c5681"
  }
}
```

```
$ cat vcpkg-configuration.json
{
  "default-registry": {
    "kind": "git",
    "repository": "https://github.com/Microsoft/vcpkg",
    "baseline": "68d349964cb4e8da561fd849d9491e6ba11c5681"
  }
}
```

```
{
  "name": "builtin-baseline",
  "dependencies": [
    "fmt"
  ],
  "vcpkg-configuration" : {
    "default-registry": {
      "kind": "git",
      "repository": "https://github.com/Microsoft/vcpkg",
      "baseline": "0e47c1985273129e4d0ee52ff73bed9125555de8"
    }
  }
}
```

Remember that vcpkg resolves version constraints using Minimal Version Selection

Remember that vcpkg resolves version constraints using Minimal Version Selection

This will become relevant later in the talk.

VCPKG ISSUES - PRELUDE

The examples will use 2 projects, `libfoo` and `foosdk`

The examples will use 2 projects, `libfoo` and `foosdk`

`foosdk` will always rely on `libfoo`

The examples will use 2 projects, `libfoo` and `foosdk`

`foosdk` will always rely on `libfoo`

The manifest will contain "baseline" as shorthand for default registry configuration

CONSTRAINTS OVERRIDEN BY BASELINE

The baseline always inserts constraints into resolution

The baseline always inserts constraints into resolution

```
{
  "name": "libfoo",
  "dependencies": [
    { "name": "fmt", "version>=": "9.1.0" }
  ],
  "baseline": "0e47c1985273129e4d0ee52ff73bed9125555de8"
}
```

The baseline always inserts constraints into resolution

```
{
  "name": "libfoo",
  "dependencies": [
    { "name": "fmt", "version>=": "9.1.0" }
  ],
  "baseline": "0e47c1985273129e4d0ee52ff73bed9125555de8"
}
```

```
baseline-constraints$ cmake -B b -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
...

The following packages will be built and installed:
    fmt:x64-linux@10.1.1
  * vcpkg-cmake:x64-linux@2023-05-04
  * vcpkg-cmake-config:x64-linux@2022-02-06#1
```

If you want the dependencies at the version you specified, you have to use old baseline.

If you want the dependencies at the version you specified, you have to use old baseline.

Using old baseline means that you use old versions of transitive dependencies

The other option is using explicit overrides

The other option is using explicit overrides

```json
{
  "name": "libfoo",
  "dependencies": [
    { "name": "fmt", "version>=": "9.1.0" }
  ],
  "overrides": [
    { "name": "fmt", "version": "9.1.0" }
  ]
}
```

The other option is using explicit overrides

```
{
  "name": "libfoo",
  "dependencies": [
    { "name": "fmt", "version>=": "9.1.0" }
  ],
  "overrides": [
    { "name": "fmt", "version": "9.1.0" }
  ]
}
```

```
baseline-constraints$ cmake -B b -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
...

The following packages will be built and installed:
    fmt:x64-linux@9.1.0
  * vcpkg-cmake:x64-linux@2023-05-04
  * vcpkg-cmake-config:x64-linux@2022-02-06#1
```

But overrides don't propagate from dependencies

But overrides don't propagate from dependencies

```
{
  "name": "foosdk",
  "dependencies": ["libfoo"]
}
```

But overrides don't propagate from dependencies

```json
{
  "name": "foosdk",
  "dependencies": ["libfoo"]
}
```

```
foosdk$ cmake -B b -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
...

The following packages will be built and installed:
  * fmt:x64-linux@10.1.1
    libfoo:x64-linux@0.0.0
  * vcpkg-cmake:x64-linux@2023-05-04
  * vcpkg-cmake-config:x64-linux@2022-02-06#1
```

To fix this, you have to add overrides from your dependencies to your own overrides.

```
{
  "name": "foosdk",
  "dependencies": ["libfoo"],
  "overrides": [
    { "name": "fmt", "version": "9.1.0" }
  ]
}
```

This breaks the dependency management abstraction.

This breaks the dependency management abstraction.

You have to manually specify versions of dependencies that you don't know about.

This breaks the dependency management abstraction.

You have to manually specify versions of dependencies that you don't know about.

And so do your own dependees.

BUT WAIT!

BUT WAIT!
THERE IS MORE

Old baseline might not contain a port from the registry

Old baseline might not contain a port from the registry

This will cause the version resolution to fail

```
$ cat ~/vcpkg/versions/b-/boost-cmake.json
{
  "versions": [
    ...
    {
      "git-tree": "bb385ffc8aa74989b8198a777f3181b3a209451a",
      "version": "1.85.0",
      "port-version": 0
    },
    ...
```

```
$ cat ~/vcpkg/versions/b-/boost-cmake.json
{
  "versions": [
    ...
    {
      "git-tree": "bb385ffc8aa74989b8198a777f3181b3a209451a",
      "version": "1.85.0",
      "port-version": 0
    },
    ...
```

```
{
  "name": "baseline-missing-port",
  "dependencies": [
    {
      "name": "boost-cmake",
      "version>=": "1.85.0"
    }
  ],
  "baseline": "16ee2ecb31788c336ace8bb14c21801efb6836e4"
}
```

```
baseline-missing-port$ cmake -B b -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
error: the baseline does not contain an entry for port boost-cmake
-- Running vcpkg install - failed
```

CONFLICT IN VERSION CONSTRAINTS

vcpkg supports only `version>=` constraints vcpkg supports only `version>=` constraints

But resolving two constraints can still fail

```json
{
  "name": "libfoo",
  "dependencies": [
    { "name": "abseil", "version>=": "20211102.1" }
  ]
}
```

```json
{
  "name": "libfoo",
  "dependencies": [
    { "name": "abseil", "version>=": "20211102.1" }
  ]
}
```

```json
{
  "name": "foosdk",
  "dependencies": [
    { "name": "abseil", "version>=": "20230802.1" },
    "libfoo"
  ]
}
```

```
$ cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
Fetching registry information from https://github.com/Microsoft/vcpk
error: version conflict on abseil:x64-linux:
       libfoo required 20211102.1, which cannot be
       compared with the version 20230802.0.

The versions have incomparable schemes:
  abseil@20230802.1 has scheme relaxed
  abseil@20211102.1 has scheme string
```

vcpkg recognizes 4 different version types

- version (relaxed version)
- version-semver
- version-date
- version-string

vcpkg only compares versions within the same domain vcpkg only compares versions within the same domain

version-strings are incomparable by definition*

In Jan 2022 I convinced Billy that version-semver and version-relaxed are the same domain

In Jan 2022 I convinced Billy that version-semver and version-relaxed are the same domain

But vcpkg does not accept date-like versions `2021.01.01` as a relaxed version.

Remember: baseline's constraints are **always** inserted.

Remember: baseline's constraints are **always** inserted.

```
{
  "name": "libfoo",
  "baseline": "1d206dae085048388c7034eff0058899fedcb1ba",
  "dependencies": [
    { "name": "abseil", "version>=": "2021-03-24" }
  ]
}
```

```
foolib$ cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
Fetching registry information from https://github.com/Microsoft/vcpk
error: version conflict on abseil:x64-linux:
       foolib required 2021-03-24, which cannot be compared
       with the baseline version 20230802.1.

The versions have incomparable schemes:
  abseil@20230802.1 has scheme relaxed
  abseil@2021-03-24 has scheme date
```

INCONSISTENT VERSION RESOLUTION

```json
{
  "name": "libfoo",
  "baseline": "d090b933e923c0a69950423ae81fb9488d2d7bff",
  "dependencies": [
    {
      "name": "boost-circular-buffer",
      "version>=": "1.80.0"
    }
  ]
}
```

```
boost-mismatch> cmake -B build -S . ^
            -DCMAKE_TOOLCHAIN_FILE=c:/ubuntu/vcpkg/scripts/buildsyst

-- Running vcpkg install
Detecting compiler hash for triplet x64-windows...
The following packages will be built and installed:
  * boost-assert:x64-windows -> 1.79.0
    boost-circular-buffer:x64-windows -> 1.80.0
  * boost-concept-check:x64-windows -> 1.79.0
  * boost-config:x64-windows -> 1.79.0
  * boost-core:x64-windows -> 1.79.0
  ...
  ...
```

I originally found this issue with Boost packages,

I originally found this issue with Boost packages,

and I got it partially fixed for Boost 1.80#1.

I originally found this issue with Boost packages,

and I got it partially fixed for Boost 1.80#1.

At least for the trivial case.

Boost packages now use version constraints when referencing other Boost packages

Boost packages now use version constraints when referencing other Boost packages

It is still easy to break by mistake though

```json
{
  "name": "libfoo",
  "dependencies": [
    { "name": "boost-icl", "version>=": "1.82.0" }
  ]
}
```

```json
{
  "name": "libfoo",
  "dependencies": [
    { "name": "boost-icl", "version>=": "1.82.0" }
  ]
}
```

```json
{
  "name": "foosdk",
  "baseline": "d090b933e923c0a69950423ae81fb9488d2d7bff",
  "dependencies": [
    { "name": "boost-circular-buffer", "version>=": "1.81.0" }
    "libfoo",
  ]
}
```

```
boost-mismatch-2> cmake -B build -S . ^
               -DCMAKE_TOOLCHAIN_FILE=c:/ubuntu/vcpkg/scripts/builds

-- Running vcpkg install
Detecting compiler hash for triplet x64-windows...
The following packages will be built and installed:
    ...
  * boost-bind:x64-windows -> 1.82.0
  * boost-build:x64-windows -> 1.82.0
    boost-circular-buffer:x64-windows -> 1.81.0
    ...
```

This issue applies to any "split package" of monolithic project

This issue applies to any "split package" of monolithic project

e.g. Qt or KDE

vcpkg mention this issue with Boost in their docs.

vcpkg mention this issue with Boost in their docs.

This is the suggested workaround:

vcpkg mention this issue with Boost in their docs.

This is the suggested workaround:

```json
{
  "default-registry": {
    "kind": "git",
    "repository": "https://github.com/Microsoft/vcpkg",
    "baseline": "3265c187c74914aa5569b75355badebfdbab7987"
  },
  "registries": [
    {
      "kind": "git",
      "repository": "https://github.com/Microsoft/vcpkg",
      "baseline": "8424da584e59e05956913bf96f87654aa3096c7e",
      "packages": [ "boost*", "boost-*"]
    }
  ]
}
```

For newer versions of Boost, you have to update the pattern to include "vcpkg-boost".

OVERBUILDING TRANSITIVE DEPENDENCIES

vcpkg has a (mis)feature called "default features" vcpkg has a (mis)feature called "default features"

This is a set of features of port to build by default

Default features often lead to large package

Default features often lead to large package

```
{
  "name": "libfoo",
  "dependencies": [
    {
      "name": "opencv4",
      "features": ["png"],
      "version>=": "4.8.0"
    }
  ]
}
```

```
libfoo$ cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
Detecting compiler hash for triplet x64-linux...
The following packages will be built and installed:
  * at-spi2-atk:x64-linux -> 2.38.0
  * at-spi2-core:x64-linux -> 2.44.1#2
  * atk:x64-linux -> 2.38.0#5
  * brotli:x64-linux -> 1.0.9#5
  * bzip2[core,tool]:x64-linux -> 1.0.8#4
  * cairo[core,fontconfig,freetype,gobject,x11]:x64-linux -> 1.17.8
  * dirent:x64-linux -> 1.23.2#2
  * egl-registry:x64-linux -> 2022-09-20
  * expat:x64-linux -> 2.5.0#3
  * fontconfig:x64-linux -> 2.14.2
  * freetype[brotli,bzip2,core,png,zlib]:x64-linux -> 2.12.1#3
  ...
Installing 1/44 vcpkg-cmake-config:x64-linux...
```

Thankfully, default features can be disabled

```json
{
  "name": "libfoo",
  "dependencies": [
    {
      "name": "opencv4",
      "default-features": false,
      "features": ["png"],
      "version>=": "4.8.0"
    }
  ]
}
```

```
libfoo$ cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
Detecting compiler hash for triplet x64-linux...
The following packages will be built and installed:
  * libpng:x64-linux -> 1.6.39#1
    opencv4[core,png]:x64-linux -> 4.8.0
  * vcpkg-cmake:x64-linux -> 2022-12-22
  * vcpkg-cmake-config:x64-linux -> 2022-02-06#1
  * vcpkg-get-python-packages:x64-linux -> 2022-06-30
  * zlib:x64-linux -> 1.2.13

Installing 1/6 vcpkg-cmake:x64-linux...
```

But vcpkg can (and will) happily ignore that

```json
{
  "name": "libfoo",
  "dependencies": [
    {
      "name": "opencv4",
      "default-features": false,
      "features": ["png"],
      "version>=": "4.8.0"
    }
  ]
}
```

```json
{
  "name": "libfoo",
  "dependencies": [
    {
      "name": "opencv4",
      "default-features": false,
      "features": ["png"],
      "version>=": "4.8.0"
    }
  ]
}
```

```json
{
  "name": "foosdk",
  "dependencies": ["libfoo"]
}
```

```
foosdk$ cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=...

-- Running vcpkg install
Detecting compiler hash for triplet x64-linux...
The following packages will be built and installed:
  * at-spi2-atk:x64-linux -> 2.38.0
  * at-spi2-core:x64-linux -> 2.44.1#2
  * atk:x64-linux -> 2.38.0#5
  * brotli:x64-linux -> 1.0.9#5
  * bzip2[core,tool]:x64-linux -> 1.0.8#4
  * cairo[core,fontconfig,freetype,gobject,x11]:x64-linux -> 1.17.8
  * dirent:x64-linux -> 1.23.2#2
  * egl-registry:x64-linux -> 2022-09-20
  * expat:x64-linux -> 2.5.0#3
  * fontconfig:x64-linux -> 2.14.2
  * freetype[brotli,bzip2,core,png,zlib]:x64-linux -> 2.12.1#3
  ...

Installing 1/45 vcpkg-cmake-config:x64-linux...
```

vcpkg flat out ignores the "default-features" option in dependencies vcpkg flat out ignores the "default-features" option in dependencies

A PR to fix this has been open for ~2.5 years

The suggested workaround is simple, and kinda dumb

```
{
  "name": "foosdk",
  "dependencies": [
    "libfoo",
    {
      "name": "opencv4",
      "default-features": false
    }
  ]
}
```

The suggested workaround is simple, and kinda dumb

```json
{
  "name": "foosdk",
  "dependencies": [
    "libfoo",
    {
      "name": "opencv4",
      "default-features": false
    }
  ]
}
```

No, you can't have this in overrides.

How do you tell whether opencv4 is real dependency?

How do you tell whether opencv4 is real dependency?

```
{
  "$comment": "not a real depepdecy",
  "name": "opencv4",
  "default-features": false
}
```

BUT WAIT!

BUT WAIT!
THERE IS MORE

It can get worse than just building too much

It can get worse than just building too much

ffmpeg's default build used to be GPL licensed

It can get worse than just building too much

ffmpeg's default build used to be GPL licensed

If your dependency used ffmpeg, you were silently linking to GPL code

It can get worse than just building too much

ffmpeg's default build used to be GPL licensed

If your dependency used ffmpeg, you were silently linking to GPL code

UNVERSIONED HELPER SCRIPTS

Some of the vcpkg's helper scripts are unversioned

Some of the vcpkg's helper scripts are unversioned

They are used from your local vcpkg checkout

Your build can change even without changing the vcpkg binary, the baseline or the version specs.

Your build can change even without changing the vcpkg binary, the baseline or the version specs.

Yes, I have had to debug this when *some* of our CI machines started randomly failing.

Your build can change even without changing the vcpkg binary, the baseline or the version specs.

Yes, I have had to debug this when *some* of our CI machines started randomly failing.

Don't let your CI machines update vcpkg implicitly.

I found a fun example yesterday:

I found a fun example yesterday:

```
$ cd ~/vcpkg
$ git checkout 826ebc235f28261dbf150fe558f6fc00b4783a3e
$ cd ~/vcpkg-examples/version-type-mismatch/foosdk
$ cmake -B build -S . \
    -DCMAKE_TOOLCHAIN_FILE=~/vcpkg/scripts/buildsystems/vcpkg.cmake

-- Running vcpkg install
error: In manifest mode, `vcpkg install` does not support
       individual package arguments.
To install additional packages, edit vcpkg.json and then run
`vcpkg install` without any package arguments.
...
```

VCPKG: GOOD PARTS

It is easy to get started

```
$ git checkout https://github.com/Microsoft/vcpkg
$ cd vcpkg && ./bootstrap-vcpkg.sh
```

It is easy to get started

```
$ git checkout https://github.com/Microsoft/vcpkg
$ cd vcpkg && ./bootstrap-vcpkg.sh
```

```
$ mkdir new-project
$ ~/vcpkg/vcpkg new --application
$ ls
vcpkg-configuration.json  vcpkg.json
```

It is easy to get started

```
$ git checkout https://github.com/Microsoft/vcpkg
$ cd vcpkg && ./bootstrap-vcpkg.sh
```

```
$ mkdir new-project
$ ~/vcpkg/vcpkg new --application
$ ls
vcpkg-configuration.json  vcpkg.json
```

```
$ ~/vcpkg/vcpkg add port fmt
Succeeded in adding ports to vcpkg.json file.
$ cat vcpkg.json
{
  "dependencies": [
    "fmt"
  ]
}
```

The baseline is a curated set of packages & package versions that are tested to build together.

The baseline is a curated set of packages & package versions that are tested to build together.

You can expect both x64 and Arm builds to work for common platforms.

vcpkg exposes the package's build system vcpkg exposes the package's build system

The exported targets will be the same, whether you installed the package manually or through vcpkg

Triplet files let you customize the build heavily

Triplet files let you customize the build heavily

```cmake
# Custom triplet for x64 Linux.
set(VCPKG_TARGET_ARCHITECTURE x64)
set(VCPKG_CRT_LINKAGE dynamic)
set(VCPKG_CMAKE_SYSTEM_NAME Linux)
```

Do you want to link everything statically?

Do you want to link everything statically?

```
set(VCPKG_LIBRARY_LINKAGE static)
```

Do you want to link everything statically?

```
set(VCPKG_LIBRARY_LINKAGE static)
```

But you need to comply with FFmpeg's LGPL license!

Do you want to link everything statically?

```
set(VCPKG_LIBRARY_LINKAGE static)
```

But you need to comply with FFmpeg's LGPL license!

```
if (${PORT} MATCHES "ffmpeg")
  set(VCPKG_LIBRARY_LINKAGE dynamic)
else ()
  set(VCPKG_LIBRARY_LINKAGE static)
endif ()
```

Do you want to target at least Zen v3 machines?

Do you want to target at least Zen v3 machines?

```
set(VCPKG_C_FLAGS "${VCPKG_C_FLAGS} -march=znver3")
set(VCPKG_CXX_FLAGS "${VCPKG_CXX_FLAGS} -march=znver3")
```

- Control symbol visibility across your dependencies?

- Control symbol visibility across your dependencies?
- Compile dependencies with sanitizers to avoid FPs?

- Control symbol visibility across your dependencies?
- Compile dependencies with sanitizers to avoid FPs?
- Keep consistent language standard everywhere?

- Control symbol visibility across your dependencies?
- Compile dependencies with sanitizers to avoid FPs?
- Keep consistent language standard everywhere?
- Compile with LTO across all of your dependencies?

- Control symbol visibility across your dependencies?
- Compile dependencies with sanitizers to avoid FPs?
- Keep consistent language standard everywhere?
- Compile with LTO across all of your dependencies?
- …

Package metadata are non-executable

LICENSING

Do you know the licenses of all your dependencies?

Do you know the licenses of all your dependencies?

Can you audit them automatically?

Do you know the licenses of all your dependencies?

Can you audit them automatically?

Do you comply with MIT?

vcpkg makes handling licenses easy

vcpkg makes handling licenses ~~easy~~ reasonably hard

vcpkg makes handling licenses ~~easy~~ reasonably hard

vcpkg copies the port's license file into the buildtree

vcpkg makes handling licenses ~~easy~~ reasonably hard

vcpkg copies the port's license file into the buildtree

It also tries to provide SPDX-compliant summary

CONCLUSION

DON'T be scared of vcpkg

DON'T be scared of vcpkg

DO be careful when using it

vcpkg is functional package manager with lot of warts

vcpkg is functional package manager with lot of warts

Some of these warts can cause lot of problems

vcpkg is functional package manager with lot of warts

Some of these warts can cause lot of problems

And it is good to be aware of them

But vcpkg also gets lot of things right

and it has served us well

If you already have working Conan setup, keep it

If you already have working Conan setup, keep it

If you are not using a package manager, try vcpkg

If you already have working Conan setup, keep it

If you are not using a package manager, try vcpkg

No, `FetchContent` is not a package manager.

THE END

https://github.com/horenmar/examples-using-vcpkg-in-anger