VCPKG

C++ LIBRARY DEPENDENCY HANDLING

Salvation from dependency hell?

KEVIN LEONARDIC

C++ USERGROUP AACHEN

04.02.2026

Website · GitHub · LinkedIn

GOALS FOR TODAY

- Provide an overview over what is possible
- Concrete examples on how to get started
- Spread the gospel of somewhat well-defined build dependencies

PACKAGE MANAGER DISCLAIMER

I will not be talking about:

- conan
- CPM
- Meson
- Bazel
- Nix{,OS} ❤️‍🩹
- ... your favorite build / package manager that I know nothing about
  - Tell me all about it afterwards!

MOTIVATION

I have some code using OpenSSL

```
int main(int argc, char *argv[])
{
    std::unique_ptr<EVP_MD_CTX, openssl_destroyer> ctx(EVP_MD_CTX_create());
    assert(static_cast<bool>(ctx));
    std::unique_ptr<const EVP_MD, openssl_destroyer> md(EVP_MD_fetch(nullptr,
    assert(static_cast<bool>(md));

    {
        int ret = EVP_DigestInit_ex(ctx.get(), md.get(), nullptr);
        check_status(ret);
    }
// ...
```

Let's compile it!

```
g++ sha256.cpp -o sha256
```

200ms later...

```
/usr/bin/ld: /tmp/ccHzkvEW.o: in function `check_status(int)':
sha256.cpp:(.text+0x170): undefined reference to `ERR_error_string_n'
/usr/bin/ld: sha256.cpp:(.text+0x196): undefined reference to `ERR_get_error'
/usr/bin/ld: /tmp/ccHzkvEW.o: in function `main':
sha256.cpp:(.text+0x1d7): undefined reference to `EVP_MD_CTX_new'
/usr/bin/ld: sha256.cpp:(.text+0x226): undefined reference to `EVP_MD_fetch'
/usr/bin/ld: sha256.cpp:(.text+0x28c): undefined reference to `EVP_DigestInit_
/usr/bin/ld: sha256.cpp:(.text+0x345): undefined reference to `EVP_DigestUpdat
/usr/bin/ld: sha256.cpp:(.text+0x3d2): undefined reference to `EVP_DigestFinal
/usr/bin/ld: sha256.cpp:(.text+0x41d): undefined reference to `OPENSSL_buf2hex
[...]
collect2: error: ld returned 1 exit status
```

Good thing my distribution provides openssl-devel

```
$ g++ sha256.cpp -lssl -lcrypto -o sha256
$ ./sha256 sha256.cpp
34BD106FDF2D17155C072D46D425423C62F243D92792A70046A2559C44619533  sha256.cpp
```

But how to...

- build it on Windows?
- build it on macOS?
- build it for Android?
- build it for the browser?

Enter CMake

```
cmake_minimum_required(VERSION 3.18)

project(Hash LANGUAGES CXX)

find_package(OpenSSL REQUIRED)

add_executable(sha256 sha256.cpp)

target_link_libraries(sha256
                      PRIVATE
                          OpenSSL::SSL OpenSSL::Crypto
)
```

One of your colleagues wants to add serialization

```
1  cmake_minimum_required(VERSION 3.18)
2
3  project(Hash LANGUAGES CXX)
4
5  find_package(OpenSSL REQUIRED)
6  find_package(Protobuf REQUIRED)
7
8  add_executable(sha256 sha256.cpp)
9
10 target_link_libraries(sha256
11                        PRIVATE
12                            OpenSSL::SSL OpenSSL::Crypto
13                            protobuf::libprotobuf-lite
14 )
```

COMMON APPROACHES

for C++ dependency management

... by platform

- Linux: System package manager
- macOS: Homebrew
- Windows: Bundling in sourcetree + binary artifacts
- Android: Bundling in sourcetree *1
- Web: Bundling in sourcetree *1

*1: I have never personally tried this approach

So what do we ideally want?

- Build CMake project on Linux, macOS and Windows
  - ... using the exact same dependencies built from source
  - ... linked statically and or dynamically
- Coupling of code and dependencies
  - ... for reproducible CI builds
  - ... for reproducible development environments

ENTER VCPKG

OPERATING MODES

- Classic mode
  - Works like your distribution package manager
    - ... or brew or chocolatey
- Manifest mode
  - Declaratively specify dependencies for your workspace
    - ... you generally want to use this

ENTER VCPKG

OPERATING MODES

- Classic mode

```
$ vcpkg install openssl protobuf
```

Installs packages in vcpkg directory

- Manifest mode

```
# vcpkg.json
{
    "dependencies": [
        "openssl",
        "protobuf"
    ]
}
```

Installs packages in your directory of choice

ENTER VCPKG

INSTALLATION

```
$ git clone https://github.com/microsoft/vcpkg.git
$ cd vcpkg
$ ./bootstrap-vcpkg.sh -disableMetrics
```

... on Windows

```
PS> .\bootstrap-vcpkg.bat -disableMetrics
```

Yeah, "phoning home" by default... did I mention vcpkg is maintained by Microsoft?

FIRST STEPS

In the following it is assumed that vcpkg is available in PATH and that VCPKG_ROOT points to the installation path, e.g. like the environment resulting from this session:

```
$ git clone https://github.com/microsoft/vcpkg.git
$ cd vcpkg
$ export VCPKG_ROOT=$(pwd -p)
$ export PATH=$PATH:$VCPKG_ROOT
```

All the examples are available at:

github.com/kevle/vcpkg_sha256_example

FIRST STEPS

Recall our earlier manifest example `0_basic`:

```
// 0_basic/vcpkg.json
{
    "dependencies": [
        "openssl",
        "protobuf"
    ]
}
```

FIRST STEPS

Now we can actually build our libraries with vcpkg!

```
$ cd 0_basic
$ vcpkg install
```

The output will look something like this:

```
Detecting compiler hash for triplet x64-linux...
Compiler found: /usr/bin/c++
The following packages will be built and installed:
  * abseil:x64-linux@20250814.1
    openssl:x64-linux@3.6.0#3
    protobuf:x64-linux@5.29.5#3
  * utf8-range:x64-linux@5.29.5
  * vcpkg-cmake:x64-linux@2024-04-23
  * vcpkg-cmake-config:x64-linux@2024-05-23
  * vcpkg-cmake-get-vars:x64-linux@2025-05-29
```

FIRST STEPS

After compilation and installation, our packages will be in `vcpkg_installed/$VCPKG_TRIPLET`:

```
$ ls vcpkg_installed/x64-linux
debug/  etc/  include/  lib/  share/  tools/
$ ls -1 vcpkg_installed/x64-linux/lib/lib{crypto,ssl,proto}*
[...]/libcrypto.a
[...]/libprotobuf.a
[...]/libprotobuf-lite.a
[...]/libprotoc.a
[...]/libssl.a
```

We get a Unix-style installation prefix layout and can find all of our static library archives.

FIRST STEPS

TERMINOLOGY

Great, but wait... what even is a triplet? And what terms are important to know when using vcpkg?

TRIPLETS

- In vcpkg, triplets specify a configuration set for libraries
  - The triplet specifies whether a library is built for static or dynamic linking
- You can also embed toolchains
  - Either directly or through VCPKG_CHAINLOAD_TOOLCHAIN_FILE
- Differentiation between target and host triplets
  - Host triplets => Configuration for build machine
  - Targets triplets => Configuration for target machine

PORTS

- "Library recipes"
- Usually one port per library
  - For example openssl and protobuf!

REGISTRIES

- Collection of ports and meta data
- Either stored in git or in the filesystem
- You can use multiple registries for different ports

BACK TO THE EXAMPLES

We have already seen how to build our libraries

```
$ cd 0_basic
$ vcpkg install
```

but since we are using CMake anyway, it can be even more convenient

```
$ cd 0_basic
$ cmake -DCMAKE_TOOLCHAIN_FILE=$VCPKG_ROOT/scripts/buildsystems/vcpkg.cmake \
        -Bbuild
$ cmake --build build
[ 25%] Running cpp protocol buffer compiler on sha256.proto
[ 50%] Building CXX object CMakeFiles/sha256.dir/sha256.cpp.o
[ 75%] Building CXX object CMakeFiles/sha256.dir/sha256.pb.cc.o
[100%] Linking CXX executable sha256
[100%] Built target sha256
```

LIVE DEMO TIME! 

SALVATION FROM DEPENDENCY HELL?

It's up to you to find out for yourself now!

BONUS STUFF

ASSET CACHING

Self-serve source file bundles

BINARY CACHING

Distribute build artifacts generated by your CI

CUSTOM REPOSITORIES

Use your own port registries, filesystem and / or git based

EXPORT

Bundle artifacts selfcontained in a tar or zip archive