Leonardo Lima
Aug 10, 2020 · 3 min read · Member-only · Listen

CPM: An Awesome Dependency Manager for C++ with CMake

C++ doesn't have a default package manager, making the development with external libraries complex, especially if you're making a multiplatform system. Because of this absence, the community has created some strategies to work around and there are even some pretty consolidated package managers out there, such as Conan and Vcpkg, but none of them are simple and easy to use like CPM (no, we aren't talking about this one).

CPM is the new kid on the block, its first version was released on April 2019, it's a CMake script that gives CMake the capacity to make version control, caching, and install the dependencies of your project, created by Lars Melchior.

Usually, on a modern C++ and CMake project, the libraries are installed on the system and **find_package()** is called to load the library configuration, forcing every developer working on the project to have the same environment (sometimes with different tools' versions, giving us a nasty headache).

CPM has a simple syntax and it's pretty straightforward, making our lives as devs way easier. We call CPMAddPackage to install an external library (or CPMFindPackage to search the system, and, if it's not found, install it). There are two ways to download a library: through a Github repository or a link. Let's use Nlohmann's JSON library as an example:

```
CPMAddPackage(
    NAME nlohmann_json
    GITHUB_REPOSITORY nlohmann/json
    VERSION 3.6.1)
```

CPMAddPackage.cmake hosted with ❤ by GitHub view raw

Or through a link:

```
CPMAddPackage(
    NAME nlohmann_json
    VERSION 3.6.1
    URL https://github.com/nlohmann/json/releases/download/v3.6.1/include.zip
    URL_HASH SHA256=69cc88207ce91347ea530b227ff0776db82dcb8de6704e1a3d74f4841bc651cf)

if(nlohmann_json_ADDED)
    add_library(nlohmann_json INTERFACE)
    target_include_directories(nlohmann_json INTERFACE ${nlohmann_json_SOURCE_DIR})
endif()
```

CPMAddPackageLink.cmake hosted with ❤ by GitHub view raw

(You will need to use add_library() and target_include_directories() if you aren't using the Github feature).

All right, to understand the simplicity and ease of CPM let's create a small project and do a quick example using spdlog, made by Gabi Melman.
This is how the source tree will look like:

```
src
|___main.cpp
CMakeLists.txt
```

This is our CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.18)

project(cpm_example)

set(CMAKE_CXX_STANDARD 17)

add_executable(cpm_example src/main.cpp)
```

CMakeMinumum.cmake hosted with ❤ by GitHub view raw

First, we have to download CPM's script on our project and include it on CMake. There are two ways to do this. The first one is manually downloading it with the following commands:

```
$ mkdir -p cmake

$ wget -O cmake/CPM.cmake
https://github.com/TheLartians/CPM.cmake/releases/latest/download/CPM.cmake
```

Then adding the line **include(cmake/CPM.cmake)** on CMakeLists.txt.

The second way is adding the following snippet to CMake, which will check if CPM exists everytime the project builds (and download it if it doesn't exist):

```
set(CPM_DOWNLOAD_VERSION 0.27.2)
set(CPM_DOWNLOAD_LOCATION "${CMAKE_BINARY_DIR}/cmake/CPM_${CPM_DOWNLOAD_VERSION}.cmake")

if(NOT (EXISTS ${CPM_DOWNLOAD_LOCATION}))
    message(STATUS "Downloading CPM.cmake")
    file(DOWNLOAD https://github.com/TheLartians/CPM.cmake/releases/download/v${CPM_DOWNLOA
endif()

include(${CPM_DOWNLOAD_LOCATION})
```

InstallCPM.cmake hosted with ❤ by GitHub view raw

Then, our source tree will look like this:

```
build
|___cmake
    |___CPM.cmake
src
|___main.cpp
CmakeListst.txt
```

To implement spdlog to our project, just add the following snippet to CMakeLists.txt, after including CPM:

```
CPMAddPackage(
    NAME spdlog
    GITHUB_REPOSITORY gabime/spdlog
    VERSION 1.7.0)
```

CPMAddPackageSpdlog.cmake hosted with ❤ by GitHub view raw

Then we have to link the library to the project:

```
target_link_libraries(cpm_example
    spdlog)
```

TargetLinkLibrariesSpdlog.cmake hosted with ❤ by GitHub view raw

In the end, our CMakeLists.txt should look like this:

```
cmake_minimum_required(VERSION 3.18)

project(cpm_example)

set(CMAKE_CXX_STANDARD 17)

set(CPM_DOWNLOAD_VERSION 0.27.2)
set(CPM_DOWNLOAD_LOCATION "${CMAKE_BINARY_DIR}/cmake/CPM_${CPM_DOWNLOAD_VERSION}.cmake")

if(NOT (EXISTS ${CPM_DOWNLOAD_LOCATION}))
    message(STATUS "Downloading CPM.cmake")
    file(DOWNLOAD https://github.com/TheLartians/CPM.cmake/releases/download/v${CPM_DOWNLO
endif()

include(${CPM_DOWNLOAD_LOCATION})

CPMAddPackage(
    NAME spdlog
    GITHUB_REPOSITORY gabime/spdlog
    VERSION 1.7.0)

add_executable(cpm_example src/main.cpp)

target_link_libraries(cpm_example
    spdlog)
```

CMakeCPMExample.cmake hosted with ❤ by GitHub view raw

Now in our main, let's use spdlog to output a "Hello World!":

```
#include <spdlog/spdlog.h>

int main() {
  spdlog::info("Hello world!");
  return 0;
}
```

SpdlogExample.cpp hosted with ❤ by GitHub view raw

And that's it, now we have to build the project and CPM will do all the hard work of downloading and showing CMake the path of our dependencies. So let's create a build directory, build the project, and run it:

```
$ mkdir -p build

$ cd build

$ cmake ..

$ make -j4

$ ./cpm_example
```

Since CPM is a dependency manager and not a package manager, if you're working on multiple projects that use the same dependencies, you can easily set up different versions of the same library.

The only downside of CPM is that every time you run a **git clean -fdx** or delete the build folder, it will remove the dependencies directory, and you will have to download everything again the next time you run cmake. But CPM lets you use a cache directory, passing the parameter **-DCPM_SOURCE_CACHE** to CMake and specifying a path. Like this:

```
$ cmake -DCPM_SOURCE_CACHE=/tmp/deps ..
```

With just a few lines of CMake code, we can manage our external dependencies of a project, making environment configuration so much easier. CPM gives you a great advantage if you are working with a multiplatform project, CI tools (such as Travis and Appveyor), building inside a container, or any other situation where you have to set up a new environment from zero.