H62PEP Computing - Worksheet 8 - Packaging Your Software

P. Evans and K. Cools

Overview

For now we have left the outputs of the build process inside the build directory (which we typically created as a subdirectory of the source directory). During development this is a perfectly fine mode of operation. As you arrive at the stage where you'd wish to deploy your software to a client, a different approach is required for the following reasons:

- You do not want to deploy the entire build directory which contains, in addition to the executables and libraries, also the intermediary build files and CMake configuration files.
- You want to let the user make decisions about what components to install. This way, basic users can save disk space whereas advanced users and client programmers can choose to install also header files and import libraries.
- The layout of the build directory depends on the build system. To keep your package as portable as possible you'd want to adhere to a more conventional layout that is (almost) the same, regardless the target platform.

Fortunately, installation and packaging is a very simple using the features cmake provides. Consider the following snippet from a typical `CMakeLists.txt`:

```
add_library(maths maths.cpp)

add_executable(calc calc.cpp)
target_link_libraries(calc maths)

add_executable(test1 test1.cpp)
target_link_libraries(test1 maths)

install(TARGETS calc maths
  RUNTIME DESTINATION bin
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib/static)

add_test(test1 test1)
```

Most of the above script you have already encountered. There are statements that define targets and inform CMake what sources and libraries these targets depend on. What is new here is the `install` command. This command allows you to list which targets are part of the install package and what the default subdirectories for the various types of outputs are.

Executables are installed in the `bin` subdirectory. For libraries the situation is slightly more involved because libraries can be either static or dynamic and because Windows and Unix systems deal differently with dynamic libraries (Remember that on Windows a dynamic library comes with an import library that client developers need to link against at link time). The following table summarizes the situation:

Static/Dynamic	Windows	Unix
Static	lib -> lib/static	a -> lib/static
Dynamic	dll -> bin	so -> lib
	lib -> lib/static	

In words: `lib/static` contains the components needed at compile time. Components needed at runtime are stored in `bin` on Windows and in `lib` on Unix.

What are these subdirectories subdirectories of? In case an installer is used (see below) they will be a subdirectory of the installation directory, which is on Windows `C:\Program Files\packagename` and on Linux `/usr/local/packagename`. Alternatively we can ask for the package to be installed by building the corresponding target created during the CMake configuration stage. On Windows installation is done by running

```
C:\...\build>cmake -G "MinGW Makefiles" -D CMAKE_INSTALL_PREFIX="C:/MathsLib-installation-dir" ..
C:\...\build>mingw32-make
C:\...\build>mingw32-make install
C:\...\build>mingw32-make package
```

The CMake variable `CMAKE_INSTALL_PREFIX` determines where the package will be installed upon building the `install` target. You have played around with it last week when you built and installed VTK.

```
C:\...\>cmake -G "Generator you like" -D CMAKE_INSTALL_PREFIX=C:/InstallDirectory/ ..
```

Alternatively, you can ask CMake to generate a package, or install the software directly using the commands:

```
C:\...\build>cmake --build . --target install
C:\...\build>cmake --build . --target package
```

The above additions to `CMakeLists.txt` allow us to install the desired targets from the build directory into an installation directory on the same machine. Even though this seems to be a mere copy-paste operation, this can be a meaningful action. Maybe, in parallel with our work on the package under consideration, you are working on another project with targets that depend on libraries in the former package. The dependent package will look for these dependencies in the standard location, i.e. in the installation directories (e.g. subdirectories of "Program Files").

Packaging on the other hand is gathering all files marked for installation, and putting them together in an installation archive, together with meta-data explaining where everything needs to go on the client's machine. In addition provisions can be supplied that allow to select only certain parts of the package.

Perhaps the simplest installation format is a (compressed) archive. Various formats are available and the choice often depends on what is most typical on the target platform. Well known examples include: zip, tar, tgz,… (the last two are often referred to as tarballs). More advanced options are distribution specific packages that interact well with the target platform's package manager. Most prominent examples in this category are `.deb` on Debian descendants such as Ubuntu and `.msi` on the Windows platform. We will focus on using `.exe` installers generated by NSIS for Windows, and `.zip` files otherwise.

Exercise: Install NSIS on your System if you are working on a Windows machine.

Using the capabilities provided by CMake, configuring packaging is extremely simple. Add the following lines at the end of your `CMakeLists.txt`:

```
if(WIN32)
    set(CPACK_GENERATOR "NSIS")
else()
    set(CPACK_GENERATOR "ZIP")
endif()
include(CPack)
```

That's it folks.

Now when running CMake, a target named `package` is created, which can be build by:

```
C:\...\build>cmake --build . --target package
```

Or:

```
C:\...\build>mingw32-make package
```

On Windows this will result in an NSIS style executable and on Unix in a simple zip-file. On Windows, run the exe; I'm sure you will recognise the resulting install wizard.

Final point

During CMake installation and packaging, only components of the current project are considered. In your final product you will most likely also want to ship the Qt and VTK dll's your executable has runtime dependencies on. CMake provides some support in automatically including these during installation/packaging. Look into the help of CMake modules `InstallRequiredSystemLibraries`, `fixup_bundle`, and `DeployQt5.cmake`. Use Dependency walker (`ldd` on Unix, `otool -L` on MacOS) to assist you in getting this right. Note that in general you should consider the license text included with your dependencies to discover if you are allowed to redistribute the, well, redistributable components of the software. Don't expect to get this right the first time around. Consider this exercise one of the goals you can set yourself to achieve by the end of the project.