Jussi Pakkanen's development blog

Observations on development and related issues

A list of common CMake antipatterns

Posted on 2013/03/26

There has been gradual movement towards CMake in Canonical projects. I have inspected quite a lot of build setups written by many different people and certain antipatterns and inefficiencies seem to pop up again and again. Here is a list of the most common ones.

Clobbering CMAKE_CXX_FLAGS

Very often you see constructs such as these:

```
set(CMAKE_CXX_FLAGS "-Wall -pedantic -Wextra")
```

This seems to be correct, since this command is usually at the top of the top level CMakeLists.txt. The problem is that CMAKE_CXX_FLAGS may have content that comes from outside CMakeLists. As an example, the user might set values with the ccmake configuration tool. The construct above destroys those settings silently. The correct form of the command is this:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -pedantic -Wextra")
```

This preserves the old value.

Adding -g manually

People want to ensure that debugging information is on so they set this compiler flag manually. Often several times in different places around the source tree.

It should not be necessary to do this ever.

CMake has a concept of build identity. There are debug builds, which have debug info and no optimization. There are release builds which don’t have debug info but have optimization. There are relwithdebinfo builds which have both. There is also the default plain type which does not add any optimization or debug flags at all.

The correct way to enable debug is to specify that your build type is either debug or relwithdebinfo. CMake will then take care of adding the proper compiler flags for you. Specifying the build type is simple, just pass -DCMAKE_BUILD_TYPE=debug to CMake when you first invoke it. In day to day development work, that is what you want over 95% of the time so it might be worth it to create a shell alias just for that.

Using libraries without checking

This antipattern shows itself in constructs such as this:

```
target_link_libraries(myexe -lsomesystemlib)
```

CMake will pass the latter one to the compiler command line directly so it will link against the library. Most of the time. If the library does not exist, the end result is a cryptic linker error. The problem is worse still if the compiler in question does not understand the -l syntax for libraries (unfortunately those exist).

The solution is to use find_library and pass the result from that to target_link_libraries. This is a bit more work up front but will make the system more pleasant to use.

Adding header files to target lists

Suppose you have a declaration like this:

```
add_executable(myexe myexe.c myexe.h)
```

In this case myexe.h is entirely superfluous. It can just be dropped. The reason people put that in is probably because they think it is required to make CMake rebuild the target in case the header is changed. That is not necessary. CMake will use the dependency information from Gcc and add this dependency automatically.

The only exception to this rule is when you generate header files as part of your build. Then you should put them in the target file list so CMake knows to generate them before compiling the target.

Using add_dependencies

This one is simple. Say you have code such as this:

```
target_link_libraries(myexe mylibrary)
add_dependencies(myexe mylibrary)
```

The second line is unnecessary. CMake knows there is a dependency between the two just based on the first line. There is no need to say it again, so the second line can be deleted.

Add_dependencies is only required in certain rare and exceptional circumstances.

Invoking make

Sometimes people use custom build steps and as part of those invoke “make sometarget”. This is not very clean on many different levels. First of all, CMake has several different backends, such as Ninja, Eclipse, XCode and others which do not use Make to build. Thus the invocation will fail on those systems. Hardcoding make invocations in your build system prevents other people from using their preferred backends. This is unfortunate as multiple backends are one of the main strengths of CMake.

Second of all, you can invoke targets directly in CMake. Most custom commands have a DEPENDS option that can be used to invoke other targets. That is the preferred way of doing this as it works with all backends.

Assuming in-source builds

Unix developers have decades worth of muscle memory telling them to build their code in-source. This leaks into various place. As an example, test data file may be accessed assuming that the source is built in-tree and that the program is executed in the directory it resides in.

Out-of-source builds provide many benefits (which I’m not going into right now, it could fill its own article). Even if you personally don’t want to use them, many other people will. Making it possible is the polite thing to do.

Inline sed and shell pipelines

Some builds require file manipulation with sed or other such shell tricks. There’s nothing wrong with them as such. The problem comes from embedding them inside CMakeLists command invocations. They should instead be put into their own script files which are then called from CMake. This makes them more easily documentable and testable.

Invoking CMake multiple times

This last piece is not a coding antipattern but a usage antipattern. People seem to run the CMake binary over again after doing changes to their build systems. This is not necessary. For any given build directory, you only ever need to run the cmake binary once: when you first configure your project.

After the first configuration the only command you ever need to run is your build command (be it make or ninja). It will detect changes in the build system, automatically regenerate all necessary files and compile the end result. The user does not need to care. This behaviour is probably residue from being burned several times by Autotools’ maintainer mode. This is understandable, but in CMake this feature will just work.

Errata: 2013/11/7

Since writing this post I have discovered that there is a valid reason for adding header files to targets. Certain IDEs such as Qt Creator and Visual Studio will only display files that belong to a target. Thus headers need to be added in the source list or otherwise they can’t be edited.

A second issue is that apparently MSBuild does not always reload a changed build file. The simple workaround for this is to re-run CMake. However it should be noted that this is an issue in MSBuild and not in CMake. Thanks to Yuri Timenkov for this information.

This entry was posted in **development** and tagged **antipattern**, **cleanliness**, **cmake** by **Jussi Pakkanen**. Bookmark the **permalink [http://voices.canonical.com/jussi.pakkanen/2013/03/26/a-list-of-common-cmake-antipatterns/]** .