KDAB

CMake and Qt

Kevin Funk

July 22

The build system for Qt 6 uses CMake. What are the benefits of CMake, and how can it help transform your build system? Here are the basic facts you need to know to take advantage of CMake for building your Qt applications.

One of the changes you may notice moving from Qt 5.x to Qt 6.x is that the Qt build system was ported from QMake to CMake. QMake was starting to show its age since it was only receiving maintenance fixes and critical updates. It had a number of issues in handling distributed builds, implementing configuration checks, and scaling for large projects. Cross-compilation, something that Qt is known for, was challenging to set up in QMake for all but the simplest cases. Those challenges, among others, caused The Qt Company and Qt project teams to move the build over to CMake.

CMake is an independent and reliable build system (and the de facto standard for C++) with a lot of flexibility to handle the increasing demands that Qt has grown into. If you're still using QMake (or even earlier make systems), you'll want to seriously consider moving to CMake since there are a lot of powerful benefits that it brings to your build. While we only have space to cover the highlights, hopefully you'll get a taste of how well it handles build files and dependency generation – two of the main tasks we need out of our build system. Community support for CMake is another big benefit. With active forums and developer assistance, you'll be able to find and solve your most thorny

CMake makes it much easier to manage dependencies from other modules and libraries, allowing you to detect and use third-party packages on a variety of platforms.

build-time challenges. Good news!

Why is Qt's use of CMake going to be good news for Qt developers? If you're not familiar with CMake, a little introduction is in order.

CMake allows developers to use their preferred tools, it is used in hundreds of projects, and it's an open source tool supported by a huge community. Because CMake directly supports IDE project exports for your choice of IDE, like VS or XCode, interactive debugging on your platform of choice will become much easier. Since CMake support in Microsoft's popular Visual Studio IDE has recently seen significant improvements, it makes CMake a good choice if you're developing on the Windows platform.

CMake also makes it much easier to manage dependencies from other modules and libraries. This is because CMake ships with a lot of ***find modules*** – CMake scripts that allow you to detect and use third-party packages on a variety of platforms.

Last but not least, CMake also has tight integration with source code analysis tools that can be run as part of a regular build, such as Clang-Tidy, cpplint, and cppcheck. Having CMake features and reporting for these tools is especially handy when using continuous integration systems that automatically build your projects.

Now that Qt itself is being built with CMake, we can expect CMake support within Qt to get better than it already is. At the same time, CMake is also automatically getting upstream patches from the Qt Project. (An example is the addition of CMake's multi-config Ninja support feature.) These changes serve the whole C++ ecosystem, not just Qt, which is welcome.

People who build Qt from source should also find things easier to manage. Since the Qt build is driven by CMake exclusively now, there are no extra configuration scripts that need to run before invoking CMake. People who already know CMake will mostly feel at home; it's just like any other big CMake project such as LLVM, OpenCV, or any of the KDE frameworks. Of course, if you're already a CMake user, you can expect Qt support and Qt integrations for CMake to continue to improve.

Bad news?

The selection of build systems supported by Qt is shrinking. Qbs has been deprecated as of December 2019, and QMake won't have any future development except bug fixes. On the bright side, having to maintain and extend three separate build systems diluted the available development talent pool that was addressing Qt's build tools, which left each tool needing a bit of attention. Simplifying things down to one main build system helps focus efforts on a single tool. Not to mention, it will make it easier for new Qt developers to pick up.

Another item to discuss is CMake syntax. CMake has some operational quirks that make it a challenge to learn and its language is not always the most pleasant or obvious. As an example of the latter, let's say you want to spawn parallel builds with one more than the number of processor cores of the build machine.

```
if(CPUS)
  math(EXPR CPUS "${CPUS} + 1")
  set(BLD_FLAGS "${BLD_FLAGS} -j${CPUS}")
endif()
```

Ugh – with a mix of strings and awkward syntax just to do a simple increment, it doesn't look too

CMake makes file manipulation easy: copying files to a network drive, computing and comparing hashes, and uploading packages to a cloud repository are examples.

pretty. It's not very fun for coding up non-trivial logic either, since CMake functions have to use global variables in order to return a value back to the caller. On the other hand, there is so much functionality built into CMake that the majority of build configurations are handled without needing to resort to these features. That's true especially in CMake version 3.12 or greater, since over all the years it has been in development CMake has significantly improved its capability as well as simplified the burden on the user. (Watch Deniz Bahadir's great side-by-side comparison of traditional and modern CMake for some specifics.)

While it might not win awards for orthogonal structure and inherent beauty, the fact that CMake can do cross-platform math and string manipulation at all puts it far ahead of plain makefiles. This can be a life saver to any build master who's been forced to resort to sed scripts or custom utilities to trick make and nmake into doing their bidding. Knowing that you have the ability to delve into more complex logic if need be gives you confidence that no matter how complex the tasks needed by your build process, you'll be able to manage them.

While CMake is definitely Turing-complete, it isn't designed for heavy-duty calculation. You won't find it adhering to in-vogue programming paradigms, and it won't have the safety nets you may have come to expect in modern programming environments. (So if you find yourself tempted to write full programs in it, you're probably doing something wrong.)

That said, it's rare that you'll need to create your own build configurations from scratch anyway. The benefit of being a widely used tool is that most common build problems problems have been already solved. It's rather easy to find an example `CMakeList.txt` structure that can be modified to meet your needs.

Mo' better

If you're starting to get convinced that CMake might be for you, here are a few more benefits you can look forward to.

- Requirements are attached to the targets and are automatically propagated as necessary through the build, which includes any requirements from third party libraries. This makes creating a complex build much less error-prone.
- CMake is aware of modern C++. If you need particular compiler features in your code or are using a particular language standard (like C++11/14/17/20), you can determine this when creating your build instead of waiting for a failure at compile time.
- File manipulation is easy. If you've had troubles bending QMake to your will at the end of a build, you'll welcome CMake's comprehensive file command. Copying files to a network drive, computing and comparing hashes, generating header files, and uploading packages to a cloud repository are similarly trivial.

Libraries under CMake

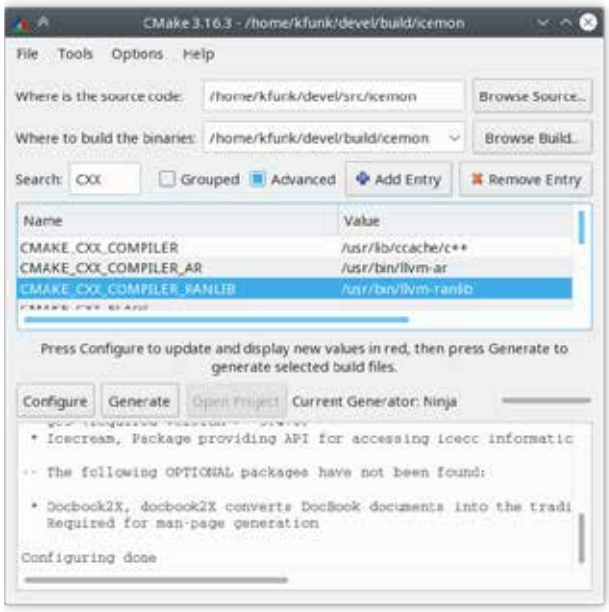
One of the biggest places where CMake is worth its weight in gold is in incorporating external libraries. A big reason is because as a library builder, you can specify not only your library's build requirements but its usage requirements

As a library builder, you can specify your library's usage requirements so that users will automatically use the right dependencies, linker options, or compiler features.

– everything that users of your library will need to incorporate it into their builds. This can go much further than just the include and library paths: it can also include things like additional dependencies, linker options, pre-processor macros, necessary language standard (such as C++14, C++17, or C++20), or required compiler features (like u8 literals, ABI version, or code generation switches).

Because CMake takes care of all the nitty gritty details of incorporating a library, being able to use someone else's code in your project can be as simple as this:

```
find_package(somecoollib)
target_link_libraries(main somecoollib)
```



CMake includes a GUI for configuration; very handy for building debug and release candidates.

Many common libraries that you might need are easily detected using CMake's *find_package* command. The 3.16 release includes built-in *find module* support for 162 commonly used libraries: BZip, Boost, curl, gif, Glut, KDE, Lua, MFC, OpenSSL, Qt, Threads, Wget and many more. (The full list for your particular cmake version can be generated with `cmake --help-module-list`.) Even if your favorite library isn't there, there's a pretty decent chance that someone has already made a CMake *find module* for it – Google it! As a start, Awesome CMake has a great curated list of CMake find modules covering several different areas.

CMake basics

Since most of the Qt community will likely end up embracing CMake sooner or later, you'll need to learn more about it at some point. Let's dive in a little deeper and look at a very simple use case.

CMake's top-level configuration file is always called `CMakeList.txt` and usually resides in the main source directory. There can be `CMakeList.txt` files in any directory that's part of your build, which gives you the option of structuring your project naturally, that is, by using separate folders for the source code of each library you're building in your project. The hello world CMake example in the Qt documentation has the fundamentals needed in a Qt-friendly CMake setup. So, let's start there by dissecting hello world's CMakeList.txt file and explaining it section by section.

```
cmake_minimum_required(VERSION 3.10.0)

project(helloworld)

set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
```

CMake has changed its default policies and syntax over the years, so it's good policy to set the expected CMake version in your CMakeList.txt files to easily find mismatches.

```
find_package(Qt5 COMPONENTS Widgets
    REQUIRED)

add_executable(helloworld
    mainwindow.ui
    mainwindow.cpp
    main.cpp
    resources.qrc
)
target_link_libraries(helloworld
    Qt5::Widgets)
```

Here's the breakdown of each chunk.

This sets the minimum required CMake level to ensure you're not using an older CMake with different behaviors. CMake has changed its default policies and added syntax over the years, so this bit of insurance allows you to get reasonable errors like "Wrong CMake version" rather than obscure problems that might be very difficult to troubleshoot. At the time of writing, CMake 3.10 was the version shipped by Ubuntu 18.04 LTS (long-term release of Ubuntu), so we chose that as a safe bet to be available on developer PCs. Your mileage may vary of course.

This sets the project name for the build. If you want, you can also add a VERSION or DESCRIPTION for your project. LANGUAGES is another keyword that specifies the development language your project supports, but since the default is C and C++ we don't bother to set it explicitly.

This is the magic that enables Qt-specific behavior, namely automatically enabling MOC processing for Qt C++ files, UIC for.ui files, and RCC for.qrc files. Pretty much any Qt project will need these target properties enabled.

This ensures that you automatically add the current source and build directories to the include path.

This pulls in your Qt dependencies, which in the case of helloworld is pretty straightforward. In addition to `Widgets`, you might need to add any other Qt components like Core, GUI, XML, SQL, and so on – just add them after `Widgets`.

Here is the main event: specifying your output target and input source files.

This wraps it all up by linking in our needed Qt library.

This is about as simple a `CMakeList.txt` file as you'll encounter, but it gives you an idea of the basic structure. For something this basic, you can see it's really pretty easy to set up and get going.

CMake is a rich tool for taking control of your entire build process, no matter how specialized, complex, or custom it is.

Summary

We've barely scratched the surface of what CMake can do and how it works. There are many CMake resources available such as online introductions (here's a good one) and references (such as the book written by the original creators). We also provide a CMake training that provides a hands-on education on some of the more popular topics:

- Qt-specific configuration and trouble-shooting
- Cross-compiling and IDE integration
- Debugging techniques when things aren't working as intended
- Properly using private and public interfaces
- Finding and using packages that aren't pre-installed
- Understanding policies and how they change CMake's behavior
- Writing template files and header files, as well as other useful file operations
- Using code generators and synthetic targets

CMake is a rich tool for taking control of your entire build process, no matter how specialized, complex, or custom it is. As a part of the Qt toolkit, we welcome it becoming even more indispensable to Qt builds and a more standardized tool for the Qt community.

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages. Founded in 1999, KDAB has offices throughout North America and Europe.

KDAB

CERTIFIED ISO 9001:2015 COMPANY

www.kdab.com