Qt Virtual Tech Con 2020

Using Modern CMake with Qt

Kevin Funk
kevin.funk@kdab.com

KDAB

About

Kevin Funk

- Senior Software Engineer
- Sales Engineer
- Consultant and trainer at KDAB since 2009
- Qt developer since 2006
- Contributor to KDE/Qt and Free Software

What is CMake?

CMake is a tool to simplify the build process for development projects across different platforms.

CMake automatically generates build systems, such as **Makefiles, Ninja** and **Visual Studio** project files.

Modern CMake?

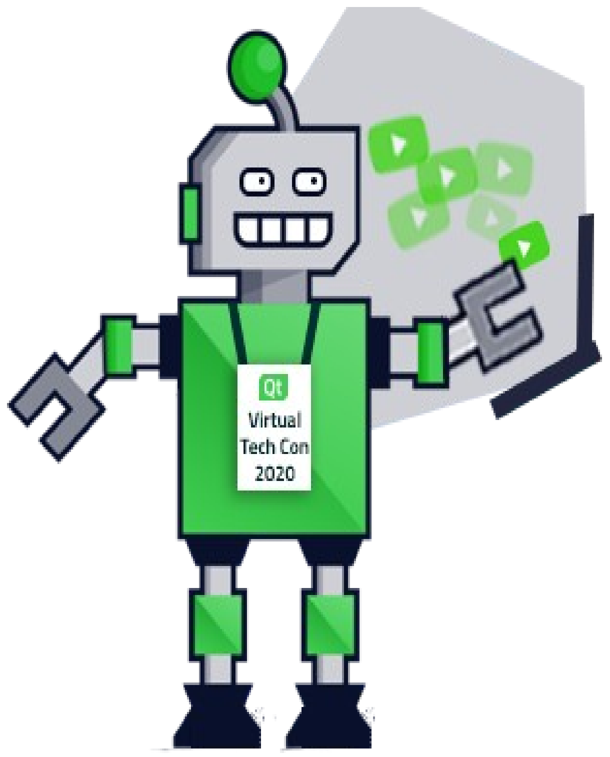
- In a nutshell
  - Code: Forget the commands *add_compile_options*, *include_directories*, *link_directories*, *link_libraries*
    - Instead use their more modern *target_** counterparts
  - Code: Prefer functions over macros
  - Code: Keep internal properties *PRIVATE*
    - E.g. do not propagate -*Werror*
  - Modules: Create and use exported targets
    - Compare *${QT_QTGUI_LIBRARY}* (old) vs. *Qt5::Gui* (modern)
  - ...

Modern CMake: Advantages

- Requirements are attached to the targets
  - Automatically propagated as necessary through the build
  - Makes creating complex builds much less error-prone
- Selecting modern C++ standards (cross-platform) is simple
  - Example: *target_compile_features(myTarget PUBLIC cxx_std_11)*
  - OR: *set(CMAKE_CXX_STANDARD 11)*

    set(CMAKE_CXX_STANDARD_REQUIRED ON)

Getting Started



Getting Started with CMake

```
# Qt with CMake example

cmake_minimum_required(VERSION 3.10.0)

project(helloworld)

set(CMAKE_AUTOMOC ON) ①
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)
set(CMAKE_INCLUDE_CURRENT_DIR ON) ②

find_package(Qt5 COMPONENTS Widgets REQUIRED) ③

add_executable(helloworld
    mainwindow.ui
    mainwindow.cpp
    main.cpp
    resources.qrc
) ④

target_link_libraries(helloworld Qt5::Widgets) ⑤
```

(1) CMake magic that enables Qt-specific behavior
for .moc/.qrc/.ui file types

(2) Ensures adding current source and build directory to the include path

(3) This pulls in Qt dependencies (here: *Qt Widgets* only) and marks them as required

(4) Add an executable target using different source file types

(5) Wrap up: Link to the needed Qt libraries

Running CMake: Command line

```
# In a terminal
# Note these are cross-platform instructions

mkdir build
cd build

cmake .. <additional args> ①

# Build via CMake
cmake --build . ②

# OR build via ANY of ③
make
nmake
ninja
...
```

(1) Run CMake on the source directory (and pass additional arguments if necessary).
Build system files will be generated.

(2) Run this to start the build

(3) Instead of going via CMake you can also invoke the build tool directly

Running CMake: Command line - contd

```
# Finding a specific Qt install

cmake -DCMAKE_PREFIX_PATH=/path/to/qt5-install
  .. <additional args> ①

# OR
export CMAKE_PREFIX_PATH=/path/to/qt5-install
cmake .. <additional args> ②
```

(1) Set the *CMAKE_PREFIX_PATH* CMake variable (via command-line args) to describe additional search paths for *find_package(...)*

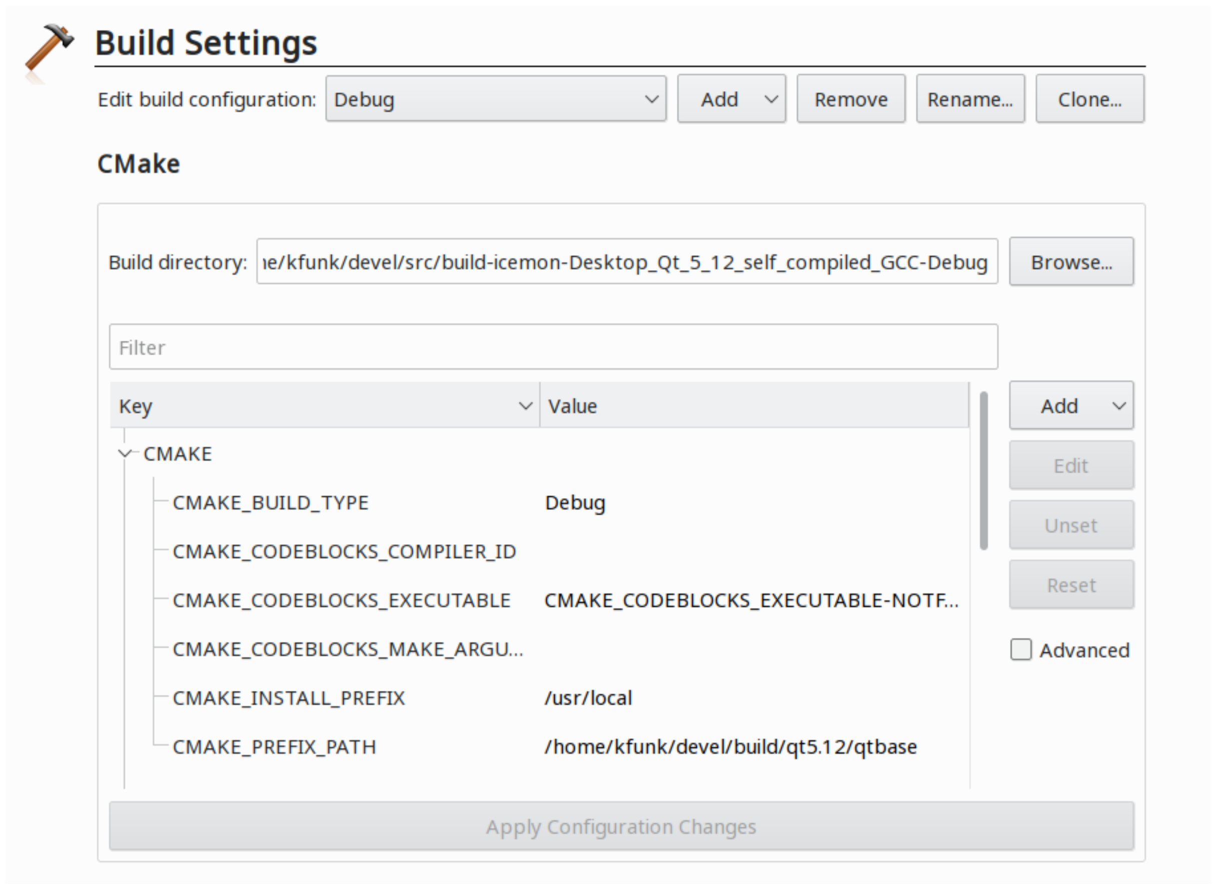
(2) OR set *CMAKE_PREFIX_PATH* as environment variable (works on all supported platforms) before invoking CMake

Running CMake: Command line - contd

```
# In case you'd like to check which Qt version
was found...

$ cd build

$ grep Qt5 CMakeCache.txt

//The directory containing a CMake
configuration file for Qt5Core.
Qt5Core_DIR:PATH=/usr/lib/x86_64-linux-gnu/
cmake/Qt5Core
//The directory containing a CMake
configuration file for Qt5Gui.
Qt5Gui_DIR:PATH=/usr/lib/x86_64-linux-gnu/
cmake/Qt5Gui
…
```

- In case Qt was *NOT* found, CMake will obviously complain

Running CMake: Via QtCreator



Build directory: ıe/kfunk/devel/src/build-icemon-Desktop_Qt_5_12_self_compiled_GCC-Debug

Key	Value
CMAKE	
CMAKE_BUILD_TYPE	Debug
CMAKE_CODEBLOCKS_COMPILER_ID	
CMAKE_CODEBLOCKS_EXECUTABLE	CMAKE_CODEBLOCKS_EXECUTABLE-NOTF...
CMAKE_CODEBLOCKS_MAKE_ARGU...	
CMAKE_INSTALL_PREFIX	/usr/local
CMAKE_PREFIX_PATH	/home/kfunk/devel/build/qt5.12/qtbase

- Simply open the top-level CMakeLists.txt
  - Go to *File* → *Open File or Project*
  - Select CMakeLists.txt, confirm
  - QtCreator will ask you which Qt Kit to use
  - Build the project as usual in QtCreator
- Benefits
  - Built-in CMake configuration GUI
  - Built-in Qt Kit handling
    - Multiple Qt versions in parallel
    - Debug vs. Release builds, etc.

CMake Qt Integration - contd

- With **AUTOMOC/AUTORCC/AUTOUIC**
  - *No* need for
    - *qt5_wrap_cpp(...)*
    - *qt5_wrap_ui(...)*
    - *qt5_add_resources(...)*
- Simplifies CMake code!
- Also leads to faster overall builds(!)

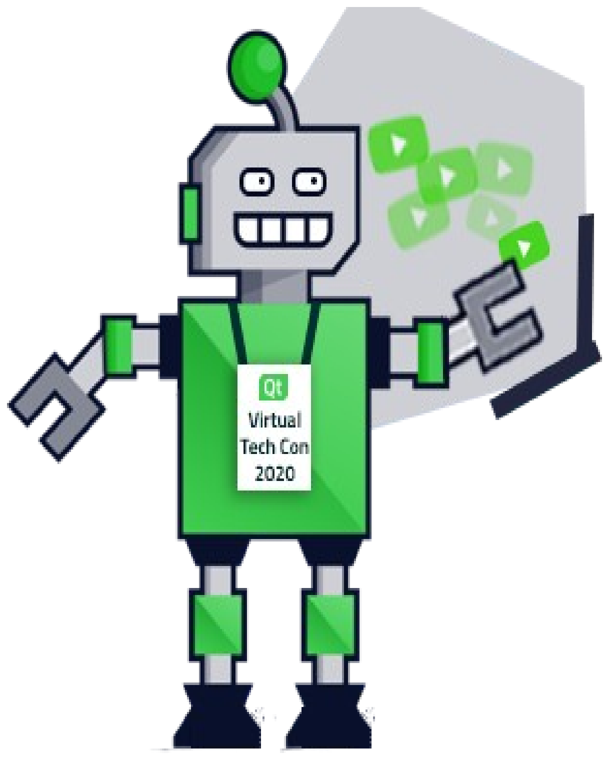
```
$ cat ./ssrc/icemon_autogen/mocs_compilation.cpp
// This file is autogenerated. Changes will be overwritten.
#include "EWIEGA46WW/moc_fakemonitor.cpp"
#include "EWIEGA46WW/moc_hostinfo.cpp"
#include "EWIEGA46WW/moc_icecreammonitor.cpp"
```

CMake Qt Integration - contd

- Special casing regarding CMake **AUTOMOC**
  - Q_OBJECT or Q_GADGET based subclass in **header**?
    - Nothing needs to be done
    - CMake will run *moc* on the **header**
  - … inside a **source** file?
    - In that case add an *#include "<basename>.moc"* at the end of the source file
    - CMake will run *moc* on the **source** file instead
  - Also looks for other Qt macros requiring *moc*, e.g. *Q_PLUGIN_METADATA*
    - List of macros-of-interest can be extended by the user

Qt Virtual Tech Con 2020

Special Cases



Translations handling

```
cmake_minimum_required(VERSION 3.10.0)

project(translation-demo)

# Business as usual, setup CMAKE_AUTOMOC, etc...

find_package(Qt5 COMPONENTS Widgets LinguistTools
REQUIRED) ①

qt5_create_translation(QM_FILES
  ${CMAKE_SOURCE_DIR} demo_de.ts demo_fr.ts) ②

add_executable(helloworld
    ...
    main.cpp
    ${QM_FILES}
) ③

target_link_libraries(helloworld Qt5::Widgets)
```

(1) CMake functions for translation handling are inside the Qt *LinguistTools* module

(2) Call *qt5_create_translations()* on source code (.cpp and .ui files).

- Calls lupdate to generate or update *.ts files* (→ in source dir)
- Calls lrelease to generate *.qm files* (→ in build dir)

Translations handling - contd

```
# Building the project...

[2/7] Generating translations/demo_de.ts
Scanning directory '...cmake-qttranslations-
example'...
Updating 'translations/demo_de.ts'...
    Found 1 source text(s) (0 new and 1 already
existing)
[3/7] Generating translations/demo_fr.ts
…

[4/7] Generating demo_de.qm
Updating
'.../cmake-qttranslations-example/demo_de.qm'...
    Generated 0 translation(s) (0 finished and 0
unfinished)
    Ignored 1 untranslated source text(s)
[5/7] Generating demo_fr.qm
…
```

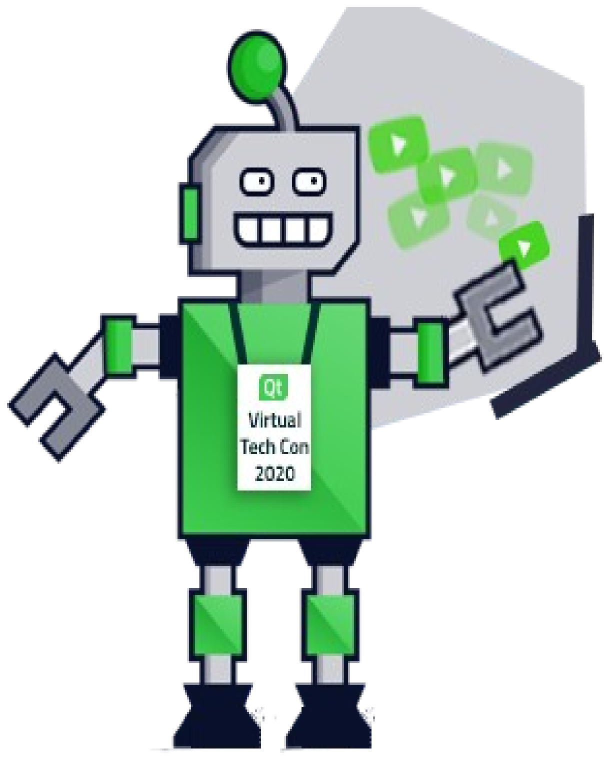
- Notice that CMake rules are driving *lupdate* & *lrelease* automatically
- Generated *.qm files* can be loaded in the application using the *QTranslator::load()* function

Big resources handling

- You might need to embed large files into resources (QRCs)
  - For example: Databases, larger image assets, …
  - But: compiler will likely fail with an “out of memory”
    - (numerous bug reports about this)
- Fix (new in Qt 5.12):
  - *qt5_add_big_resources(…)*
  - CMake counterpart to QMake’s *CONFIG+=resources_big*
  - Similar to *qt5_add_resources(…)*, but directly generates object files instead of C++ code

Qt Virtual Tech Con 2020

General Recommendations



General Recommendations

- Do not overuse global variables, “global” commands
  - Also do not overwrite vars like *CMAKE_CXX_FLAGS*, *amend them*!
  - Avoid functions like *include_directories(…), link_libraries(…)*
- Embrace using *targets* and *properties*
  - Propagate properties where needed – using *PUBLIC* keyword
    - This includes compile definitions, flags, include paths, etc.
    - Keep in mind: For a given target dependency chain A → B → C, properties set *PUBLIC*ly on target C “bubble” up to target A ⇒ Useful!
  - Avoids repetitive CMake code

General Recommendations - contd

- Do not overuse *file(GLOB ...)*
  - It would be trivial to simply add all .cpp files:

    ```
    file(GLOB SRC_FILES ${PROJECT_SOURCE_DIR}/src/*.cpp)
    add_executable(myProject ${SRC_FILES})
    # … and done!
    ```

  - But: CMake needs to be re-run in case new .cpp files are added, otherwise the build system might simply ignore them – error prone!
  - Better: List all files in the *add_executable(...)* or *add_library(...) call*

General Recommendations - contd

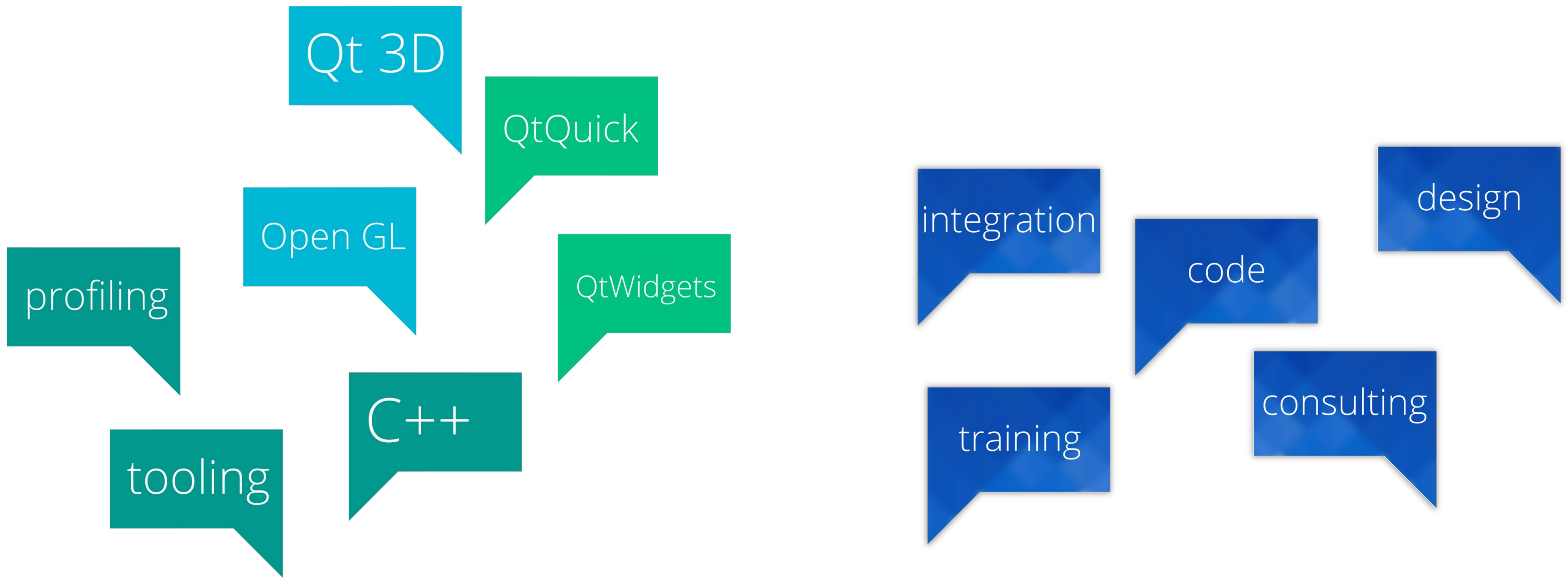
- Improving CMake runtime performance
  - Consider switching to the *Ninja* generator
  - Much simpler build system (compared to e.g. Visual Studio's *msbuild*)
    - Thus easier and quicker to generate for CMake
    - Also the build tool itself is much more compact
    - Just reads a *single* file containing build instructions
  - See also: https://blog.kitware.com/improving-cmakes-runtime-performance/

- Last but not least: *Treat CMake like production code!*
  - Keep it clean and also refactor when needed

Resources

- Well-written intro to *Modern CMake*:
  - https://cliutils.gitlab.io/modern-cmake/
- Qt and CMake Whitepaper (brand new!)
  - https://www.kdab.com/wp-content/uploads/stories/KDAB-whitepaper-CMake.pdf
- Qt5 CMake Manual
  - https://doc.qt.io/qt-5/cmake-manual.html

KDAB Services



Thank you!

- Learn more at www.resources.qt.io
- Try for Free at www.qt.io/download/

Qt World Summit 2020, Palm Springs, October 20-22
www.qtworldsummit.com

Follow us:
Website: www.kdab.com
Twitter: @KDABQt
Linkedin: /company/kdab/
Youtube: /KDAB/
My mail: kevin.funk@kdab.com