

# Utiliser CMake pour compiler des projets Qt


*Qt by Nokia*

par Johan Thelin traducteur : Guillaume Belz Qt Quarterly

Date de publication : 22/08/2010

Dernière mise à jour : 06/09/2010


Pour la gestion de la compilation de projets, Qt est fourni avec son propre utilitaire : QMake. Cependant, le développeur peut souhaiter travailler avec d'autres outils, par choix personnel ou pour répondre à certains besoins. Cet article présente en détail comment compiler des projets Qt en utilisant CMake : le processus de compilation, les paramètres de configuration, la gestion des modules Qt.

Cet article est une traduction autorisée de  **Using CMake to build Qt projects**, par Johan Thelin.

N'hésitez pas à commenter cet article !

I - L'article original.....	3
II - Introduction.....	3
III - Un exemple simple.....	3
IV - Ajouter plus de Qt.....	6
V - Les modules de Qt.....	7
VI - La valeur ajoutée et de la complexité.....	7
VII - Divers.....	7

## I - L'article original

Qt Quarterly est une revue trimestrielle électronique proposée par Nokia à destination des développeurs et utilisateurs de Qt. Vous pouvez trouver les  **versions originales**.

Nokia, Qt, Qt Quarterly et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction de l'article  **Using CMake to build Qt projects** de Johan Thelin paru dans la Qt Quarterly Issue 33.

Cet article est une traduction de l'un des tutoriels en anglais écrits par **Nokia Corporation and/or its subsidiary(-ies)**, inclus dans la documentation de Qt. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia.

## II - Introduction

Qt est fourni avec l'outil **QMake** pour la gestion de la compilation inter-plateforme. Cependant, il existe d'autres systèmes disponibles tels qu'**Autotools**, **SCons** et **CMake**. Ces outils répondent à d'autres besoins, par exemple sur les dépendances externes.

Lorsque le projet **KDE** est passé de Qt3 à Qt4, le projet est passé de l'outil de compilation **Autotools** à **CMake**. Cela a donné une position particulière à **CMake** dans le monde Qt, concernant le nombre d'utilisateurs et la prise en charge des fonctionnalités et de la qualité. Du point de vue de l'environnement de travail, Qt Creator prend en charge **CMake** depuis la version 1.1 (1.3 si vous souhaitez utiliser une chaîne de compilation Microsoft).

## III - Un exemple simple

Dans cet article, nous nous concentrerons sur **CMake** lui-même et comment l'utiliser en conjonction avec Qt. Pour faire cela, commençons par un aperçu d'un projet simple mais typique basé sur **CMake**. Comme vous pouvez le voir dans la liste ci-dessous, le projet se compose de quelques fichiers source et d'un fichier texte.

```
$ ls
CMakeLists.txt
helloworld.cpp
helloworld.h
main.cpp
```

Fondamentalement, le fichier `cmakelists.txt` remplace le fichier de projet utilisé par **QMake**. Pour compiler le projet, il faut créer un répertoire de compilation et exécuter **CMake** puis `make` dans celui-ci. La raison de la création du répertoire de compilation est que **CMake** a été conçu dès le départ en intégrant la notion de compilation hors-source. Il est possible de configurer **QMake** pour placer les fichiers intermédiaires en dehors de l'arborescence des sources, même si cela exige des étapes supplémentaires. Avec **CMake**, c'est le comportement par défaut.

```
$ mkdir build
$ cd build
$ cmake .. && make
```

```

hello-world/b: bash
File Edit View Scrollback Bookmarks Settings Help
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/g++
-- Check for working CXX compiler: /usr/bin/g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for Q_WS_X11
-- Looking for Q_WS_X11 - found
-- Looking for Q_WS_MIN
-- Looking for Q_WS_MIN - not found.
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found.
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found.
-- Found Qt-Version 4.6.2 (using /usr/bin/qmake)
-- Looking for _POSIX_TIMERS
-- Looking for _POSIX_TIMERS - found
CMake Warning (dev) in CMakeLists.txt:
  No cmake_minimum_required command is present.  A line of code such as

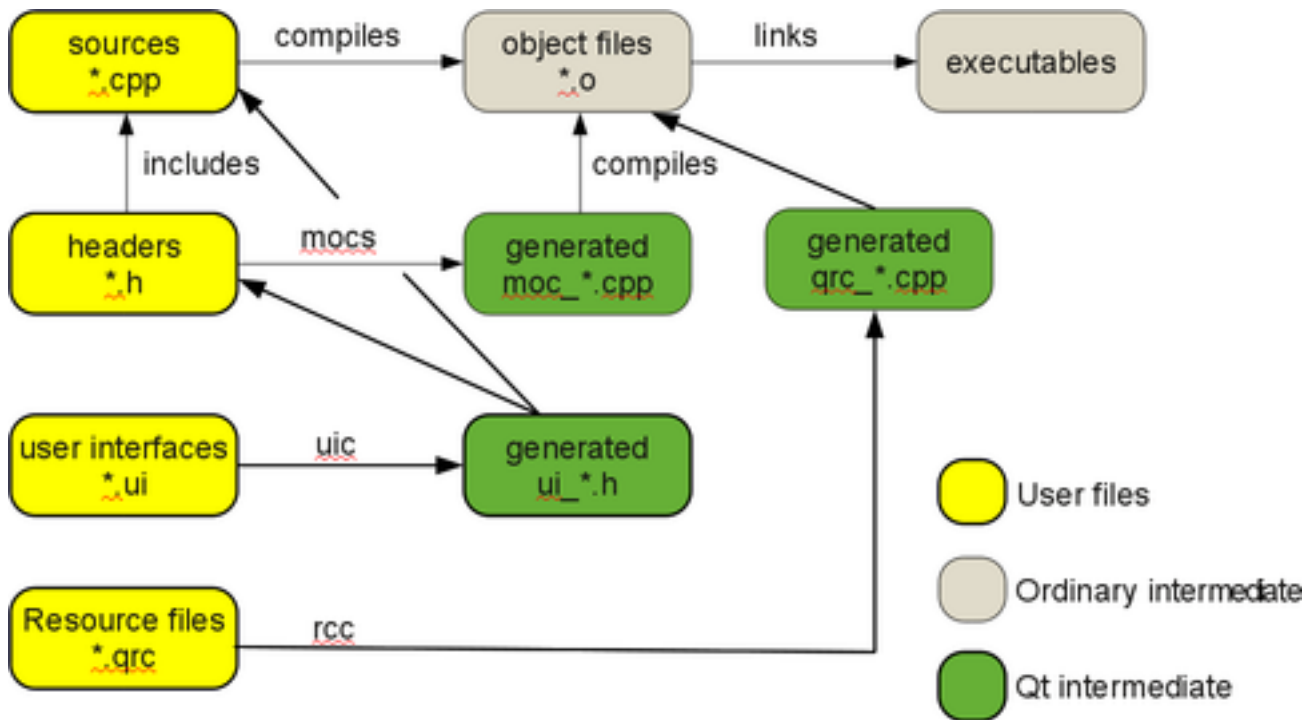
    cmake_minimum_required(VERSION 2.8)

  should be added at the top of the file.  The version specified may be lower
  if you wish to support older CMake versions for this project.  For more
  information run "cmake --help-policy CMP0000".
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /home/e8johan/work/ttk/nokia/nokia/Qt Quarterly/CMake/qc-cmake-examples/hello-world/build
[ 25%] Generating moc_helloworld.cpp
Scanning dependencies of target helloworld
[ 50%] Building CXX object CMakeFiles/helloworld.dir/main.o
[ 75%] Building CXX object CMakeFiles/helloworld.dir/helloworld.o
[100%] Building CXX object CMakeFiles/helloworld.dir/moc_helloworld.o
Linking CXX executable helloworld
[100%] Built target helloworld
e8johan@storbettan:~/work/ttk/nokia/nokia/Qt Quarterly/CMake/qc-cmake-examples/hello-world/build$
  
```

### *CMake compilant un projet simple*

L'argument donné à **CMake** se réfère au répertoire dans lequel se situe le fichier `cmakelists.txt`. Ce fichier contrôle l'ensemble du processus de construction. Pour bien le comprendre, il est important de regarder à quoi ressemble le processus de compilation. La figure ci-dessous montre comment les fichiers de l'utilisateur (sources, en-têtes, formulaires et fichiers de ressources) sont traités par les différents générateurs de code de Qt avant de rejoindre le flux de compilation C++ standard. Puisque **QMake** a été conçu pour gérer ce flux, il en cache tous les détails.



Lors de l'utilisation de **CMake**, les étapes intermédiaires doivent être manipulées de manière explicite. Cela signifie que les en-têtes avec des macros `Q_OBJECT` doivent être traités par `moc`, les formulaires d'interface utilisateur par `uic` et les fichiers de ressources par `rcc`.

L'exemple de base que nous avons commencé est un peu plus simple. Il est limité à un fichier en-tête unique qui doit être traité par `moc`. Mais, d'abord, le fichier `cmakelists.txt` définit un nom de projet et inclut le paquet Qt4 comme composante nécessaire.

```
PROJECT(helloworld)
FIND_PACKAGE(Qt4 REQUIRED)
```

Ensuite, toutes les sources impliquées dans le processus de construction sont affectées à deux variables. La commande `SET` assigne la variable indiquée en premier avec les valeurs qui suivent. Les noms `helloworld_SOURCES` et `helloworld_HEADERS` sont choisis arbitrairement. Vous pouvez les nommer comme vous le souhaitez.

```
SET(helloworld_SOURCES main.cpp helloworld.cpp)
SET(helloworld_HEADERS helloworld.h)
```

Notez que les en-têtes ne comprennent que ceux qui doivent être traités par `moc`. Tous les autres fichiers d'en-tête peuvent être omis du fichier `cmakelists.txt`. Cela implique également que, si vous ajoutez une macro `Q_OBJECT` à l'une de vos classes, vous devez veiller à ce qu'elle soit indiquée ici.

Pour appeler `moc`, la macro `QT4_WRAP_CPP` est utilisée. Elle crée une liste des noms des fichiers produits et l'ajoute dans la variable indiquée comme premier paramètre. Dans ce cas, la ligne se présente comme suit :

```
QT4_WRAP_CPP(helloworld_HEADERS_MOC ${helloworld_HEADERS})
```

Comment se déroule cette étape ? Tous les fichiers d'en-tête sont traités par `moc` et le nom des fichiers source résultant sont répertoriés dans la variable `helloworld_HEADERS_MOC`. Encore une fois, le nom de variable est choisi par convention et n'est pas imposé.

Afin de compiler une application Qt, les répertoires include de Qt doivent être ajoutés et un ensemble de paramètres doit être défini. Ceci est réalisé par les commandes INCLUDE et ADD\_DEFINITIONS.

```
INCLUDE(${QT_USE_FILE})
ADD_DEFINITIONS(${QT_DEFINITIONS})
```

Enfin, **CMake** doit connaître le nom de l'exécutable à créer et les bibliothèques qui doivent être liées. Ceci est réalisé facilement par les commandes ADD\_EXECUTABLE et TARGET\_LINK\_LIBRARIES. Maintenant **CMake** sait ce qu'il faut compiler, à partir de quoi et en passant par quelles étapes.

```
ADD_EXECUTABLE(helloworld ${helloworld_SOURCES}
    ${helloworld_HEADERS_MOC})
TARGET_LINK_LIBRARIES(helloworld ${QT_LIBRARIES})
```

Lors de l'examen de la liste ci-dessus, il s'appuie sur un certain nombre de variables commençant par QT\_. Elles sont générées par le paquet Qt4. Toutefois, en tant que développeur, vous devez explicitement y faire référence, puisque **CMake** n'est pas conçu aussi étroitement avec Qt que l'est **QMake**.

## IV - Ajouter plus de Qt

Au-delà de l'exemple initial, nous allons maintenant regarder un projet contenant des ressources et des formulaires d'interface utilisateur. L'application résultante sera très similaire à celle présentée ci-dessus, mais toute la magie de la chose est cachée sous la capot.

Le fichier CMakeLists.txt commence par nommer le projet et inclure les paquets de Qt4 - le dossier complet peut être téléchargé comme un paquet de codes sources qui accompagne cet article. Ensuite, tous les fichiers d'entrée sont énumérés et attribués à leurs variables correspondantes.

```
SET(helloworld_SOURCES main.cpp helloworld.cpp)
SET(helloworld_HEADERS helloworld.h)
SET(helloworld_FORMS helloworld.ui)
SET(helloworld_RESOURCES images.qrc)
```

Les nouveaux types de fichiers sont ensuite traités par QT4\_WRAP\_UI et QT4\_ADD\_RESOURCES. Ces macros fonctionnent de la même manière que QT4\_WRAP\_CPP et signifie que les fichiers résultants sont assignés à la variable donnée en argument le plus à gauche. Notez que les fichiers d'en-tête générés par uic sont nécessaires, puisque nous avons besoin de créer une relation de dépendance entre eux et l'exécutable final. Sinon, ils ne seront pas créés.

```
QT4_WRAP_CPP(helloworld_HEADERS_MOC ${helloworld_HEADERS})
QT4_WRAP_UI(helloworld_FORMS_HEADERS ${helloworld_FORMS})
QT4_ADD_RESOURCES(helloworld_RESOURCES_RCC ${helloworld_RESOURCES})
```

Tous les fichiers obtenus sont ensuite ajoutés comme des dépendances à la macro ADD\_EXECUTABLE. Cela inclut les fichiers d'en-tête générés par uic. Cela permet de conserver la dépendance de l'exécutable pour le fichier helloworld.ui par l'intermédiaire du fichier d'en-tête ui\_helloworld.h.

```
ADD_EXECUTABLE(helloworld ${helloworld_SOURCES}
    ${helloworld_HEADERS_MOC}
    ${helloworld_FORMS_HEADERS}
    ${helloworld_RESOURCES_RCC})
```

Avant que ce fichier puisse être utilisé pour compiler le projet, il y a une petite mise en garde à prendre en compte. Comme tous les fichiers intermédiaires sont générés en dehors de l'arborescence des sources, l'en-tête du fichier généré par uic ne sera pas trouvé par le compilateur. Pour gérer cela, le répertoire de compilation doit être ajouté à la liste des répertoires à inclure.

```
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_BINARY_DIR})
```

Avec cette ligne, tous les fichiers intermédiaires seront disponibles dans le chemin d'inclusion.

## V - Les modules de Qt

Jusqu'à présent, nous nous sommes appuyés sur les modules **QtCore** et **QtGui**. Pour être en mesure d'utiliser d'autres modules, l'environnement de **CMake** doit être réglé pour les activer. En les mettant à TRUE en utilisant la commande SET, les modules inclus peuvent être contrôlés.

Par exemple, pour activer le support OpenGL, ajoutez la ligne suivante à votre CMakeLists.txt.

```
SET(QT_USE_QTOPENGL TRUE)
```

Les modules les plus couramment utilisés sont contrôlés par les variables suivantes :

```
QT_USE_QTNETWORK
QT_USE_QTOPENGL
QT_USE_QTSQL
QT_USE_QTXML
QT_USE_QTSVG
QT_USE_QTTEST
QT_USE_QTDBUS
QT_USE_QTSCRIPT
QT_USE_QTWEBKIT
QT_USE_QTXMLPATTERNS
QT_USE_PHONON
```

De plus, la variable QT\_DONT\_USE\_QTGUI peut être utilisée pour désactiver l'utilisation de **QtGui**. Il y a une variable semblable pour désactiver **CMake**, mais c'est plus pour avoir l'ensemble des fonctionnalités disponibles que pour ajouter une valeur réellement utile.

## VI - La valeur ajoutée et de la complexité

L'utilisation de **CMake** n'est pas aussi triviale que celle de **QMake**, l'avantage est de disposer de plus de fonctionnalités. Le point le plus notable lors du passage **QMake** à **CMake** est le support de la compilation en dehors des sources. Cela peut vraiment modifier les habitudes et permet de faciliter la gestion des versions du code.

Il est également possible d'ajouter des paramètres conditionnels pour différentes plateformes et scénarios de compilation. Par exemple, utiliser des bibliothèques différentes pour différentes plateformes ou encore de modifier les paramètres du projet en fonction de la situation.

Les autres fonctionnalités puissantes sont la capacité de générer plusieurs exécutables et bibliothèques lors d'une même compilation. Ceci, en combinaison avec le module **QtTest**, peut aider à gérer la compilation des situations complexes à partir d'une configuration unique.

Le choix entre **CMake** et **QMake** est vraiment très facile. Pour les projets directement basé sur Qt, **QMake** est le choix évident. Lorsque l'accumulation des exigences franchit le seuil de complexité pour **QMake**, **CMake** peut prendre sa place. Avec le soutien de Qt Creator pour **CMake**, les mêmes outils peuvent encore être utilisés.

## VII - Divers

Au nom de toute l'équipe Qt, j'aimerais adresser le plus grand remerciement à Nokia pour nous avoir autorisés à traduire cet article !

Merci à **dourouc05**, à **eusebe19** et à **jacques\_jean** pour leur relecture et leurs conseils.