Utiliser CMake pour compiler des projets Qt

par Johan Thelin Louis du Verdier (Site Web) Qt Quarterly

Date de publication : 23/04/11

Dernière mise à jour :

Qt fournit l'outil QMake pour gérer les problèmes de génération multi-plateforme. Cependant, il existe d'autres système de compilation disponibles, comme autotools, SCons et CMake. Ces outils répondent à de diverses critères, comme les dépendances externes.

Quand le projet KDE est passé de Qt 3 à Qt 4, le projet a changé d'outil de génération, passant d'autotools à CMake. Cela a donné à CMake une position spéciale au sein du monde de Qt - à la fois du point de vue du nombre d'utilisateurs, du point de vue de la qualité et du support de diverses fonctions. Vu sous l'angle de vue d'un workflow, Qt Creator supporte CMake depuis la version 1.1 (1.3 si l'on souhaite utiliser une chaine d'outils de Microsoft).

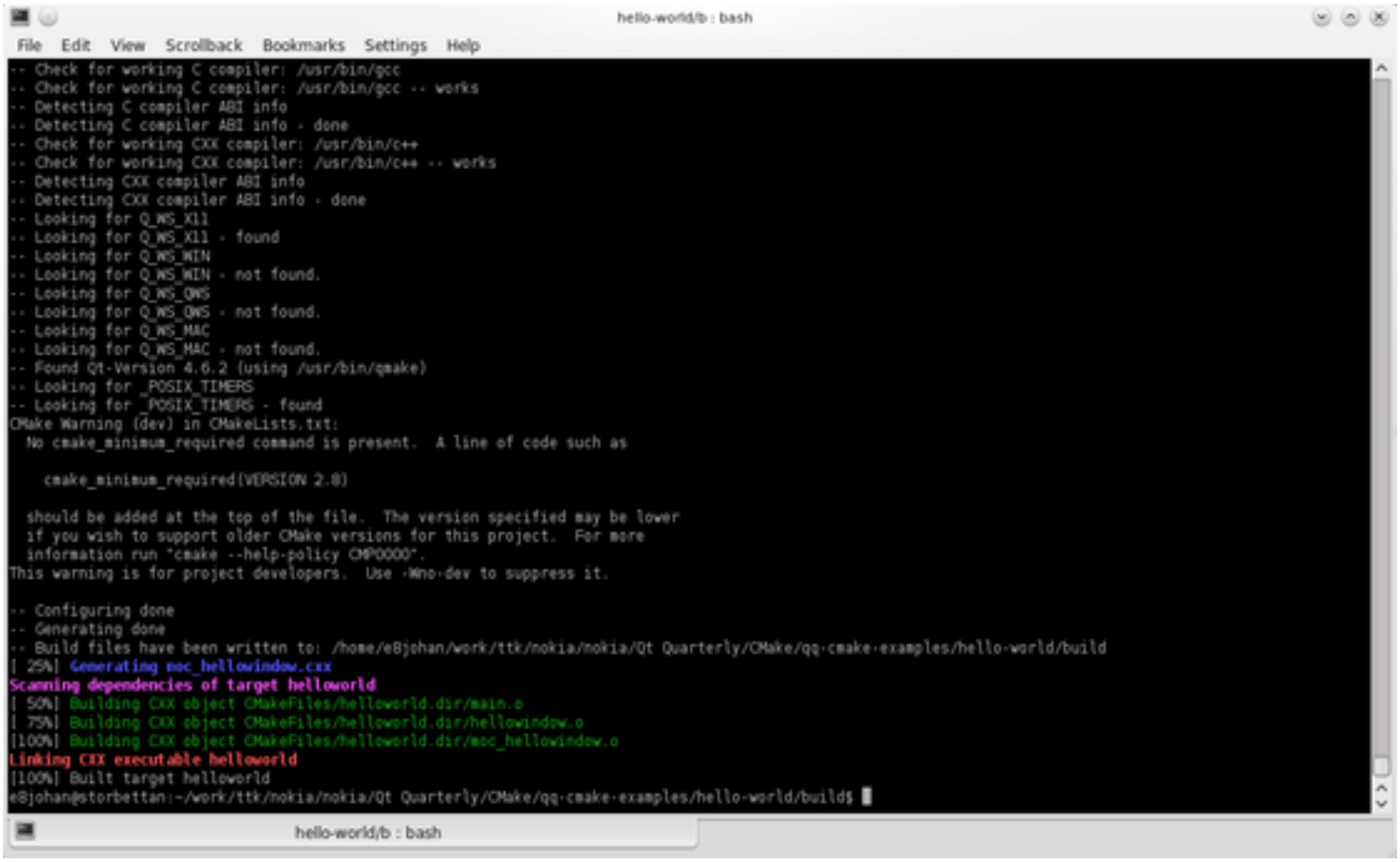
I - Un exemple simple

Dans cet article, nous allons nous focaliser sur CMake lui-même, et sur la manière de l'utiliser en conjonction avec Qt. Pour cela, commençons par une introduction d'un projet simple mais classique, basé sur CMake. Comme vous pouvez le constater avec la liste ci-dessous, le projet consiste en plusieurs fichiers sources ainsi qu'un fichier texte.

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

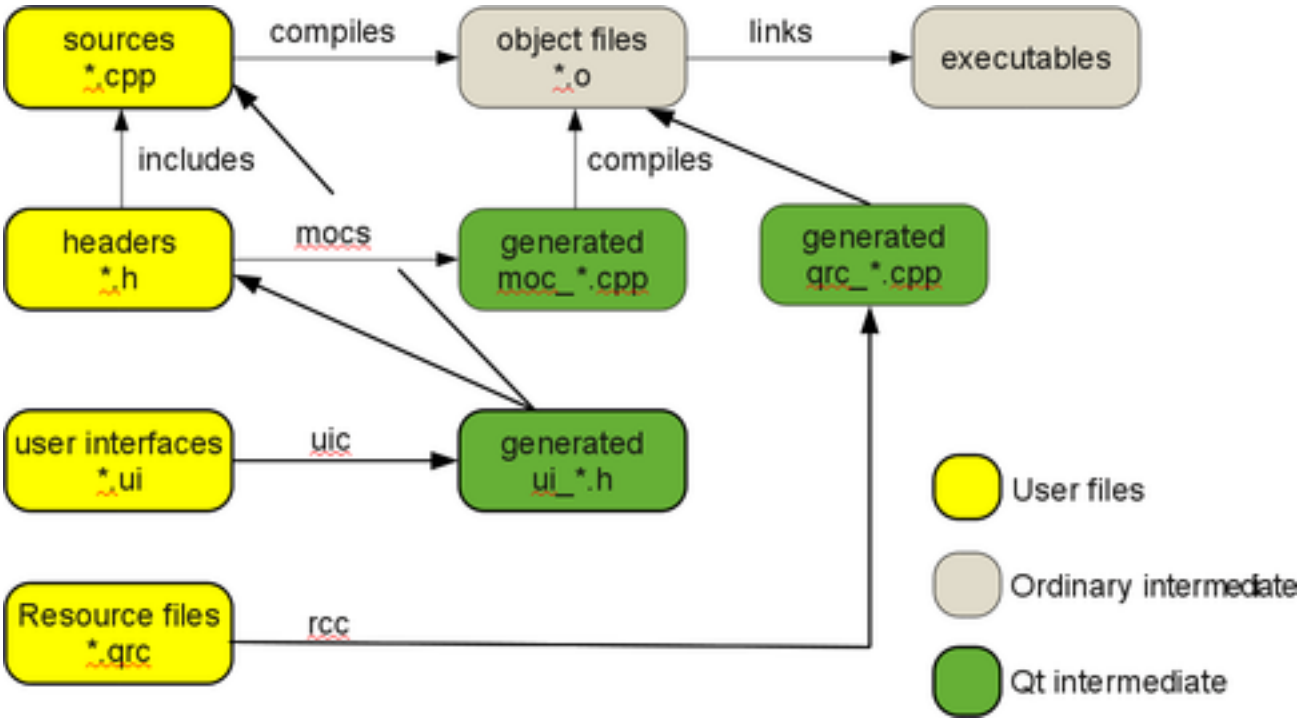
Communément, le fichier CMakeLists.txt remplace le fichier de projet utilisé par CMake. Pour générer le projet, créez un répertoire de génération et lancez CMake ; nous partirons de là. La raison de la création d'un dossier de génération est que CMake a été pensé depuis sa première ligne de code comme une générateur hors sources. Il est possible de configurer QMake pour placer des fichiers indéterminés hors des sources, mais cela requiert des étapes supplémentaires. Avec CMake, c'est le comportement par défaut.

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



CMake générant un projet basique.

L'argument passé à CMake se réfère au répertoire où le fichier CMakeLists.txt réside. Ce fichier contrôle la totalité du processus de génération. Afin de comprendre cela intégralement, il est important de savoir reconnaître l'apparence d'une procédure de génération. Le schéma situé ci-dessous montre comment les fichiers (sources, d'en-tête et ressources) sont traitées par les divers générateurs de code de Qt avant de rejoindre les étapes de compilation standard du C++. Puisque QMake a été conçu pour gérer ces étapes, il masque la totalité de leurs détails.



Le système de génération de Qt.

Lorsqu'on utilise CMake, les étapes intermédiaires doivent être gérées explicitement. Cela signifie qu'un fichier d'en-tête avec la macro Q_OBJECT doit être traité par moc, que les formes d'interface utilisateur doivent, par uic et les fichiers de ressources par rcc.

Dans l'exemple que nous avons commencé par un Hello world, c'est cependant un peu plus simple. Il se limite à un unique fichier d'en-tête qui nécessite d'être passé par le moc. Le CMakeLists.txt définit tout d'abord un nom de projet puis inclut le pack de Qt 4 en tant qu'élément nécessaire.

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

Alors, toutes les sources impliquées dans le processus de génération sont assignées dans deux variables. La commande SET assigne la variable listée dans un premier temps avec la valeur qui suit. Les noms, helloworld_SOURCES et helloworld_HEADERS, sont des conventions. Vous pouvez les nommer comme bon vous semble.

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

Notez que les fichiers d'en-être incluent uniquement les en-têtes qui ont besoin d'être traitées par moc. Toutes les autres en-têtes peuvent être laissées hors du fichier CMakeLists.txt. Cela implique que, si vous ajoutez une macro Q_OBJECT dans n'importe laquelle de vos classes, vous devrez assurer que son en-tête soit listée.

Pour appeler le moc, la macro QT4_WRAP_CPP est utilisée. Elle assigne le nom des fichiers résultants à la variable listée dans un premier temps. Dans ce cas, la ligne ressemble à ceci :

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

Ce qui se produit est que tous les fichiers d'en-tête sont traités par moc et que les noms des fichiers résultants sont listés dans la variable helloworld_HEADERS_MOC. À nouveau, la variable est nommée conventionnellement, et non par obligation.

Afin de générer une application Qt, les répertoires d'inclusion de Qt ont également besoin d'être ajoutés, tout autant qu'une gamme de defines doit être instaurée. Ces éléments sont gérés par le biais des commandes INCLUDE et ADD_DEFINITIONS.

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

Enfin, CMake a besoin de connaître le nom de l'exécutable résultant et de savoir ce qu'il doit y lier. C'est conventionnellement géré par les commandes ADD_EXECUTABLE et TARGET_LINK_LIBRARIES. Dès lors, CMake sait quoi générer, depuis quelles étapes et par quelles procédures.

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

En passant en revue la liste ci-dessus, il relie un nombre donné de variables commençant par QT_. Elles sont générées avec le pack de Qt 4. Cependant, en tant que développeur, vous devez vous référer à elles, dans le sens où CMake n'a pas été fait pour suivre Qt aussi étroitement que le fait QMake.

II - Mettre plus de Qt

En allant au-delà de l'exemple initial, nous allons maintenant observer un projet contenant à la fois des ressources et des formes d'interface utilisateur. L'application résultante sera plus ou moins similaire à sa prédicatrice, mais toute la magie prend place derrière ces apparences.

Le fichier CMakeLists.txt débute par le nommage du projet et par l'inclusion du pack de Qt 4 - le fichier complet peut être téléchargé dans l'archive accompagnant cet article (*NDT : aucune archive n'accompagne l'article original*). Alors, tous les fichiers d'entrée sont listés et assignés à leur variable correspondante.

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

Les nouveaux types de fichiers sont gérés par QT4_WRAP_UI et QT4_ADD_RESOURCES. Ces macros opèrent de la même façon que QT4_WRAP_CPP. Cela signifie que les fichiers résultants sont assignés à des variables données en tant qu'arguments dits le plus à gauche. Notez que les fichiers d'en-tête générés par uic sont nécessaires, tout autant que nous avons besoin de générer un arbre de dépendances entre eux et l'exécutable final. Dans le cas contraire, ce dernier ne sera pas créé.

```
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 résultants sont alors ajoutés en tant que dépendances à la macro ADD_EXECUTABLE. Cela inclut les fichiers d'en-tête générés par uic. Cela établit la dépendance depuis l'exécutable jusqu'au fichier hellowindow.ui, par l'intermédiaire de l'en-tête ui_hellowindow.h.

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

Avant que ce fichier puisse être utilisé pour générer le projet, il reste une petite chose à gérer. Comme tous les fichiers intermédiaires sont générés à l'extérieur de l'arbre des sources, le fichier d'en-tête généré par uic ne sera pas localisé par le compilateur. Dans le but de gérer cela, le répertoire de génération a besoin d'être ajouté à la liste des répertoires d'inclusion.

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

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

III - Les modules de Qt

Jusqu'à maintenant, nous avons traité de modules de QtCore et de QtGui. Pour pouvoir utiliser d'autres modules, l'environnement de CMake doit être configuré pour les activer. En les définissant à TRUE via 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 en utilisant 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

En plus d'elles, la variable QT_DONT_USE_QTGUI est disponible pour désactiver l'utilisation de QtGui. Il y a une variable similaire pour désactiver QtCore, mais c'est plus pour faire office de fonction complète que pour mettre à disposition une valeur utile.

IV - Ajout de valeur et de complexité

Il n'est pas trivial d'utiliser CMake comme QMake, mais les apports correspondent à avoir plus de propriétés. Le point le plus notable lors d'un mouvement de QMake à CMake est le support intégré de CMake pour les générations hors sources. Cela peut réellement changer les habitudes, et ainsi rendre le code « versionniste« plus simple.

Il est également possible d'ajouter des conditions pour de diverses plateformes et scénarios, par exemple pour utiliser de différentes bibliothèques pour de différentes plateformes, tout aussi bien que régler le même projet différemment selon les situations.

Une autre fonctionnalité puissante est la capacité à générer de multiples exécutables et bibliothèques dans une même génération, aussi bien qu'il est possible d'utiliser des exécutables et bibliothèques donnés dans la même génération. Cela, en combinaison avec le module QtTest, peut permettre de gérer des situations de génération complexes avec pour base une unique configuration.

Le choix entre CMake et QMake est plutôt simple. Pour les projets Qt normaux, QMake est un choix judicieux. Lorsque les nécessités de génération dépassent le seuil de complexité de QMake, CMake peut prendre sa place. Avec le support de Qt Creator de CMake, les mêmes outils peuvent être utilisés.