Qmake, le système de build multi-plate-forme qu'il vous faut.

Qmake est l'outil de build fourni avec le framework Qt. La puissance de ce dernier repose, entre autres, sur la grande portabilité du code Qt. Cette portabilité nécessite absolument un système de construction totalement multi-plate-forme, c'est le rôle de Qmake. Voyons ce qui en fait un outil attrayant.

Qmake n'est pas, à proprement parler un système de build. C'est un outil qui permet de simplifier considérablement le processus de création des fichiers de build. Typiquement, Qmake va être utilisé pour générer les fichiers Makefile nécessaires à la construction d'un projet. Actuellement, Qmake permet de générer des Makefiles pour toutes les plates-formes supportées par Qt. Il est aussi capable de générer les fichiers projet pour Microsoft Visual studio ou Xcode permettant ainsi d'utiliser les outils propres à certaines plates-formes. L'intérêt de cet outil réside dans son utilisation pour des projets écrits ou non en Qt. Pour les projets Qt, Qmake génère automatiquement les cibles pour Moc (Meta Object Compiler) et Uic (User Interface Compiler).

Les bases...

Commençons par les bases, Qmake se repose sur un fichier texte (UTF8 pour bien faire) dit "fichier projet". Ce fichier est une suite de variables ou de directives qui seront utilisées pour générer les instructions de compilation. La façon la plus simple d'obtenir un squelette de fichier projet est de faire appel à Qmake dans le répertoire racine de votre projet :

```
qmake -project
```

Prenons un premier exemple simpliste. Créez un répertoire exemple1. Dans celui-ci, créez un répertoire src.

```
mkdir -p exemple1/src
```

Créez dans le répertoire src, un fichier main.cpp contenant le " programme " suivant :

```
#include <iostream>

int main(){
        std::cout << "HelloWorld\n" ;
        return 0;
}
```

Maintenant, dans le répertoire exemple1, lancez " qmake -project ". Celui-ci va créer un fichier appelé exemple1.pro contenant ceci :

```
TEMPLATE = app
TARGET =
DEPENDPATH += . src
INCLUDEPATH += .
# Input
SOURCES += src/main.cpp
```

Nous reviendrons sur ce fichier, pour le moment transformez-le en fichier de build en tapant " qmake exemple1.pro ". Cela aura pour effet de générer le Makefile, lancez-le en tapant " make ", vous verrez alors votre programme se construire.

Vous constaterez aussi que votre programme se retrouve lié à Qt (et à toutes ses dépendances)... cela fait beaucoup pour notre " Helloworld "... D'autant plus que nous n'utilisons pas Qt ! C'est donc le moment d'expliquer à Qmake que nous ne voulons pas qu'il lie notre application à Qt. Revenons donc sur le fichier projet. Nous constatons d'abord qu'il ressemble énormément à un fichier de configuration et que chaque ligne présente un schéma identique :

```
VARIABLE = VALEUR
```

D'autres types d'instructions existent mais la majeure partie d'un fichier projet est constituée de déclarations de variables. La première disponible est la variable TEMPLATE, celle-ci contient le template qui sera utilisé pour générer le projet et peut prendre les valeurs suivantes :

- app : crée un Makefile pour générer une application ;
- lib : crée un Makefile pour générer une librairie ;
- subdirs : crée un Makefile par répertoire déclaré dans la variable SUBDIRS ;
- vcapp : spécifique à Windows, crée un fichier de projet Visual studio pour une application ;
- vclib : spécifique à Windows, crée un fichier de projet Visual studio pour une librairie.

Dans notre cas nous utilisons le template app. La variable suivante est TARGET, qui est, par défaut, vide (la cible générée prend alors le nom du répertoire). Viennent ensuite :

- DEPENDPATH : la liste des répertoires qui vont être scrutés pour résoudre les dépendances ;
- INCLUDEPATH : la liste des répertoires contenant les fichiers d'en-têtes ;
- SOURCES : la liste des fichiers sources du projet ;
- si nous avions des fichiers d'en-têtes, ils seraient listés dans la variable HEADERS.

Lors de la construction précédente du binaire vous avez pu constater que celui-ci était compilé et lié contre Qt alors que nous ne l'utilisons pas. Ceci n'est pas anodin et un " ldd exemple1 " vous fera constater l'étendue des dégâts ! Pour remédier à ce problème rien de plus simple : il suffit d'ajouter la définition suivante à notre fichier projet :

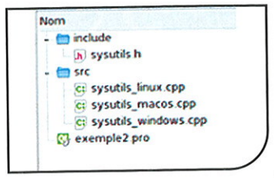
```
CONFIG -= qt
```

Relancez la compilation. Le Makefile est automatiquement regénéré (Qmake prend soin de mettre en place la règle adéquate). Constatez aussi que, cette fois, aucune librairie superflue n'est incluse, et ldd vous dit merci ! Qu'avons-nous fait avec la variable CONFIG ? Nous avons simplement demandé à Qmake de ne pas générer les règles qui lient Qt à notre application.

...pour s'amuser !

Attelons-nous maintenant à construire un projet multi-plate-forme avec du code dit " plateform specific ". Nous allons, pour cela, construire une librairie partagée d'utilitaires systèmes. Cette librairie contient un ensemble de fonctions telles que forceRemoveDirectory(const QString &), par exemple, qui font appel aux utilitaires propres à chaque système (del ou rm dans le cas présent).

Notre librairie va compiler contre QtCore car nous souhaitons bénéficier des types de Qt (entre autres), mais nous ne souhaitons pas compiler contre la partie GUI, nous verrons comment faire. Les librairies partagées sont évidemment spécifiques à une plate-forme : share object (.so) sous GNU/Linux, DLLs sous Windows et dynlib sous Mac OS. Il va falloir régler ce problème qui pourrait être épineux si nous ne disposions pas de Qmake.



Dans un premier temps voici l'arborescence de notre projet figure ci-contre. Tout d'abord, générez le squelette de base du fichier projet en tapant :

```
qmake -project -t lib
```

Dans un premier temps, voyons comment gérer la compilation des sources spécifiques à chaque OS. Qmake met à notre disposition un système de " scopes " très pratique. Les scopes sont des conditions qui permettent de prendre en compte les spécificités de chaque situation. Une des utilisations vraiment intéressante des scopes est la configuration du build en fonction de l'OS. Ainsi il suffit, pour ne compiler que le fichier source relatif à un OS, d'ajouter les définitions suivantes au fichier projet :

```
win32 {
  SOURCES += src/sysutils_windows.cpp
}
macx {
  SOURCES += src/sysutils_macos.cpp
}
unix{
  SOURCES += src/sysutils_linux.cpp
}
```

Ce squelette nous permet aussi d'ajouter, pour chaque système d'exploitation, l'option adaptée aux librairies partagées. Il suffira de rajouter : CONFIG += shared pour les Linux/Unices, CONFIG += dll pour Windows et CONFIG += dylib lib_bundle pour Mac OS X. Dans le dernier cas, l'option supplémentaire permet de créer le bundle particulier aux librairies de Mac OS X.

Pour utiliser les types de Qt, nous devons utiliser QtCore. En revanche, comme nous désirons nous passer de QtGui, il faut le préciser à Qmake (ces 2 modules étant inclus par défaut) :

```
QT -= gui
```

La variable QT permet de configurer les modules de Qt utilisés par notre projet. La déclaration par défaut (lorsque rien n'est spécifié) est :

```
QT = core gui
```

Donc quand nous écrivons :

```
QT -= gui
```

C'est équivalent à :

```
QT = core
```

Les opérateurs += et -= se comportant comme nous l'attendons en concaténant et en supprimant une valeur de la liste. Le signe = représentant, bien sûr, l'affectation.

Les scopes de Qmake deviennent indispensables une fois couplés aux possibilités des objets qui peuvent être déclarés dans le fichier projet. Un objet est une variable qui contient plusieurs membres. Certaines variables prédéfinies du fichier projet font une bonne utilisation de ces objets si les membres requis sont remplis. Prenons l'exemple de la variable prédéfinie INSTALLS. Celle-ci prend en argument une liste d'objets contenant au minimum le chemin d'installation. La forme la plus simple est donc la suivante :

```
target.path = /usr/lib
INSTALLS = target
```

Ici, nous spécifions que les cibles produites lors de la construction doivent être installées dans le répertoire /usr/lib. Il peut arriver que nous ayons plus que les cibles de construction à installer. Dans le cas présent nous voulons installer aussi le fichier d'en-tête. Pour cela, un nouvel objet est nécessaire :

```
# Les fichiers concernés par cet objet sont les .h du répertoire
 include/
includes.files = include/*.h
# ceux-ci seront installés dans le répertoire /usr/include
includes.path = /usr/include
Une fois ceci fait, il est nécessaire d'ajouter l'objet includes
 à la variable INSTALLS :
INSTALLS += includes
```

Les objets de la variable INSTALLS peuvent posséder un autre membre : extra. Celui-ci permet d'exécuter un certains nombre de commandes lors du build. Par exemple, si nous désirons installer les sources compressées de notre librairie :

```
sources.files = src/sysutils_linux.cpp.gz
sources.path = /usr/src/example2
sources.extra = gzip -c src/sysutils_linux.cpp > src/sysutils
_linux.cpp.gz
INSTALLS += sources
```

Ces possibilités sont, bien entendu, à mettre en relation avec les scopes conditionnels concernant les systèmes d'exploitation. Pour terminer ce rapide tour d'horizon des possibilités de Qmake, ajoutons qu'il est possible de déclarer des liaisons avec d'autres librairies si celles-ci supportent pkgconfig, en utilisant la syntaxe suivante :

```
CONFIG += link_pkgconfig
PKGCONFIG += ogg dbus-1
```

Nous n'utilisons pas cette possibilité dans notre exemple.

J'espère que cet article d'introduction vous aura convaincu que la construction multi-plate-forme n'est pas l'enfer fréquemment décrit, du moment que l'on choisit les bons outils...

■ Arnaud **Dupuis**
Uperto/Devoteam