

Compilation des projets Qt4

par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 26/09/2007

Dernière mise à jour :

La compilation des projets avec Qt peut être complexe. Il faut gérer les fichiers à l'aide de moc, uic et autres. Il est plus simple de les gérer automatiquement avec l'outil standard proposé par Qt ou à l'aide d'autres outils.

I - Utilisation de QMake

- I-A - Création d'un projet et modification

- I-B - Utilisation avancée

 - I-B-1 - Les options globales

 - I-B-2 - Les autres options utiles

 - I-B-3 - Compilation conditionnelle

 - I-B-4 - QMake comme langage de script

II - Utilisation d'autres outils

- II-A - CMake


- II-B - SCons

I - Utilisation de QMake

QMake est l'outil fourni avec Qt. Il est capable de gérer les fichiers supplémentaires générés par Qt et de détecter quand ils sont nécessaires.


I-A - Création d'un projet et modification

Si vous avez un certain nombre de fichiers existants, il suffit d'exécuter `qmake -project` pour créer un fichier `.pro` ou pour le mettre à jour. Ce fichier contient les paramètres du projet.

 *Si le fichier est modifié et si `qmake -project` est réexécuté, les modifications sont perdues.*

Voici les différents paramètres indispensables :

- HEADERS est la liste des en-têtes à prendre en compte. Ces en-têtes sont analysés pour trouver les fichiers possédant la macro `Q_OBJECT`.
- SOURCES est la liste des fichiers sources à compiler, à laquelle s'ajoute la liste des sources générés automatiquement par Qt.
- TEMPLATE indique le type de projet, `app` indique un exécutable, `lib` est une bibliothèque, `vcapp` et `vclib` génèrent un fichier solution et `subdirs` indique que les sous-dossiers contiennent aussi des projets à compiler.
- TARGET est le nom de l'application, par défaut il s'agit du dossier parent
- DEPENDPATH contient la liste des chemins de dépendances des bibliothèques nécessaires à l'édition des liens des projets
- INCLUDEPATH contient la liste des chemins où trouver les en-têtes nécessaires à la compilation des projets

 *HEADERS = {une liste de fichiers} écrase l'ancienne valeur de HEADERS, HEADERS += {une liste} ajoute à HEADERS une liste de fichiers et HEADERS -= {une liste} supprime une série de fichiers.*

La création des fichiers de projet `make` ou `nmake` (par exemple) est effectuée par l'instruction `qmake`. Il suffit d'exécuter le `Makefile` ou de construire le projet dans le cas de la création d'un fichier `.sln` avec Visual Studio.

Voici un petit exemple tiré de l'article sur les signaux et slots :

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += exemple.h
SOURCES += exemple.cpp
```

I-B - Utilisation avancée

L'utilisation avancée consiste à modifier ou ajouter d'autres objets dans le fichier `.pro`.

I-B-1 - Les options globales

Ce que j'appelle les options globales sont les suivantes :

- CONFIG indique les options globales du projet
- QT indique quelles sont les bibliothèques Qt à utiliser. Par défaut core et gui sont définis dans cette liste, il est possible d'ajouter network, opengl, sql, svg, xml ou qt3support pour ajouter l'une de ces sous-bibliothèques

Les options de compilation pour CONFIG sont les suivantes :

- release indique que le projet sera compilé en mode release
- debug indique que le projet sera compilé en mode debug
- debug_and_release indique que les modes debug et release seront utilisés
- build_all construit les deux modes en même temps
- ordered indique que les sous-dossiers doivent être construits dans l'ordre indiqué par SUBDIRS
- warn_on indique au compilateur de maximiser les warnings
- warn_off éteint les avertissements du compilateur

CONFIG s'occupe aussi du type de projet de manière fine :

- qt indique que le projet nécessite les bibliothèques Qt (par défaut)
- opengl indique qu'OpenGL est nécessaire
- thread indique que le projet est multi-thread (potentiellement)
- x11 crée une application X11
- windows entraîne la création d'une application graphique Win32
- console entraîne la création d'une application en console Win32
- dll génère une bibliothèque dynamique
- staticlib génère une bibliothèque statique
- plugin indique que la bibliothèque est un plugin Qt
- designer indique que le plugin est un plugin pour Qt Designer
- uic3 indique que uic3 doit travailler sur les données de la variable FORMS3 si elle existe, sur FORMS dans le cas contraire
- no_iflags_merge empêche que la liste des bibliothèques desquelles le projet dépend soit analysée et les doublons supprimés
- resources indique que rcc doit travailler sur les données de la variable RESOURCES si elle existe


Des points spécifiques au C++ peuvent être utilisés :

- exceptions entraîne l'utilisation des exceptons (Qt ne nécessitant pas les exceptions, cette option n'est pas indispensable)
- rtti indique d'activer le support RTTI (dynamic_cast<> et autres)
- stl entraîne le support de la STL

Enfin, une série d'options est dépendante de l'OS utilisé :

- flat est utilisé pour les projets Visual Studio, les en-têtes et les fichiers sources n'étant alors pas hiérarchisés.
- embed_manifest_dll ajoute le manifeste dans l'application (Visual Studio 2005)

- ppc crée un binaire PPC (Mac)
- x86 crée un binaire pour MacIntel
- app_bundle crée un bundle pour l'exécutable
- lib_bundle crée un bundle pour la bibliothèque

 Sous Linux, l'option `PKGCONFIG` permet d'utiliser `pkg-config` pour connaître la configuration nécessaire pour utiliser certains paquets, comme `dbus-1`.

I-B-2 - Les autres options utiles

`DEFINES` ajoute une liste de macros à définir. Par exemple `DEFINES+=QQCH` entraîne l'ajout de `-DQQCH` à la ligne de commande.

`DESTDIR` modifie le dossier de destination du projet.

`DISTFILES` contient une liste de fichiers à ajouter à la cible `dist` (qui crée une archive du projet).

`FORMS` est la liste des fichiers `.ui`

`INSTALLS` est la liste des objets à installer. Le dossier d'installation est donné par l'attribut `path` des objets (`objet.path = mon_dossier`).

`LIBS` est la liste des bibliothèques à ajouter. Il est possible d'utiliser `-l` et `-L` pour spécifier bibliothèque et dossier sous Linux.

`SUBDIRS` est la liste des sous-dossiers à explorer pour un sous-projet.


`TRANSLATIONS` est la liste des fichiers de traduction de l'application.

I-B-3 - Compilation conditionnelle

Il existe des incompatibilités entre Windows et Linux, nécessitant des macros différentes, des fonctions différentes, ... Pour cela, `qmake` a une solution. Il suffit d'utiliser des blocs avec accolades :

```
win32{
  SOURCES += win32.cpp
}
```

Avec `!win32` comme tête de bloc (`!` étant l'opérateur **not**), toutes les configurations différentes de `win32` utiliseront ce qui est indiqué dans le bloc. A la place de `win32`, on peut mettre `unix` ou `macx`.

 Sans aller jusqu'à créer un bloc, il est possible d'écrire `win32:debug:SOURCES+=win32_debug.cpp` pour un fichier spécifique à `win32` en mode `debug`.

Ce type de bloc est semblable à un test `if`, et donc il est possible d'ajouter un bloc `else` (tiré de la documentation de `qmake`):

```
win32:xml {
    message(Building for Windows)
    SOURCES += xmlhandler_win.cpp
} else:xml {
    SOURCES += xmlhandler.cpp
} else {
    message("Unknown configuration")
}
```

I-B-4 - QMake comme langage de script

Comme tout langage de script, il est possible de créer d'autres variables. Par exemple :

```
MY_DEFINES = $$DEFINES
```

créer une variable MY_DEFINE dont le contenu est celui de la variable DEFINES.

De même, des fonctions peuvent être créées et utilisées pour les blocs :

```
defineTest(allFiles){
    files = $$ARGS

    for(file, files) {
        !exists($$file) {
            return(false)
        }
    }
    return(true)
}
```

On voit ici qu'il existe des fonctions prédéfinies dans qmake :

- `basename(variablename)` retourne le nom d'un fichier à partir de son nom complet
- `CONFIG(config)` teste si une certaine configuration est active
- `contains(variablename, value)` retourne vrai si la variable contient la valeur value
- `count(variablename, number)` retourne vrai si la variable contient au moins number éléments
- `dirname(file)` retourne le nom du dossier à partir du nom d'un fichier
- `error(string)` affiche une erreur
- `eval(string)` évalue la chaîne de caractères
- `exists(filename)` retourne vraie si le fichier existe
- `find(variablename, substr)` retourne vrai si la sous-chaîne existe dans la variable
- `for(iterate, list)` itère sur une liste
- `include(filename)` inclut le contenu d'un fichier dans le projet et retourne vrai dans ce cas
- `infile(filename, var, val)` retourne vrai si le fichier contient une variable var contenant la valeur val
- `isEmpty(variablename)` retourne vrai si la variable est vide
- `join(variablename, glue, before, after)` concatène les éléments de la variable avec la glue
- `member(variablename, position)` retourne l'élément à la position spécifiée
- `message(string)` affiche un message
- `prompt(question)` pose une question

- `quote(string)` convertir la chaîne en une entité et retourne le résultat
- `replace(string, old_string, new_string)` remplace une sous-chaîne par une autre
- `sprintf(string, arguments...)` affiche un message
- `system(command)` exécute une commande système
- `unique(variablename)` retourne une liste de contenant aucun doublons
- `warning(string)` affiche un avertissement

II - Utilisation d'autres outils

Je ne rentrerai pas dans les détails de l'utilisation de CMake et de SCons car cela nécessiterait un autre tutoriel pour chacun d'eux. En revanche, j'expose ici les outils pour automatiser en partie la compilation de projets utilisant Qt.

II-A - CMake

Qt est supporté par CMake grâce à l'instruction `FIND_PACKAGE(Qt4)`. A ce moment, une série de variables sont placées et de fonctions disponibles.

Les premières variables intéressantes sont `QT_XXX_LIBRARY` où `XXX` est une bibliothèque de Qt, par exemple `QtCore` ou `QtGui`. Il s'agit du nom de ces bibliothèques sur le système. Elles doivent être expressément indiquées pour être liées à des projets construits par CMake.

Pour activer l'utilisation de Qt dans un sous-projet géré par CMake, il faut exécuter `INCLUDE(${QT_USE_FILE})`.

Enfin, pour utiliser `moc`, `uic` et `rcc`, il faut utiliser les fonctions :

- `QT4_WRAP_CPP()` qui crée une liste de fichiers `.cpp` cibles à partir des fichiers d'entête donnés en paramètres (ces fichiers doivent ensuite être donnés à `ADD_EXECUTABLE` ou `ADD_LIBRARY`)
- `QT4_WRAP_UI()` qui crée une liste de fichiers `.h` à partir des fichiers `.ui`
- `QT4_ADD_RESOURCE()` qui génère des fichiers `.cpp` à partir des ressources

II-B - SCons

SCons est un outil relativement jeune, mais il permet de gérer correctement les fichiers générés par `moc` et par `uic`, et ceci de manière automatique. Pour cela, il faut indiquer lors de la création de l'environnement qu'il faut utiliser l'outil Qt, par exemple de cette manière : `Environment(tools=['default','qt'])`.

Ainsi, les fichiers d'entête seront analysés pour savoir s'ils doivent être "moccés" et les fichiers `.ui` seront automatiquement transformés en fichiers d'en-tête.



Il est possible de créer un objet `Moc()` ou un objet `Uic()` à la main, comme c'est le cas pour les autres objets avec SCons.

