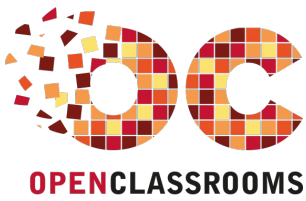
Débugger avec Qt

Par Julien Rosset (Darkelfe)



www.openclassrooms.com

Dernière mise à jour le 24/08/2011

Sommaire

Débugger avec Qt

Par  Julien Rosset (Darkelfe)

Mise à jour : 24/08/2011
Difficulté : Intermédiaire

Lorsque l'on programme, il y a une constante qui revient à chaque fois : le débuggage.

Petit rappel pour ce qui ne savent pas ou plus : le débuggage est « l'art » de suivre toutes les opérations d'un programme afin de déterminer l'origine d'une ou plusieurs erreurs.

En effet, lorsque l'on réalise un programme ou une librairie (quel que soit le langage) il y a toujours un moment où on a besoin de débugger. Il s'agit en général d'un travail long et souvent pénible car les erreurs n'apparaissent pas toujours lors de la compilation. Il faut donc faire en sorte de les traquer une à une.

Ce tutoriel vous propose donc d'apprendre une des nombreuses manières de débugger vos programmes ou vos librairies, et ce grâce à Qt.

Pour comprendre ce tutoriel il faut connaitre un minimum le C++ (notamment la POO et la notion d'objet). Et comme les techniques de débuggage portent sur Qt, il est mieux de savoir de quoi on parle.

Sommaire du tutoriel :

Comment débugger ?

Il existe de nombreuses manières de débugger, certaines fonctionnant mieux que d'autres. Mais, en général, toutes fonctionnent relativement bien et ont chacune leurs avantages / désavantages.
Le choix d'une technique de débuggage ressemble au choix d'un OS (**Operating System** : système d'exploitation) : chacun dit que le sien est meilleur que les autres et tente d'en convaincre tout le monde (j'exagère un peu, je sais). Personnellement je pense que la meilleure technique de débuggage est celle avec laquelle vous vous sentez le plus à l'aise.

Parmi les nombreuses manières de débugger un programme, voici les deux principales :

- Utiliser un *debugger* : cette technique consiste à utiliser un programme qui intègre des fonctions de débuggage comme les « breakpoints » ou la visualisation des variables en temps réel ;
  - **Avantage** : facilité de mise en œuvre, utilisation simple et rapide ;
  - **Désavantage** : — (ce n'est pas vrai, mais voir ci-dessous) ;
- Utiliser la console : ici, on modifie le code pour ajouter des lignes qui donnent des informations sur le déroulement du programme ;
  - **Avantage** : — (idem, voir plus bas) ;
  - **Désavantage** : obligation de re-compiler à chaque modification du débuggage.

Heu… Je ne vois pas trop où est la difficulté de choisir telle ou telle technique. Tu es sûr de ce que tu dis ?

Absolument certain, mais le débuggage ne s'utilise pas que pour débugger des programmes. On peut aussi débugger des librairies/plugins et autres objets non exécutables. Dans ces cas-là, il est impossible d'utiliser la visualisation des variables et les "breakpoints" directement sur les fichiers de sortie. En effet, ceux-ci ne disposent pas de fonction *main*, donc le débugger ne sait pas par où il doit commencer. Donc les *debuggers* perdent tout leur intérêt dans ce genre de cas. La seule manière d'utiliser un *debugger* avec une librairie/plugin, c'est de créer un code exemple puis de lancer le *debugger* avec l'exemple.
On peut donc deviner qu'il devient plus facile d'utiliser la deuxième méthode.

Comme je l'ai dit plus haut, ce tutoriel est là pour vous apprendre à vous servir des outils mis en place par Qt pour le débuggage. Comme Qt est une librairie, elle ne propose pas de *debugger* en tant que programme, mais elle offre de nombreux outils pour afficher des « choses » dans la console. C'est donc sur cet aspect qu'on va se pencher.

Ne perdez pas de vue que c'est aussi utilisable pour des programmes, notamment sur les systèmes embarqués (quoique la plupart d'entre eux ont des *debuggers* spécifiques fournis). Donc n'hésitez pas à lire la suite.

De plus, les fonctions de débuggage proposées par Qt peuvent transmettre des informations au *debugger* (principalement sous Windows). Donc la suite n'est pas forcément à négliger.

Mode release/debug

La première chose à savoir lorsqu'on souhaite faire du débuggage avec Qt, c'est qu'il existe, en gros, deux modes de compilation :

- Le mode **debug** : lors de l'exécution et notamment lors d'une erreur, les fonctions renvoient plus d'informations concernant ce qui c'est produit. En contrepartie les librairies sont plus lourdes (environ facteur 6 pour QtCore) ;
- Le mode **release** : l'inverse du mode *debug*. Les librairies sont plus légères mais donnent moins d'informations.

D'une manière générale, lorsque vous téléchargez une librairie, vous obtenez la version *release* (s'il y avait un mode *debug*). En effet, une librairie en mode *debug* ne donne des informations que sur elle-même en cas de problème interne, donc ça importe peu pour la plupart des utilisateurs. Cependant, il reste toujours possible de récupérer la version *debug* d'une librairie.
Comme la librairie Qt se veut « libre », et qu'on peut la recompiler à volonté, les créateurs ont aussi donné accès aux versions *debug*, celles-ci étant incluses dans les « packages ».

Pour savoir si vous utilisez une librairie en mode *release* ou non sous Qt, regardez le nom de la dll/so : si elle finit par un « d », sans tenir compte du chiffre (exemple « QtCored4.dll »), c'est qu'elle est en mode *debug*.

Changer le mode

Comme vous pouvez le constater, il peut être fort utile de pouvoir compiler à volonté soit en mode *debug*, soit en mode *release*. D'ailleurs, le changement se fait non seulement pour les librairies de Qt dont vous avez besoin, mais cela permet deux-trois choses dans votre propre programme ou votre propre librairie que nous verrons légèrement plus tard.

Pour changer de mode, c'est très simple. Si vous avez suivi le tutoriel de M@teo21 sur Qt, tâchez de vous rappeler comment on compile un projet. En fait, ça se résumait à trois « instructions » dans la console (dans le cas d'un nouveau projet). Les voici :

Code : Console

```
qmake -project
qmake
make
```

Si on prend les instructions une à une, voici ce qu'elles font :

- `qmake -project` : Qt génère automatiquement votre fichier de projet (.pro) en fonction des fichiers présents ;
- `qmake` : Qt lit votre fichier projet (.pro) et génère les *makefiles* ;
- `make` : Qt utilise MinGW (le compilateur) et les *makefiles* pour créer le fichier de sortie (en général un exécutable).

La modification que nous devons apporter se situe entre la première et la deuxième étape. En effet, le mode de compilation se décide dans le fichier de projet.
Si vous l'ouvrez (avec le bloc-notes par exemple), vous devriez obtenir quelque chose comme ceci :

Code : Autre

```
#############################################################################
# Automatically generated by qmake (2.01a) lun. 23. mars 22:46:38 2009
#############################################################################

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

# Input
SOURCES += main.cpp
```

Pour ceux que ça intéresserait, la documentation de Qt sur le qmake est très bien fournie. Et le réglage d'un projet Qt permet de faire de grandes choses.

Pour le réglage du mode de compilation, tout se passe avec la variable CONFIG. Comme celle-ci est déjà pré-remplie (même si on ne la voit pas), il faudra employer le symbole += pour indiquer qu'on souhaite ajouter des propriétés.

Si vous souhaitez faire de la compilation en mode *debug*, vous avez deux choix :

- ne rien faire. Par défaut Qt est réglé sur le mode *debug*. Mais il est possible que ça change en fonction des versions (les versions antérieures de Qt étaient en mode *release* par défaut) ;
- ajouter « debug » après le += de la variable CONFIG.

Et si vous préférez le mode *release*, il suffit de mettre « release » à la place de « debug ».

Facile, non ?

Si vous vous amusez à placer les deux en même temps, c'est celui placé en dernier qui est pris en compte.

Quel que soit le mode de compilation choisi, vous pouvez à tout moment ajouter aussi « console ». Cela force — dans le cas d'une application graphique — à ouvrir une console en arrière-plan. C'est très pratique pour voir vos propres messages de débuggage (on va voir dans un instant comment faire) mais aussi pour voir ceux de Qt, notamment concernant les signaux ou les slots qui ne se sont pas connectés ensemble (quelle qu'en soit la raison).

Le mode « console » n'a d'effet que sous Windows. Pour les autres, tout dépend de la manière dont vous lancez le programme.

Imaginons qu'on se place en mode *debug*. Voici ce qu'on obtient :

Code : Autre

```
#############################################################################
# Automatically generated by qmake (2.01a) lun. 23. mars 22:46:38 2009
#############################################################################

TEMPLATE = app
CONFIG +=  console debug
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
```

```
SOURCES += main.cpp
```

Vous pouvez voir que j'ai aussi ajouté le mot « console ».

Mode *debug* et *release* en même temps

En fait, il est possible de considérer qu'il existe un troisième mode de compilation… Celui-ci est une combinaison des deux autres car il effectue les deux compilations en même temps (enfin l'une après l'autre, sans pause).
Pour cela, remplacez le « debug » ou le « release » par « **debug_and_release** ».

La documentation de Qt dit qu'il est possible que l'utilisation de ce mode puisse causer des effets inattendus. Je vous conseille donc de faire très attention quand vous vous lancez dans ce genre de choses.

Dans ce cas, pour compiler, il vous faudra faire `make all` à la place de `make`.
Cependant, il reste possible d'utiliser `make` pour compiler les deux modes. Pour faire ça, ajoutez en plus de « debug_and_release » un « **build_all** ». Voilà, maintenant vous pouvez compiler les deux modes à la suite avec un simple `make`.

Prêt(e) à entrer dans le débuggage profond ?

Afficher des messages de débuggage

Enfin, on attaque la partie vraiment pratique.

Avant que tu ne commences… Pourquoi présenter des fonctions faites par Qt alors qu'un simple « cout » suffirait (maintenant qu'on sait afficher la console) ?

Très bonne question. En fait absolument rien ne vous empêche d'employer le bon vieux (mais fidèle) « cout ».
Mais voici deux-trois raisons de ne PAS l'utiliser :

- Une question « d'homogénéité » : dans tout votre programme on utilise Qt, alors pourquoi juste pour le débuggage utiliserait-on la bibliothèque standard ?
- Un grand nombre de classes Qt intègrent déjà un opérateur de flux vers la console pour se décrire elles-mêmes. C'est mieux que de devoir toutes les réécrire pour cout.

Après c'est à vous de voir ce que vous préférez.

Les fonctions de base

Qt intègre quatre fonctions dédiées au débuggage et aux rapports d'erreurs. Chacune a un but défini. Les voici :

- qCritical : le simple rapport d'erreur, qui ne fait pas obligatoirement quitter le programme (voir **qFatal**) ;
- qDebug : taillée exprès pour le débuggage ;
- qFatal : identique à **qCritical**, mais implique en générale que le programme se termine juste après (une erreur fatale donc ) ;
- qWarning : pour les messages de prévention.

Je vous invite vivement à consulter la doc les concernant, c'est très utile.
Par défaut, les messages sont envoyés sur « stderr » (le flux standard des erreurs). Dans le cas d'une application graphique sans console, c'est envoyé vers le débugger (s'il est présent, sinon adieu le message )

ATTENTION : Ces fonctions sont très très basiques et fonctionnent comme les (très vieilles) fonctions du C, c'est-à-dire du genre de la fonction « printf » (ouh, que c'est vieux). C'est-à-dire une chaîne de caractère avec des « %s » pour les chaînes de caractères, « %d » pour les entiers, etc.
De plus elles vont automatiquement à la ligne. Donc pas besoin de finir avec des « \n ».

QUOI ? Mais mais… et la classe QString ?

Je pense que les programmeurs de Qt (merci à Nanoc), lorsqu'ils ont créé ces fonctions, ont réfléchi à la chose suivante : si on emploie des fonctions comme **qCritical** ou **qFatal**, c'est qu'*a priori* quelque chose ne va pas. Si on utilise la classe QString alors que quelque chose ne va pas, QString risque aussi d'avoir un problème. Et là on va avoir de gros problèmes car une erreur se produit pendant le rapport d'une autre erreur. Et cette erreur va elle aussi être rapportée en appelant QString, qui provoquera une nouvelle erreur qui…

Assez peu pratique, non ? Et puis dans le cas où il se produit une erreur dans QString, on ne peut pas appeler les fonction prévues car celle-ci fonctionne avec QString.

Cependant, dans le cadre de vos projets, vous partez de l'hypothèse que Qt fonctionne correctement. De toute manière, s'il y a un problème interne, c'est Qt qui gère. Donc rien ne vous empêche d'utiliser QString avec ces fonctions. Et comme nous parlons de Qt, ils ont quand même prévu les choses, et mettent donc à notre disposition deux mécanismes pour nous simplifier la vie.

Simplifiez-vous la vie

qPrintable

Certains connaissent déjà cette fonction miracle : qPrintable. On en peut pas faire plus simple : vous lui donnez en paramètre une QString et elle vous sort... un « char * ». Parfait pour le donner à votre fonction.

Voyons un exemple :

Code : C++

```
qFatal(qPrintable(tr("Erreur fatale : la variable %1 n'existe pas").arg(i)));
```

Comme vous pouvez le voir, ça permet aussi d'utiliser directement la fonction « tr » (pour d'éventuelles traductions !).

Personnellement, je vous conseille de créer une petite macro, du genre :

Code : C++

```
#define FATAL_ERROR(msg) qFatal(qPrintable(msg))
```

Comme ça, plus de problème.

Passons donc au deuxième mécanisme (le meilleur).

QtDebug

Comme je vous l'ai dit légèrement plus haut, un des avantages de Qt c'est que nombre de ses classes savent se représenter seules dans une console. Pourtant aucune ne semble avoir de fonction du style `char * debug () const`. Alors comment font-elles ?

En fait il est possible d'utiliser nos quatre fonctions comme « cout », c'est-à-dire avec des opérateurs de flux. Pour cela, il suffit d'inclure « QtDebug » dans votre fichier :

Code : C++

```
#include <QtDebug>
```

Ceci ne fonctionne PAS pour la fonction `qFatal`. En effet celle-ci est là pour afficher un ultime message d'adieu avant la fin brutale du programme. On n'est pas là pour décrire toutes les variables du programme.

Rien ne vaut un bon exemple, donc :

Code : C++

```
qDebug() << "Début du programme, il est " <<
```

```
QTime::currentTime().toString("HH:mm::ss");
```

Notez la présence des parenthèses après l'appel de la fonction. Elles sont obligatoires. Comme leur version « de base », ces fonctions vont automatiquement à la ligne.

Les fonctions supra-complexes

Comme chacun le sait peut-être, Qt ne fait pas dans les demi-mesures. Donc on dispose encore de deux fonctions… très particulières. Celles-ci font partie intégrante de la classe QObject, donc tout le monde (ou presque) en dispose :

- dumpObjectInfo : affiche des informations sur l'objet, notamment sur les signaux ou les slots.
- dumpObjectTree : affiche un espèce d'arbre de l'objet et des ses enfants.

À vous de tester !

Avec ça, on a déjà de quoi réfléchir, non ? Alors pour ceux qui ont réussi à digérer, voici la suite.

Rediriger les flux

Comme je l'ai dit, les messages sont expédiés sur *stderr*. De ce fait, dans le cas d'une application graphique, on est obligé de faire afficher la console. Ce n'est pas très beau (surtout lorsqu'elle disparaît dès la fin du programme).

Le truc génial, ça serait de pouvoir expédier le tout dans un fichier. Mais à faire pour les quatre fonctions ça risque de se révéler pénible. Heureusement, grâce à Qt, on peut rediriger chaque fonction vers le même fichier ou chacun dans le sien. On peut même inserer du texte « obligatoire » (comme « WARNING : » pour chaque appel à `qWarning`).

La méthode n'est peut-être pas des plus claires, mais elle marche. Pour commencer, il faut déclarer une fonction ayant ce prototype (**rappel :** seul le nom de la fonction peut changer) :

Code : C++

```
void myMessageOutput (QtMsgType type, const char *msg)
```

Ensuite, c'est dans cette fonction que vous déciderez de faire ce que vous voulez des messages, etc.

Pour mettre en application cette fonction, il suffit de faire :

Code : C++

```
qInstallMsgHandler(myMessageOutput);
```

De préférence dans le *main*, avec `QApplication`.

Le mieux est de vous mettre l'exemple de Qt (visible ici) :

Code : C++

```
#include <qapplication.h>
#include <stdio.h>
#include <stdlib.h>

 void myMessageOutput(QtMsgType type, const char *msg)
 {
     switch (type) {
     case QtDebugMsg:
         fprintf(stderr, "Debug: %s\n", msg);
```

```
            break;
        case QtWarningMsg:
            fprintf(stderr, "Warning: %s\n", msg);
            break;
        case QtCriticalMsg:
            fprintf(stderr, "Critical: %s\n", msg);
            break;
        case QtFatalMsg:
            fprintf(stderr, "Fatal: %s\n", msg);
            abort();
        }
    }

    int main(int argc, char **argv)
    {
        qInstallMsgHandler(myMessageOutput);
        QApplication app(argc, argv);
        ...
        return app.exec();
    }
```

Voilà, vous savez comment détourner les messages d'erreur pour les expédier où vous voulez (même dans le néant).
Dans la prochaine partie, je vais vous donner encore deux conseils pour mieux débugger, puis je vous laisse vous débrouiller.

Un code unique quel que soit le mode de compilation

Tout au long de ce tutoriel, il y a quelque chose dont je n'ai pas parlé, mais que j'aimerais quand même vous présenter car vous allez forcément tomber dessus un jour.

Le problème

Imaginez que vous avez fait un superbe programme, et en vous souvenant de ce tutoriel, vous l'avez truffé de `qDebug` et `qWarning`.
Maintenant que vous l'avez débuggé à 100 %, vous voulez faire le compiler en mode *release*. Rien de plus facile, vous changez votre projet et hop, c'est fait.

Et où est le problème ?

Maintenant exécutez le programme. Et, oh, stupeur, les messages s'affichent aussi. Pas très joli pour la première version de votre super programme.
Bien évidemment, vous pouvez les enlever puis recompiler. Mais il va falloir le faire à chaque fois... Un peu pénible, avouez-le.
Pour pallier ce problème, il existe deux solutions.

Les solutions

La première consiste à modifier votre projet en ajoutant à *DEFINES* (comme pour CONFIG) ceci : **QT_NO_DEBUG_OUTPUT** et **QT_NO_WARNING_OUTPUT**. Ça va enlever les messages, mais ça ne marche que pour `qDebug` et `QWarning`.

La deuxième provient d'une simple observation : lorsque vous compilez en mode *release*, le « #define » **QT_NO_DEBUG** est passé à l'ensemble de vos fichiers.
Si on résume, vous disposez d'un « #define » automatique dans le cas d'une compilation en mode *release*. Dans les autres cas, il n'est pas là. Il faut donc, dans votre code, faire ceci pour chaque message :

Code : C++

```
#ifndef QT_NO_DEBUG
    qDebug("...");
#endif
```

Dans la mesure où ça peut être très fastidieux, je vous propose (vous faites comme vous l'entendez) de procéder comme suit, dans un fichier .h inclus partout :

Code : C++

```
#ifdef QT_NO_DEBUG
#define debug(msg)
#else
#define debug(msg) qDebug(qPrintable(msg));
#endif
```

Analysons le code rapidement : dans le cas où vous êtes en mode *debug*, tout va bien, la fonction « debug » est remplacée par `qDebug`. Lors de l'exécution, les messages s'affichent correctement.

Mais si vous passez en mode *release*, lorsque vous appelez la fonction « debug », elle est remplacée par… rien ! Bien évidemment les messages ne s'affichent plus lors de l'exécution (et pour cause !).

De cette manière donc, vous pouvez utiliser la fonction « debug » directement avec les QString et n'afficher les messages que si vous êtes en mode *debug*.

Fonctions de débuggage

J'avais promis d'en finir, mais je ne peux m'empêcher de vous parler d'elles…

Ces deux fonctions n'ont d'effet qu'en mode *debug*, et ce de manière automatique, donc pas besoin de faire de macros.

Même si la documentation est limpide en ce qui les concerne, nous sommes sur le Site du Zéro, donc je vais légèrement parler d'elles.

Elles sont là pour faire de simples tests et vérifier que tout va bien (elles sont souvent employées pour l'opérateur [] pour tester si l'index est dans les limites autorisées). Je vous demande donc ~~d'applaudir~~ d'accueillir `Q_ASSERT` et `Q_ASSERT_X` !

Q_ASSERT

Très basique. Vous lui donnez comme seul paramètre un test. Si le test est vrai, tout va bien, le programme se poursuit. Par contre, s'il a le malheur d'être faux… il est mal, très mal !

En effet, Qt ne plaisante pas avec `Q_ASSERT`. Si le test est faux, ça affiche automatiquement un petit message et pfiout… terminé le programme (d'ailleurs, il arrive à Windows de se plaindre de cette fin soudaine avec un deuxième message).

Le message possède une forme standard, il ressemble donc à ceci :

Code : Console

```
ASSERT: "Test" in file main.cpp, line 10
```

« Test » est bien sûr remplacé par votre test, ainsi que le nom du fichier et la ligne.

Pour ceux qui connaissent, cette fonction est identique à la fonction `assert` du C++ (seul le texte en sortie change très légèrement). Le choix de la fonction à employer ne dépend que de vous.

Bien que ce soit parfaitement compréhensible par un programmeur, le jour où un utilisateur lambda tombe dessus… Il est bien embêté.
C'est pourquoi nous passons à la deuxième candidate.

Q_ASSERT_X

Il s'agit en fait de la grande sœur de `Q_ASSERT`. En plus du test, elle accepte deux paramètres supplémentaires, qui peuvent facilement être décrits par *où* et *quoi* :

- **Où** : ça vous permet de dire où s'est produite l'erreur (en un peu mieux que « fichier "blabla" à la ligne 10 »). En règle générale, on met le nom de la fonction ainsi que le nom de l'objet (en reprenant un morceau du code source de Qt) : « QVector::at » ;
- **Quoi** : qu'est-ce qui est à l'origine du problème, par exemple « index invalide ».

Le C++ ne dispose pas d'une fonction équivalente !

Personnellement, j'ai plus tendance à employer la deuxième fonction, mais c'est vous qui décidez.

Rien ne parle mieux qu'un exemple, donc le voici :

Code : C++

```
#include <QCoreApplication>

int main (int argc, char **argv)
{
 QCoreApplication app(argc, argv);

 int * valeurs = new int[3];

 valeurs[0] = 0;
 valeurs[1] = 1;
 valeurs[2] = 2;

 int i = 0;
 while(i <= 3)
 {
  Q_ASSERT_X(i >= 0 && i <= 2, "main", "Valeur invalide pour 'i'");

  qDebug(qPrintable(QString("valeurs[%1] =
%2").arg(i).arg(valeurs[i])));
  i++;
 }

 return app.exec();
}
```

Code : Console

```
valeurs[0] = 0
valeurs[1] = 1
valeurs[2] = 2
ASSERT failure in main: "Valeur invalide pour 'i'", file main.cpp, line 16
```

Bien évidemment, il s'agit d'un pur « cas d'école », le code ici présent ne servant pas à grand-chose.
Vous pouvez aussi voir que la fonction `Q_ASSERT_X` affiche toutes les informations nécessaires et quitte toute seule si le test échoue.
C'est fini ! FI-NI !

Vous voilà ~~enfin~~ déjà au bout de ce tutoriel. J'espère que mes explications sur le débuggage ont été comprises et surtout utiles. Prenez bien le temps de relire, n'hésitez pas à repasser pour vérifier (j'ai moi-même été pris par de solides doutes durant l'écriture de ce tutoriel)

Maintenant je ne veux plus voir que des programmes qui écrivent deux pages A4 par fonction en mode *debug*.

Sur ce, je vous laisse retourner à vos chers programmes !

Ce tutoriel a été corrigé par les zCorrecteurs.