De QObject aux méta-objets, une plongée au cœur des fondations de Qt

par Louis du Verdier

Date de publication : 09/02/2010

Dernière mise à jour : 20/06/2010

Qt est un framework dont les fondations ne sont pas connues de la totalité des gens qui l'utilisent. Bien entendu, il est possible de se demander en quoi un codeur d'interfaces graphiques peut avoir besoin d'utiliser *explicitement* les méta-objets dans son code. Il est toujours intéressant de savoir sur quoi est fondé un framework. Quel est le modèle objet utilisé ? Que sont les méta-objets ? Ce sont des questions auxquelles cet article a pour but de répondre.

N'hésitez pas à commenter cet article !

0 - Introduction

Comme vous devez l'avoir compris en vous lançant dans la lecture de cet article, j'ai pour objectif de vous en apprendre plus sur les fondations de Qt, les piliers mêmes de ce framework (ensemble de bibliothèques) aux allures si prometteuses. Pour arriver à mes fins, je vous ferai étudier une grande partie de ce qui existe, depuis la classe QObject jusqu'aux méta-objets, en passant bien sûr par les signaux, les slots et bien d'autres choses encore.

Comme cet article a pour but d'être informatif, nous verrons uniquement des extraits de code permettant d'illustrer ce qui y est dit, donc de servir de support aux explications. Ainsi, nous n'étudierons pas ou uniquement à titre exceptionnel des « gros » codes.

Je précise que, même si je vous transmettrai ici de nombreuses connaissances, je ne pourrai me montrer parfaitement exhaustif. Toutefois, si vous relevez dans cet écrit une erreur, un oubli quelconque méritant d'être corrigé ou autre, je vous prie de bien vouloir me contacter par **message privé**, je serai ravi de pouvoir enrichir son contenu.

I - À propos de QObject

I-A - Pourquoi QObject est-il à la base de Qt ?

Comme vous le savez déjà, Qt est orienté objet et qui dit POO (Programmation Orientée Objet) dit forcément héritage. Le principe de l'héritage est plus ou moins le même que celui d'un arbre généalogique comportant des classes, qui héritent elles-mêmes d'autres classes : nous en arrivons forcément à une classe de base située à la racine de cet arbre, que l'on appelle aussi classe mère ou classe parente. Dans le cas de Qt, il s'agit de QObject.

Cette classe a pour avantage de ne correspondre à rien de « graphique ». Cela permet à Qt de gagner en puissance car elle fournit au final nombre d'utilitaires, comme le système de méta-objets dont nous allons longuement parler dans cet article sans pour autant être exhaustif, comme dit dans l'introduction. Ces utilitaires sont pour la plupart très ciblés sur le concept d'objet. QObject a donc indéniablement plus sa place que les autres classes actuellement existantes comme pilier central du framework Qt.

QObject, même si c'est la classe mère, n'est pas la classe parente de toutes les autres : certaines d'entre elles, comme QString, n'ont pas besoin des outils qu'elle propose.

I-B - Qu'en est-il du modèle objet de Qt ?

À la différence du C++, de nature plutôt statique, Qt fournit une efficacité d'exécution nécessaire à la réalisation d'une interface utilisateur, à un niveau de flexibilité relativement élevé.

Le modèle objet de Qt, axé sur QObject et sur le Moc, qui sera détaillé plus tard dans l'article, offre de nombreux avantages et utilités dont les principaux sont les suivants :

- une gestion de la communication entre les objets par le biais d'un système le plus souvent appelé mécanisme de signaux et de slots (partie IV) ;
- une hiérarchie logique aux utilités multiples entre les objets, telles que les conversions de type (partie III) ;
- un système événementiel relativement utile lors du développement d'interfaces utilisateur (Qt est parfois considéré comme événementiel) ;
- une gestion de la mémoire aisée pour les développeurs, notamment par une destruction de la totalité des objets enfants lorsque l'objet parent est détruit ;
- l'introspection et les propriétés (partie V).

La documentation insiste sur le fait que les objets de Qt doivent être considérés comme des éléments possédant une identité, non comme des valeurs. Malgré les apparences, il faut noter que le modèle objet de Qt se base comme n'importe quelle application codée en C et/ou en C++ sur le C++, ce qui explique en partie la portabilité du framework.

II - Qu'est-ce que Q_OBJECT ?

Q_OBJECT est une macro, un pseudo-programme permettant par le biais d'un identifieur de remplacer des morceaux de codes tels que des defines par des valeurs, fonctions ou autres. Cette macro permet à la classe la contenant dans sa section privée de posséder un méta-objet, à condition qu'elle hérite directement ou non de QObject (pour le cas où la classe ne serait pas dérivée de QObject, il existe aussi la macro Q_GADGET, toutefois très peu documentée).

Le fait de l'employer à l'intérieur d'une classe permet l'utilisation de ce que fournit le système de méta-objets, comme par exemple les signaux et les slots qui ne seraient pas disponibles sans cela. Si vous souhaitez en apprendre plus sur ce que fournit la macro, consultez la partie V qui traite le sujet des méta-objets.

Il faut noter que certaines méthodes comme tr(), inherits() et autres nécessitent la présence de la macro pour fonctionner correctement, et c'est pour cela qu'il est recommandé à plusieurs reprises par la documentation de l'utiliser « dans toutes les sous-classes de QObject sans prêter attention au fait qu'elles utilisent actuellement ou non les signaux, les slots et les propriétés ».

III - Une question d'héritage

Veuillez noter avant tout que chaque partie de cet article est unie par les méta-objets. Cette partie de l'article est donc en continuité avec le reste de celui-ci.

III-A - Conversions de type et QObject

À titre de rappel, une conversion de type avec Qt retourne, en cas de réussite, un objet de type particulier d'un objet subissant l'opération. Par rapport aux conversions de type du C++, celles de Qt sont légèrement différentes car elles ne nécessitent pas le support **RTTI** (Run Time Type Information).

Dans cette sous-partie, nous allons considérer deux exemples de conversions de type assez similaires :

- qobject_cast ;
- qvariant_cast.

Quel est le but de ces deux présentations ? Tout simplement de vous montrer l'utilité de Q_OBJECT pour pouvoir aborder le concept de méta-objet sans pour autant que vous vous heurtiez à l'inconnu.

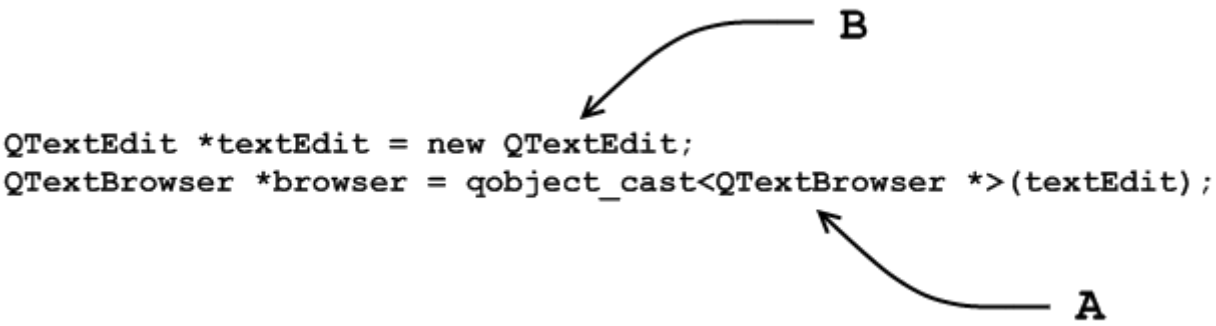
III-A-1 - La conversion de type qobject_cast

```
A objet = qobject_cast<A>(B) ;
```

Cette conversion de type est un cast de pointeur qui fonctionne à travers les branches de l'héritage : elle retourne un objet de type A à partir d'un objet de type B, mais sous certaines conditions. Tout d'abord, le type A doit être un type dérivé du type B. Par exemple, le code suivant n'est pas correct :

```
QTextEdit *textEdit = new QTextEdit;
QTextBrowser *browser = qobject_cast<QTextBrowser *>(textEdit);
```

Dans ce cas, la conversion de type retournera 0 car la condition énoncée ci-dessus n'est pas respectée : QTextEdit n'est pas une classe dérivée de QTextBrowser, c'est l'inverse, donc une incompatibilité est présente.



Toutefois, une conversion de type est réversible, elle permet de repasser d'un type à l'autre, à condition que la conversion précédente n'ait pas retourné 0. La seule nécessité est donc qu'il y ait un rapport d'héritage entre les deux types.

```
QTextEdit *textEdit = new QTextEdit;
QObject *cast_textEdit = qobject_cast<QObject *>(textEdit);
QTextEdit *recast_textEdit = qobject_cast<QTextEdit *>(casted_textEdit);
```

Ce qu'il faut tout de même savoir est que qobject_cast ne fait pas de distinction entre les types personnalisés et les types inclus par défaut avec Qt (exemple : QTextEdit).

Il peut être intéressant de présenter ici la fonction QObject::inherits() qui retourne un booléen dont la valeur dépend de l'héritage d'une classe ou non.

```
QTextEdit *textEdit = new QTextEdit;
bool a = textEdit->inherits("QWidget"); // a == true
bool b = textEdit->inherits("QTextBrowser"); // b == false
```

La seconde condition est que la classe A soit déclarée avec la macro Q_OBJECT. Dans le cas inverse, la valeur de retour serait indéfinie, si le compilateur ne signalait pas une erreur. En effet, qobject_cast requiert les méta-informations fournies par le système de méta-objets pour fonctionner correctement.

D'un point de vue pratique, qobject_cast est très utile lors de l'utilisation de sender(). Placé dans un slot, cette fonction renvoie un QObject de l'objet ayant envoyé le signal l'appelant et la conversion de type qobject_cast permet d'en récupérer un widget du type d'un QPushButton. Nous verrons plus amplement cela dans la partie sur les signaux et les slots.

III-A-2 - La conversion de type qvariant_cast

QVariant est une classe permettant de faire le lien entre la plupart des types de données, tels que uint, bool, QString, etc. Elle permet par le biais de fonctions telles que toString() d'exprimer sa valeur sous différents types de données, mais dès qu'il s'agit de passer de QVariant à QColor, par exemple, et bien il faut nécessairement passer par une conversion de type, qui est ici qvariant_cast. On aurait bien évidemment pu passer par autre chose mais je préfère ne parler que de cela ici.

D'un point de vue syntaxique, cette conversion de type est identique à qobject_cast, mais il s'agit bien de l'un des rares points communs entre les deux conversions de type. En effet, elle ne prend pas en compte l'héritage car il est impossible de parler de cela pour des types comme int et bool. Comme l'héritage n'a aucun rôle ici, le système de méta-objets n'en a donc pas non plus. Ainsi, la macro Q_OBJECT n'influe en rien sur qvariant_cast. Au final, on ne parle que de compatibilité entre les types. Si cette compatibilité n'est pas présente, la valeur de retour de la conversion sera, comme pour qobject_cast, 0.

```
QVariant color = QColor(0, 0, 0, 0);
qDebug() << qvariant_cast<QColor>(color); // QColor(ARGB 0, 0, 0, 0)
qDebug() << qvariant_cast<QString>(color); // #000000
```

```
qDebug() << qvariant_cast<quint32>(color); // 0
```

On peut donc déduire de la présentation des deux conversions de type que même si le système de méta-objets est relativement présent, Qt ne repose pas totalement dessus.

IV - Allons plus loin avec les signaux et les slots

IV-A - Le principe

Depuis sa création, Qt propose un mécanisme ingénieux couramment appelé système de signaux et de slots. Ce système est relativement pratique quand il est présent dans des applications graphiques car il permet la communication entre plusieurs objets. Comme la documentation recommande de parler d'eux comme des identités, non comme des valeurs, nous pouvons considérer l'exemple d'un bouton sur lequel l'utilisateur cliquerait. Si l'intérêt de la présence de ce bouton était de quitter l'application lors d'un clic dessus, l'interaction entre ce bouton et l'objet de l'application, qApp, serait d'appeler une fonction de fermeture, quit(). Dans ce cas de figure, l'objet *émetteur* serait le bouton et l'objet *receveur* serait qApp. Pour créer une interaction entre ces deux objets, il faudrait créer une connexion entre eux. Par chance, Qt fournit la méthode statique et thread-safe connect() qui permet de mettre en place ce type de connexion.

```
connect(émetteur, signal, receveur, slot) ;
```

Comme vous pouvez le constater, connect() prend quatre arguments dont nous connaissons l'émetteur et le receveur, mais il accepte aussi un cinquième, le type de connexion, dont nous ne parlerons pas dans cette partie. L'argument signal correspondrait dans l'exemple du bouton à un clic, donc au signal clicked() de QPushButton. Un signal peut être considéré comme une condition à l'appel de son slot. Quant à lui, le slot correspondrait à l'action produite lors d'une interaction entre l'émetteur et le receveur, soit ici à la fermeture de l'application, donc à l'appel de la fonction quit(). Plus simplement dit, c'est une fonction appelée lors de l'interaction.

```
QPushButton *bouton = new QPushButton("Quitter");
connect(bouton, SIGNAL(clicked()), qApp, SLOT(quit())) ;
```

Une connexion entre deux objets requiert le système de méta-objets, donc la présence de la macro Q_OBJECT dans la section privée de la classe où est faite la connexion. Toutefois, le slot quit() est un slot implémenté par défaut avec Qt et non un slot créé par l'utilisateur. Le système de méta-objets n'est donc pas nécessaire pour ce cas de figure. Sans le système de méta-objets, connecter deux objets par le biais d'un signal personnalisé et/ou un slot personnalisé n'aura aucun effet. Notez que le nombre de connexions d'un objet avec un autre n'est pas limité.

De la même manière que l'on peut connecter entre eux deux objets, il est possible de les déconnecter. Cette fois-ci, c'est la fonction disconnect() qui entre en jeu, et avec les mêmes arguments que ceux entrés dans la fonction connect().

```
disconnect(bouton, SIGNAL(clicked()), qApp, SLOT(quit())) ;
```

Comme vous pouvez le constater dans les codes présentés, il n'est pas nécessaire que l'émetteur reçoive des informations de la part du receveur et vice versa : les deux objets restent indépendants l'un de l'autre lors d'une connexion. Il est d'ailleurs possible d'émettre une communication de valeurs lors d'une connexion/déconnexion. Un exemple qui illustrerait assez bien le principe serait une connexion entre un slider et une barre de progression : quand la valeur du slider changera, la barre de progression prendra la même valeur que celle du slider :

```
QSlider *slider = new QSlider(Qt::Horizontal);
QProgressBar *progressBar = new QProgressBar;
connect(slider, SIGNAL(valueChanged(int)), progressBar, SLOT(setValue(int)));
```

Il faut garder en tête que le mécanisme de signaux et de slots est extrêmement flexible. Selon la documentation, il est possible d'émettre environ deux millions de signaux par seconde, quand ils sont connectés à un même receveur, sous un i586-500 (la 5° génération de Pentium).

IV-B - Signaux et slots personnalisés

Jusqu'alors, nous n'avons vu que des signaux et des slots présents par défaut avec Qt. Ces signaux, même s'ils sont utiles, ne sont pas suffisants pour combler tous les besoins des programmeurs. C'est pour cela qu'il est possible de créer et de gérer ses propres signaux et slots. Comme dit dans la sous-partie précédente, l'utilisation de signaux et/ou de slots personnalisés nécessite la présence de la macro Q_OBJECT dans la section privée de la classe où ils sont gérés.

IV-B-1 - Les signaux personnalisés

La création de signaux se fait lors de la déclaration de la classe. On déclare un signal comme on déclare une fonction, soit par un simple prototype. La différence entre les deux est que le programmeur ne gère pas l'implémentation d'un signal, c'est le système de méta-objets qui s'en occupe. En C++, une fonction se déclare dans le header sous le mot-clé public ou private. Avec Qt, un signal se déclare sous un autre mot-clé : signals.

```
class Label : public QLabel
{
 Q_OBJECT
 signals:
 void released();
 // Reste de la déclaration
};
```

J'ai choisi de vous présenter un exemple de classe dérivée de QLabel pour vous montrer comment créer un signal released(), déclenché lorsque le bouton de la souris est relâché après un clic sur celui-ci. J'aurais pu choisir de vous présenter un exemple avec un signal nommé pressed() mais dans la majorité des cas, on préfèrera utiliser le relâchement au lieu de l'enfoncement. Pour cela, la seule chose à faire est d'appeler le signal released() par le biais d'une instruction particulière, emit, lors de l'appel de l'évènement mouseReleaseEvent().

```
void Label::mouseReleaseEvent(QMouseEvent *event)
{
 Q_UNUSED(event);
 emit released();
}
```

De là, il est possible d'effectuer une connexion des plus basiques :

```
connect(this, SIGNAL(released()), qApp, SLOT(quit()));
```

Bien entendu, il est possible de transmettre des valeurs lors de l'émission du signal personnalisé. Par exemple, dans le cas présenté, passer en argument la position de la souris lors de l'évènement est possible.

```
emit released(event->pos());
```

Le prototype devra, en conséquence, contenir un argument de type QPoint pour que la compilation se déroule sans problème. Ce qu'il faut tout de même retenir est qu'il est impossible de passer une valeur lors de la connexion, la valeur est uniquement transmise lors de l'émission du signal. Ce code est donc incorrect :

```
connect(this, SIGNAL(released(QPoint(12, 12))), qApp, SLOT(quit())); // Incorrect
```

Un équivalent correct de cette connexion pourrait uniquement être mis en place lors de l'appel de l'instruction emit, non ailleurs.

De même, il est incorrect de placer un nom de variable dans une connexion : il faut uniquement placer le type.

```
connect(this, SIGNAL(released(QPoint point)), qApp, SLOT(quit())); // Incorrect
```

Ainsi, la création de signaux est extrêmement simple : une déclaration et l'utilisation de l'instruction emit. Le système de méta-objets permet dans le cas présent au programmeur de ne pas avoir à implémenter le signal. Toutefois, si vous vous demandez où est placée cette fameuse implémentation, regardez le fichier moc_[Nom du fichier d'en-tête].cpp généré par le Moc (appelé explicitement : projet Visual Studio et autres, ou implicitement : qmake, etc.). Elle y est faite sous le même prototype que celui que vous avez déclaré dans votre header pour le signal dont il est question.

Maintenant que je vous ai parlé des signaux personnalisés, je peux vous parler d'un autre aspect de la fonction connect(), bien que nous n'en verrons pas la totalité : il est possible de connecter deux signaux entre eux. Une telle connexion ferait que lorsqu'un signal est émis, l'autre l'est aussi. Pour vérifier la véracité de mes dires, testez ces deux lignes (deux signaux y sont utilisés : released() et released(QPoint)) :

```
connect(this, SIGNAL(released(QPoint)), this, SIGNAL(released()));
connect(this, SIGNAL(released()), qApp, SLOT(quit()));
```

Lorsque le signal released(QPoint) est émis, released() l'est aussi, ce qui engendre la fermeture de l'application.

Une dernière chose sur les signaux : il est possible qu'à un moment ou à un autre vous souhaitiez, pour une raison ou une autre, qu'un objet cesse d'envoyer des signaux. Pour cela, il suffit d'appeler la méthode blockSignals() en passant en argument true. Dès lors, l'objet n'enverra plus aucun signal.

IV-B-2 - Et les slots personnalisés

Les slots ne sont rien de plus que des fonctions standards ayant la capacité d'être appelées par des signaux lors de connexions : ils peuvent sans problème être appelés normalement, être virtuels ou autres. Par conséquent, la création d'un slot se fait exactement comme la déclaration d'une fonction, donc avec un prototype et une implémentation, sauf que leur déclaration se situe sous l'institution public slots, private slots ou encore protected slots, afin de les différencier des fonctions normales.

```
public slots:
void about();
```

Dans le slot about(), présent en tant que slot public, il est par exemple possible d'ouvrir une boîte de dialogue, les possibilités ne manquent pas :

```
void MainWindow::about()
{
 QMessageBox::information(this, "Boîte de dialogue", "Cette boîte de dialogue "
   "correspond à une implémentation basique d'un slot personnalisé.");
}
```

Plus tôt, nous avons vu que les connexions entre signaux étaient possibles. Ce n'est pas le cas pour les slots car une connexion nécessite au minimum un signal. D'ailleurs, les slots ont eux aussi besoin du système de méta-objets pour fonctionner correctement : la présence de la macro Q_OBJECT est donc une fois de plus nécessaire.

Dans l'univers de Qt, il existe ce qu'on appelle communément des connexions automatiques. Ce type de connexions est, par exemple, présent dans les IU (Interface utilisateur) avec les slots dont les prototypes sont architecturés sous la forme suivante :

```
void on_[Nom de l'objet]_[Nom du signal]([Arguments]);
```

Nous ne verrons pas en détail ce type de connexions.

La méthode sender() a été introduite dans la partie sur la conversion de type qobject_cast. Comme elle fonctionne dans le cadre d'un slot, il peut être intéressant de la présenter dès maintenant. Cette méthode renvoie un QObject de l'objet ayant envoyé le signal appelant le slot dont il est question. Cette particularité permet d'effectuer de multiples connexions sur le même slot, ce qui peut être très pratique dans certains cas où les connexions seraient répétitives.

Voici un court exemple permettant de vérifier si le texte de l'émetteur d'un slot, présumé être un QLineEdit, correspond à un chemin de fichier existant :

```
void MainWindow::verif()
{
 QLineEdit *lineEdit = qobject_cast<QLineEdit *>(sender());
 if(lineEdit && QFileInfo(lineEdit->text()).exists())
 QMessageBox::information(this, "Vérification", "Le chemin est valide.");
}
```

IV-C - QSignalMapper

Il arrive parfois qu'un développeur utilisant Qt ait besoin d'effectuer de multiples connexions sur le même slot. L'alternative à effectuer des connexions fastidieuses vers des slots ayant le même effet ou bien à passer par la méthode sender() serait de passer par QSignalMapper, qui permet assez facilement de réaliser ce type de connexions, en renvoyant en même temps un identifiant de l'émetteur, tel un QString, un int, un QWidget ou encore un QObject. Retourner un QWidget ou un QObject peut être très pratique une fois combiné avec une conversion de type telle que qobject_cast.

Pour réaliser des connexions multiples sur le même slot avec QSignalMapper, la connexion de deux signaux peut être très intéressante à mettre en œuvre. Je pense que l'heure est venue de vous montrer un court exemple d'utilisation :

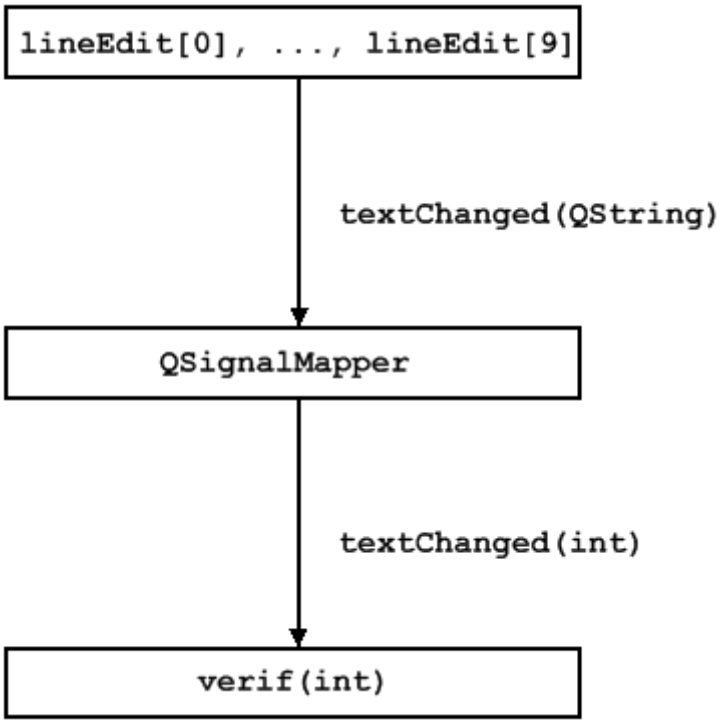
```
MainWindow::MainWindow()
{
 setCentralWidget(new QWidget);
 QVBoxLayout *layout = new QVBoxLayout;
 centralWidget()->setLayout(layout);

 QSignalMapper *mapper = new QSignalMapper(this);
 connect(mapper, SIGNAL(mapped(int)), this, SIGNAL(textChanged(int)));
 connect(this, SIGNAL(textChanged(int)), this, SLOT(verif(int)));

 for(int i = 0; i < 10; i++)
 {
  lineEdit.append(new QLineEdit(QString::number(i + 1)));
  mapper->setMapping(lineEdit[i], i);
  connect(lineEdit[i], SIGNAL(textChanged(QString)), mapper, SLOT(map()));
  layout->addWidget(lineEdit[i]);
 }
}

void MainWindow::verif(int index)
{
 QString text = lineEdit[index]->text();
 if(QFileInfo(text).exists())
 QMessageBox::information(this, "Vérification", "Le chemin du champ "
   + QString::number(index + 1) + " est valide.");
}
```

Ce code fonctionne avec trois éléments préalablement déclarés dans l'en-tête :

- le signal textChanged(int) ;
- le slot verif(int) ;
- la liste QList <QLineEdit *> lineEdit.

Dans mon code, je déclare un QSignalMapper qui permettra les multiples connexions grâce à sa capacité de constituer une interface entre plusieurs signaux et un slot possédant un seul paramètre. Grâce à la connexion présente dans la boucle for, lorsque le texte d'un des QLineEdit de la liste change, le slot map() est appelé. Le fait d'utiliser la fonction setMapping() avec comme second argument un int permet de faire en sorte que le signal mapped(int) soit émis lorsque map() est appelé. Pour finir, je connecte ensuite ce signal au slot verif(int).

Si on regarde le principe globalement, les choses se font comme sur le schéma suivant :



Dès lors, nous pouvons passer à la partie sans doute la plus importante de l'article, celle sur le système de méta-objets.

V - Au cœur des méta-objets

V-A - L'intérêt du système de méta-objets

Une grande question peut se poser : pourquoi utiliser un tel système ? En réalité, la principale raison de sa présence est le besoin d'instaurer un mode de communication entre les objets, qu'ils aient ou non un lien entre eux (nous verrons d'ailleurs, dans la partie suivante, que les conséquences de la présence d'un tel système sont les critiques dont il est l'objet). D'après ce que nous avons vu et ce que nous allons voir, le système de méta-objets est nécessaire aux dispositifs suivants :

- le système de signaux et de slots ;
- les informations de type disponibles lors de l'exécution, donc l'introspection ;
- le système de propriétés.

V-B - Méta-objets et templates

Le fait que Qt utilise un outil additionnel, le Moc (traité à la suite) et qu'il utilise une implémentation basée sur des macros fait l'objet de critiques, car il rend toute programmation avec Qt différente d'une programmation pure C++. Il s'avère que les développeurs de Qt voyaient cette utilisation comme une nécessité pour pouvoir fournir le mécanisme de signaux et de slots ainsi que les dispositifs introspectifs, non comme un moyen de se démarquer du C++.

Toutefois, certaines personnes pensent que les templates, déjà largement utilisés dans Qt, seraient plus appropriés pour gérer cela. Dans **cette page**, la documentation explique en détail pourquoi l'utilisation d'outils additionnels est préférable à l'utilisation des templates.

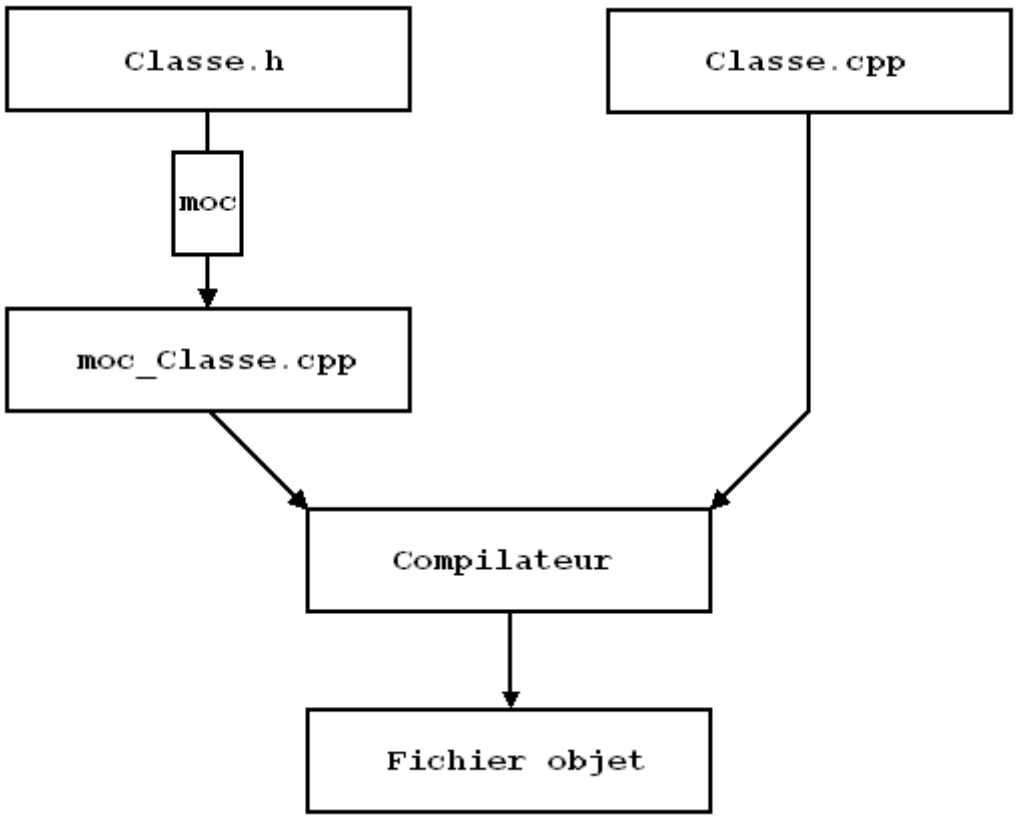
Vous pouvez aussi retrouver un débat complet sur la question **ici**. Ce débat a pour base une opposition de GTK et de Qt et s'oriente progressivement vers le sujet des méta-objets, en passant et en repassant par le sujet des signaux et

des slots. La principale critique émise est notamment le fait énoncé plus haut, précisant que toute programmation avec Qt est différente d'une programmation pure C++ (fait contesté : il est possible d'éviter de diverger d'une programmation pure C++). L'intérêt du Moc est remis en cause à plusieurs reprises.

V-C - Qu'est-ce que le Moc ?

Le Moc est le compilateur de méta-objets. Make se sert du Moc lorsqu'il est appelé, mais celui-ci peut aussi être appelé manuellement. Son fonctionnement est simple : il lit les fichiers sources et retrouve les déclarations de classe où la macro Q_OBJECT est utilisée. Une fois cela terminé, le Moc va générer des fichiers reconnaissables par leur nom sous la forme de moc_[Nom du fichier d'en-tête].cpp (dans le cas d'une utilisation de Q_OBJECT dans un fichier d'en-tête) ou bien sous la forme de [Nom du fichier].moc (dans le cas d'une **utilisation de la macro sans fichier d'en-tête**), qui contiendront le code de méta-objets, par exemple nécessaire à l'utilisation des signaux et des slots.

Voici un schéma illustrant les étapes de la compilation de deux fichiers respectivement nommés Classe.cpp et Classe.h avec un compilateur quelconque. Ce dernier fichier contient dans le cas présent, à la différence du fichier Classe.cpp, une déclaration de classe dans laquelle la macro Q_OBJECT est employée.



Bien entendu, le fichier objet de moc_Classe.cpp est lui aussi généré après l'utilisation du compilateur car ce fichier est bel et bien considéré comme un fichier source.

Ce système a beau être attrayant, il possède tout de même des points faibles. Par exemple, il ne gère pas correctement les classes templates qui se voient donc privées des signaux et des slots. La documentation détaille **ici** les limitations que cause le Moc.

V-D - Les propriétés

Les propriétés constituent un dispositif multiplateforme s'avérant être le pilier de la majorité des classes de Qt utilisant le système de méta-objets. Elles prennent d'ailleurs base sur ce système. Afin d'expliquer clairement comment s'en servir, nous allons étudier un petit code contenant une classe nommée MyWidget, héritant de QWidget et ayant la macro Q_OBJECT définie dans sa section privée. Ci-dessous se trouve une première utilisation des propriétés avec la propriété « windowTitle » :

```
MyWidget widget;
widget.setProperty("windowTitle", "MyWidget");
```

Ici, nous instancions notre classe MyWidget et nous définissons la propriété windowTitle avec la valeur MyWidget. Une alternative plus commune de l'utilisation de setProperty consiste à utiliser la fonction setWindowTitle. Toutefois, utiliser cette fonction à la place de setProperty ne contourne pas l'utilisation du système de propriétés, car elle s'avère être ce qu'on appelle la fonction d'écriture WRITE de la propriété windowTitle, dont nous allons parler à la suite.

Pour paraître plus concret, les propriétés constituent un outil utilisé par les modules QtScript et QtDBus pour déterminer ce qui peut être exporté depuis l'objet au script/à l'environnement D-Bus, ainsi que par le module QtDeclarative pour déterminer ce qui sera exporté depuis une classe dans QML.

Cette courte introduction a pour objectif de vous familiariser avec les propriétés. Nous allons voir en détail comment définir une propriété (et cela en premier lieu car nécessaire à la compréhension des propriétés fournies par défaut), comment gérer les valeurs d'une propriété, l'utilité des propriétés dynamiques et d'autres éléments. Par la suite, nous verrons l'intérêt des propriétés du point de vue des méta-objets.

V-D-1 - Comment définir une propriété ?

Une propriété peut être définie si et seulement si la classe concernée profite du système de méta-objets. La définition d'une telle chose s'effectue à l'aide de la macro Q_PROPERTY() dont le prototype fourni par la documentation est le suivant :

```
Q_PROPERTY(type name
  READ getFunction
  [WRITE setFunction]
  [RESET resetFunction]
  [NOTIFY notifySignal]
  [DESIGNABLE bool]
  [SCRIPTABLE bool]
  [STORED bool]
  [USER bool]
  [CONSTANT]
  [FINAL])
```

On peut donc considérer une déclaration de propriété comme celle-ci :

```
Q_PROPERTY(bool myProperty READ getValue WRITE setValue);
```

READ et WRITE sont en réalité des fonctions d'accès, respectivement associées à l'accès en lecture et en écriture. Elles permettent respectivement comme leur nom l'indique de récupérer la valeur de la propriété et de la modifier. Ces deux fonctions doivent respecter des règles :

- la fonction READ doit être du type de la propriété et ne doit pas prendre d'argument ;
- la fonction WRITE doit être de type void et doit prendre un seul argument du type de la propriété.

Ici, la fonction getValue devra donc être de type bool et ne prendre aucun argument. Quant à elle, la fonction setValue devra être de type void et prendre un argument de type bool.

```
class MyWidget : public QWidget
{
 Q_OBJECT
 Q_PROPERTY(bool myProperty READ getValue WRITE setValue);

 public:
 MyWidget();
 void setValue(bool);
 bool getValue() const;

 private:
 QVariant value;
};
```

Le QVariant value, qui aurait d'ailleurs pu être tout simplement un bool, a pour but de prendre la valeur de la propriété myProperty dans les implémentations des fonctions setValue et getValue. Je vous présente ces implémentations, en insistant sur le fait qu'il est parfaitement possible de procéder différemment :

```
void MyWidget::setValue(bool v)
{
 value = QVariant(v);
}
bool MyWidget::getValue() const
{
 return qvariant_cast<bool>(value);
}
```

En sachant que la fonction property, avec comme argument le nom de la propriété (ici, myProperty) retourne un QVariant de la valeur de la propriété, il est possible de s'interroger sur l'utilité de la variable value. Il se trouve que la fonction property se sert de la fonction READ (quand elle est présente) pour retourner une valeur. Si la fonction READ s'appuyait sur la fonction property, une boucle infinie serait créée, d'où l'intérêt de value.

V-D-2 - Changer la valeur d'une propriété

À moins que la propriété ait pour but d'être constante (donc à moins qu'il y ait l'attribut CONSTANT dans la déclaration de la propriété), il peut être intéressant de pouvoir changer la valeur d'une propriété.

Il existe deux méthodes pour changer la valeur d'une propriété :

- utiliser la fonction setProperty avec deux arguments, respectivement le nom de la propriété et sa nouvelle valeur ;
- utiliser tout simplement la fonction WRITE, quand elle existe, en passant en argument la nouvelle valeur.

```
MyWidget widget;
widget.setProperty("myProperty", true);
widget.setValue(true);
```

La fonction setProperty retourne un booléen du fait que l'assignation d'une nouvelle valeur ait réussi ou échoué.

Parfois, le programmeur a besoin de savoir quand la valeur d'une propriété change, et donc de récupérer un signal à chaque modification. C'est par exemple le cas de la propriété text de QLineEdit qui a pour signal de notification textChanged, avec pour argument un QString contenant la nouvelle valeur.

```
Q_PROPERTY(bool myProperty READ getValue WRITE setValue NOTIFY valueChanged);
// ...
signals:
void valueChanged(bool);
```

Le signal valueChanged doit être émis lors de la modification de la propriété, donc dans la fonction WRITE :

```
void MyWidget::setValue(bool v)
{
 value = QVariant(v);
 emit valueChanged(v);
}
```

Il faut toutefois noter que certaines propriétés n'ont pas de signal de notification, mais qu'il existe tout de même des signaux en rapport. Par exemple, les signaux pressed et clicked de QAbstractButton ne sont pas appelés lorsque la valeur de la propriété down change.

V-D-3 - Les propriétés dynamiques

Les propriétés dynamiques constituent en elles-mêmes de bonnes raisons de se servir des propriétés dans un projet. Le principal avantage fourni par les propriétés dynamiques est qu'elles peuvent être ajoutées *pendant* l'exécution. La documentation informe les utilisateurs de propriétés dynamiques qu'elles sont ajoutées à l'objet, et non au méta-objet, ce qui a pour conséquence que la propriété ne peut être retirée par invalidation depuis une instance.

L'ajout d'une propriété dynamique se fait par un appel de la fonction setProperty, en passant un nom de propriété n'existant pas déjà. Toutefois, cette fonction retournera forcément false, même si l'assignation de la valeur à la nouvelle propriété dynamique a été un succès.

Pour plus d'informations sur les propriétés, veuillez vous référer à **cette page**.

V-E - L'introspection

Un des grands avantages du système de méta-objets de Qt est de fournir un système d'introspection. L'introspection est la capacité d'un programme à s'examiner, à s'étudier. Pour être plus précis, il est possible de dire que le **RTTI** constitue en lui-même un sous-ensemble de l'introspection.

Le RTTI (soit *Run Time Type Information* en anglais et donc *Informations de Type disponibles à l'Exécution* en français) peut être considéré comme un mécanisme permettant comme son nom l'indique de déterminer le type des objets durant l'exécution. Dans le cas du C++, le RTTI est associé à l'opérateur typeid et à la classe type_info, donc au header <typeinfo>. Pour plus d'informations concernant le RTTI sur ce langage, veuillez consulter **cet article**.

Avec Qt, les informations de type peuvent être gérées de la même manière qu'en C++ mais bénéficient d'un support plus efficace. Ce support est basé sur les méta-objets et sur QObject, la classe de base de la quasi-totalité des classes de Qt. Il est donc principalement géré par le compilateur de méta-objets (Moc).

Dans le cas de Qt, l'introspection permet, par exemple, de connaître le nom, le type de retour, le type d'accès, etc. d'une méthode, d'un constructeur ou autre. Cet exemple n'a pas été choisi au hasard. En effet, un exemple sera fourni durant cette partie, concernant la récupération des informations des méthodes et des constructeurs d'une classe. Comment utiliser l'introspection ? Cette question est celle à laquelle il me faudra répondre dans cette partie.

Comme vous avez probablement dû le comprendre, chaque classe dérivée de QObject contenant la macro Q_OBJECT possède son propre méta-objet, soit une seule et unique instance de QMetaObject. Cette instance peut être récupérée à l'aide de la fonction metaObject() de QObject. L'instance récupérée est logiquement constante car les méta-objets ne peuvent pas subir directement de traitement intercessif, c'est-à-dire qu'on ne peut pas modifier, par exemple, le nom d'une classe par le biais de son méta-objet. Cette fonction nécessite la présence d'une instance de la classe concernée. Dans le cas où aucune instance ne serait accessible pour utiliser la fonction metaObject(), la variable staticMetaObject peut être utilisée (exemple : QDialog::staticMetaObject). Toutefois, son cas ne sera pas traité ici.

```
const QMetaObject *_metaObject = metaObject();
```

A partir d'une instance de QMetaObject, on peut utiliser un bon nombre de fonctions informatives, telles que les fonctions method(), constructorCount(), access() et bien d'autres.

```
QString name(_metaObject->className());
// name contient le nom de la classe dont le méta-objet est _metaObject.
```

Voici un fragment de code dans lequel on stocke toutes les méthodes et tous les constructeurs de la classe concernée dans un tableau, en indiquant la « signature » de la méthode, son type de retour (bool, QString, etc.), son type de fonction (signal, slot, méthode ou constructeur) et son type d'accès (privé, protégé ou public).

```
QTableWidget *table = new QTableWidget(this);
table->setColumnCount(3);
table->setHorizontalHeaderItem(0, new QTableWidgetItem("Type d'accès"));
table->setHorizontalHeaderItem(1, new QTableWidgetItem("Type de méthode"));
table->setHorizontalHeaderItem(2, new QTableWidgetItem("Type de fonction"));

table->setRowCount(_metaObject->methodCount() + _metaObject->constructorCount());
for(int i = 0; i < _metaObject->methodCount(); i++)
{
 table->setVerticalHeaderItem(i, new QTableWidgetItem(_metaObject->method(i).signature()));
 table->setItem(i, 2, new QTableWidgetItem(_metaObject->method(i).typeName()));

 if(_metaObject->method(i).access() == QMetaMethod::Private) table->setItem(i, 0, new
 QTableWidgetItem("Privé"));
 if(_metaObject->method(i).access() == QMetaMethod::Protected) table->setItem(i, 0, new
 QTableWidgetItem("Protégé"));
 if(_metaObject->method(i).access() == QMetaMethod::Public) table->setItem(i, 0, new
 QTableWidgetItem("Public"));
 if(_metaObject->method(i).methodType() == QMetaMethod::Signal) table->setItem(i, 1, new
 QTableWidgetItem("Signal"));
 if(_metaObject->method(i).methodType() == QMetaMethod::Slot) table->setItem(i, 1, new
 QTableWidgetItem("Slot"));
 if(_metaObject->method(i).methodType() == QMetaMethod::Method) table->setItem(i, 1, new
 QTableWidgetItem("Méthode"));
}
for(int tmp = _metaObject->methodCount(); tmp < _metaObject->methodCount() + _metaObject-
>constructorCount(); tmp++)
{
 int i = tmp - _metaObject->methodCount();
 table->setItem(tmp, 1, new QTableWidgetItem("Constructeur"));
 table->setItem(tmp, 2, new QTableWidgetItem(_metaObject->constructor(i).typeName()));

 table->setVerticalHeaderItem(tmp, new QTableWidgetItem(_metaObject->constructor(i).signature()));
 if(_metaObject->constructor(i).access() == QMetaMethod::Private) table->setItem(tmp, 0, new
 QTableWidgetItem("Privé"));
 if(_metaObject->constructor(i).access() == QMetaMethod::Protected) table->setItem(tmp, 0, new
 QTableWidgetItem("Protégé"));
 if(_metaObject->constructor(i).access() == QMetaMethod::Public)  table->setItem(tmp, 0, new
 QTableWidgetItem("Public"));
}
```

À l'exécution, il est possible de voir de diverses méthodes, selon la classe concernée. On constate d'ailleurs que les signaux et les slots sont aussi stockés dans le méta-objet. C'est avec ce code que l'on peut comprendre une chose : les méta-objets ne disposent que des fonctions qui leur sont fournies par le Moc. Un slot par exemple est ajouté au méta-objet, tandis qu'une méthode publique ne l'est pas sans la présence d'une macro du type de Q_INVOKABLE.

Voici un petit tableau contenant les principales macros utiles pour le recensement de méthodes par le Moc :

Macro	Description rapide
Q_INVOKABLE	Permet au méta-objet d'appeler la méthode.
Q_SIGNAL	Marque la méthode en tant que signal.
Q_SLOT	Marque la méthode en tant que slot.

Il existe bien entendu d'autres macros, comme par exemple Q_SCRIPTABLE.

V-F - Les méta-objets

Les méta-objets constituent le point central de l'article et nous les avons déjà abordés théoriquement dans les parties situées avant celle-ci. La partie précédente envisage une première manière de coder avec les méta-objets dans un but introspectif, ce qui correspond à une petite partie de ce que le système de méta-objet fournit.

Notez que cette partie n'a pas pour but de recenser toutes les macros mises à disposition par le système de méta-objets. Elle présentera les différents dispositifs mis à disposition sous la forme présentation/exemple.

V-F-1 - Les superclasses

On nomme superclasse d'un objet sa classe parente. Les superclasses sont utilisées par QMetaObject pour toutes ses fonctions retournant un offset, par exemple par classInfoOffset().

Il est possible de récupérer le méta-objet de la superclasse d'un objet héritant directement ou non de QObject en appelant la fonction superClass() de QMetaObject. Si cet objet profite d'un héritage multiple, superClass() retournera le méta-objet de la première classe dont il hérite, dont forcément une classe héritant de QObject. Si l'objet n'a pas de superclasse, superClass() retourne 0.

Exemple d'utilisation :

```
const QMetaObject *_metaObject = metaObject()->superClass();
while(_metaObject != 0)
{
 qDebug() << _metaObject->className();
 _metaObject = _metaObject->superClass();
}
```

Étant en haut de la hiérarchie des classes de Qt, QObject sera le dernier nom de classe affiché par qDebug().

V-F-2 - Fonctions d'index

Les fonctions d'index (soit dans Qt 4.6 indexOfConstructor(), indexOfMethod(), indexOfEnumerator(), indexOfProperty(), indexOfSlot(), etc.) sont des fonctions permettant de récupérer l'index d'un constructeur, d'une méthode, etc. à partir de son nom, ou bien -1 si aucun index n'a été trouvé. La documentation précise l'utilité des fonctions d'index en disant que lors de la connexion de signaux à un slot, Qt utilise la fonction indexOfMethod(). Les paramètres passés doivent être normalisés, donc par exemple être mis sous une forme possédant le minimum d'espaces. Il faut donc qu'ils soient passés sous la fonction normalizedSignature().

Exemple d'utilisation dans une classe possédant un slot dont le prototype est myFunction(QString) :

```
const QMetaObject *_metaObject = metaObject();
qDebug() << _metaObject->indexOfMethod(_metaObject->normalizedSignature("myFunction(QString)"));
```

V-F-3 - L'appel de méthodes

Le système de méta-objets permet aussi d'appeler des méthodes par le biais de la fonction statique invokeMethod() et de ses fonctions de convenance. Voici le prototype complet de cette fonction :

```
bool QMetaObject::invokeMethod(QObject *obj,
   const char *member,
   Qt::ConnectionType type,
   QGenericReturnArgument ret,
   QGenericArgument val0 = QGenericArgument(0),
   QGenericArgument val1 = QGenericArgument(),
   //...,
```

```
    QGenericArgument val9 = QGenericArgument());
```

Selon le *type* de connexion, cette fonction appellera la méthode renseignée par *member* (qui doit être recensée par le Moc), de l'objet *obj*, avec les arguments renseignés, et retournera un booléen du fait que l'appel ait réussi ou non.

Les arguments *val0*, *val1*, etc. passés doivent être sous forme de QGenericArgument. Nous utilisons donc une macro nommée Q_ARG, où l'on renseigne le type puis la valeur de l'argument souhaité, qui retourne un objet de ce type.

Exemple d'utilisation dans une classe possédant un slot dont le prototype est myFunction(QString) :

```
const QMetaObject *_metaObject = metaObject();
qDebug() << _metaObject->invokeMethod(this, "myFunction", Qt::DirectConnection, Q_ARG(QString, "Qt is
 good !"));
```

V-F-4 - La création d'instances

Tout comme il peut appeler des méthodes, le système de méta-objets permet aussi de créer des instances. Cela peut être fait à l'aide de la fonction newInstance() de QMetaObject, dont le prototype est le suivant :

```
QObject* QMetaObject::newInstance(QGenericArgument val0 = QGenericArgument(0),
     QGenericArgument val1 = QGenericArgument(),
     ...,
     QGenericArgument val9 = QGenericArgument()) const
```

Cette fonction permet de créer une instance de l'objet auquel le méta-objet appartient. Lors de l'appel de cette fonction, Qt va tenter de trouver un prototype correspondant aux arguments donnés et va l'appeler. Il va de soi que les constructeurs concernés doivent pouvoir être appelés, donc être déclarés avec la macro Q_INVOKABLE pour être recensés par le Moc. Si aucun prototype ne correspond, 0 est retourné.

Comme pour l'appel d'une méthode par le système de méta-objets, *val0*, *val1*, etc. passés doivent être sous forme de QGenericArgument, donc être renseignés avec la macro Q_ARG.

Exemple d'utilisation dans le cadre d'une classe nommée MyWidget, dérivée de QWidget, dont le prototype est Q_INVOKABLE MyWidget(QObject *parent = 0) :

```
const QMetaObject *_metaObject = metaObject();
MyWidget *newObject = qobject_cast<MyWidget *>(_metaObject->newInstance(Q_ARG(QObject *, this)));
if(newObject) newObject->show();
```

V-F-5 - Les propriétés

Si nous ne considérions pas l'aspect des propriétés du côté des méta-objets, il nous aurait été possible de voir les propriétés comme un outil peu important, peu utile. Or, la déclaration d'une propriété à l'aide de la macro Q_PROPERTY engendre son recensement par le Moc, qui va l'ajouter dans le méta-objet.

La fonction property() d'une instance de QObject retourne la valeur de la propriété passée en argument. Toutefois, passer l'index de la propriété dans la fonction property() d'une instance de QMetaObject retournera un QMetaProperty.

Voici un exemple retournant le nom de toutes les propriétés d'un méta-objet :

```
const QMetaObject *_metaObject = metaObject();
for(int i = 0; i < _metaObject->propertyCount(); i++)
qDebug() << _metaObject->property(i).name();
```

QMetaProperty possède de multiples méthodes pouvant, par exemple, retourner un QMetaMethod du signal de notification. Pour plus d'informations, veuillez consulter **cette page**.

VI - Conclusion

Les méta-objets sont donc bien plus complexes et plus utiles qu'on ne pourrait le croire, même s'ils ne correspondent pas à un outil qu'un codeur d'interface graphique utilise fréquemment dans son propre code. Il les utilise, oui, mais ne s'en occupe a priori pas. Il faut noter que cet article présente une partie importante du système, mais n'est pas non plus exhaustif. Si vous souhaitez en apprendre plus sur le sujet, vous pouvez toujours consulter la documentation, par exemple.

VII - Remerciements

J'adresse ici de chaleureux remerciements à **Alp**, **yan**, **dourouc05** et **XHeLL** pour leur aide et leurs judicieux conseils qui m'auront été fort utiles durant tout le long de la rédaction de cet article. Je remercie plus particulièrement **dourouc05** pour son encadrement, pour sa correction orthographique, pour les heures qu'il a passées à relire et pour son soutien.

Un grand merci à **jacques_jean** pour sa relecture orthographique et à **Shulgin** pour ses remarques sur le sujet.