

# Utilisation des Regex avec Qt

par Hugo Stephan ([Site personnel](#))

Date de publication : 12/02/2009

Dernière mise à jour : 30/09/2010

Ce tutoriel a pour but de vous présenter l'utilisation des expressions régulières (Regex), ainsi que la façon de les mettre en œuvre avec Qt.

I - Introduction.....	3
I-A - But de ce tutoriel.....	3
I-B - Mises à jour.....	3
II - Les expressions régulières.....	3
II-A - Exemples.....	5
II-A-1 - Tester la forme d'une adresse mail.....	5
II-A-2 - Tester la forme d'une adresse IP.....	6
II-A-3 - Tester la validité d'une adresse Internet.....	6
III - Mise en œuvre des expressions régulières avec Qt.....	6
III-A - Capturer du texte.....	8
III-B - Les autres fonctions membres de la classe QRegExp.....	9
III-B-1 - La fonction isValid () const.....	9
III-B-2 - La fonction errorString ().....	10
III-B-3 - La fonction exactMatch (const QString & str) const.....	10
III-B-4 - la fonction indexOf (const QString & str, int offset = 0, CaretMode caretMode = CaretAtZero) const.....	10
III-B-5 - la fonction lastIndexOf const QString & str, int offset = -1, CaretMode caretMode = CaretAtZero) const.....	10
III-B-6 - La fonction matchedLength () const.....	10
III-B-7 - La fonction numCaptures () const.....	11
III-C - Accesseurs et mutateurs.....	11
III-D - Tests d'états de la classe QRegExp.....	11
IV - Réalisation d'un interpréteur.....	11
IV-A - Cahier des charges.....	11
IV-A-1 - Présentation de l'interface graphique.....	12
IV-A-2 - Structure du projet.....	13
IV-B - Réalisation du projet.....	14
IV-B-1 - mainwindow.h.....	14
IV-B-2 - mainwindow.cpp.....	14
IV-B-3 - main.cpp.....	17
C - Idées d'améliorations.....	18
V - Conclusion.....	18

## I - Introduction

Les expressions régulières sont des chaînes de caractères qui permettent de décrire un motif ("pattern" en anglais), elles sont des outils très puissants, elles permettent de réaliser une foule de choses, par exemple un analyseur lexical. Les expressions régulières sont souvent considérées comme ardues et incompréhensibles car elles sont assez difficiles à aborder, mais après avoir lu ce tutoriel vous serez apte à utiliser les Regex sans problèmes dans vos programmes.

### I-A - But de ce tutoriel

Ce tutoriel a pour but de vous présenter les expressions régulières avec Qt. Nous allons d'abord nous intéresser à la syntaxe des expressions régulières en elles-mêmes, indépendamment des langages, puis nous verrons comment les mettre en œuvre avec Qt.

### I-B - Mises à jour

- **vendredi 24 avril 2009** : ajout des commandes FILE\_append et FILE\_rename dans le projet d'interpréteur (modifications de différents codes dans la section IV-B-2) ;
- **vendredi 24 avril 2009** : ajout d'une fonction de génération des fichiers à inclure dans le projet d'interpréteur (modifications de l'interface graphique, section IV-A-1 et de différents code dans la section IV-B-2) ;
- **jeudi 16 mai 2009** : corrections suite aux suggestions de lrmadDen ;
- **mardi 19 mai 2009** : corrections orthographiques apportées par dourouc05 ;
- **jeudi 30 septembre 2010** : corrections orthographiques de ram-0000 et jacques\_jean apportées par johnlamericaïn.

## II - Les expressions régulières

La syntaxe des expressions régulières peut paraître peu abordable dans un premier temps, mais après avoir compris les principes que je vais exposer là, vous devriez très bien comprendre la plupart des expressions régulières.

La plupart des caractères n'ont aucune signification particulière et peuvent être utilisés tels quels, sauf quelques-uns qui devront être échappés :

```
r
```

Acceptera toutes les chaînes de caractères possibles avec la lettre r.

```
ro
```

Acceptera toutes les chaînes contenant "ro", comme "roi", "rose", "arthrose", mais pas "rio".

Pour laisser le choix entre plusieurs caractères, il faut les mettre entre crochets, ainsi :

```
[ft]rotte
```

Acceptera "frotte" ou "trotte", mais pas "carotte".

Pour spécifier beaucoup de caractères (qui se suivent), vous pouvez indiquer le début et la fin séparés par un tiret, par exemple :

```
[a-z]
```

Acceptera tous les caractères de l'alphabet (en minuscule). Ainsi pour nous faciliter la tâche, il existe plusieurs classes abrégées :

Classe	Signification
<code>\a</code>	Le caractère ASCII (0x07), (le beep).
<code>\f</code>	Le caractère ASCII (0x0C),
<code>\n</code>	Le caractère ASCII (0x0A), (le retour à la ligne).
<code>\r</code>	Le caractère ASCII (0x0D), (le retour chariot).
<code>\t</code>	Le caractère ASCII (0x09), (la tabulation horizontale).
<code>\v</code>	Le caractère ASCII (0x0B), (la tabulation verticale).
<code>\xhhh</code>	Le caractère correspondant à <i>hhh</i> exprimé en hexadécimal.
<code>\0ooo</code>	Le caractère correspondant à <i>ooo</i> exprimé en octal.
<code>.</code>	N'importe quel caractère.
<code>\d</code>	N'importe quel chiffre.
<code>\D</code>	N'importe quel caractère n'étant pas un chiffre.
<code>\s</code>	Un espace blanc.
<code>\S</code>	N'importe quel caractère n'étant pas un espace blanc.
<code>\w</code>	Un caractère alphabétique ( <code>QChar::isLetterOrNumber()</code> , <code>QChar::isMark()</code> , ou bien <code>'_'</code> ).
<code>\W</code>	Un caractère non alphabétique (inverse de <code>\w</code> ).

Il existe également divers opérateurs :

Opérateurs	Signification
<code>?</code>	L'élément est répété, au maximum une fois.
<code>*</code>	L'élément est présent, au moins zéro fois.
<code>+</code>	L'élément est présent, au moins une fois.
<code>{ n }</code>	L'élément est présent <i>n</i> fois.
<code>{ n, }</code>	L'élément est présent au moins <i>n</i> fois.
<code>{ n, m }</code>	L'élément est présent entre <i>n</i> et <i>m</i> (compris) fois.
<code>{ ,m }</code>	L'élément est présent au maximum <i>m</i> fois.

Le caractère `^` permet d'inverser un ensemble, il peut aussi être utilisé pour préciser que la chaîne doit commencer par une expression (tout comme le caractère `$` pour la fin de la chaîne), le caractère `|` permet de laisser le choix entre deux expressions :

```
^(debut|begin)
```

Acceptera les chaînes commençant par "debut" ou "begin".

Voici la liste des méta caractères à échapper :

Caractères	Échappés en
#	\#
!	\\!
^	\\^
\$	\\\$
(	\\(
)	\\)
[	\\[
]	\\]
{	\\{
}	\\}
?	\\?
+	\\+
*	\\*
.	\\.
\\	\\\\

## II-A - Exemples

Dans cette partie, nous allons créer divers Regex pour vous familiariser avec leur utilisation.

### II-A-1 - Tester la forme d'une adresse mail

Comment tester la validité d'une adresse mail ? Notez que cet exemple n'est pas adapté à un contexte réel et n'est là que pour illustrer de manière théorique l'utilisation des Regex.

Une adresse mail doit être de la forme : utilisateur@fai.truc. Voici la Regex :

```
^[\\w|\\.]+@\\[\\w]+\\. [\\w]{2,4}$
```

Cela peut certes sembler assez incompréhensible, mais il faut prendre le temps de bien décomposer les étapes.

```
^[\\w|\\.]+
```

Déjà ici on analyse la première partie de l'adresse, celle-ci peut être composée de n'importe quel caractère (\\w) ou d'un point (\\w n'inclut pas le point), le tout répété au moins une fois.

```
@ [\\w]+
```

Ici on précise qu'il doit y avoir un @ suivi de n'importe quel caractère alphabétique (\\w) répété au moins une fois.

```
\\. [\\w]{2, 4}
```

Dans cette dernière partie, on précise qu'il doit y avoir un caractère "." (le point est échappé, pour ne pas signifier n'importe quel caractère), suivi de deux à quatre caractères alphabétiques (ce qui couvre .fr, .com, .gouv, etc.).

#### Cette Regex acceptera les chaînes :

- nom@fai.com
- agent.secret@truc.gouv
- b@c.fr
- etc...

## Mais elle n'acceptera pas les chaînes :

- nom.nom@fai.truci
- @fai.com
- machin.truc.com

## II-A-2 - Tester la forme d'une adresse IP.

Comment tester la validité d'une adresse IP ? Notez que cet exemple n'est pas adapté à un contexte réel et n'est là que pour illustrer de manière théorique l'utilisation des Regex.

Une adresse IP doit être de la forme xxx.xxx.xxx.xxx, voici une Regex possible :

```
^[\d]{2,3}\. [\d]{2,3}\. [\d]{2,3}\. [\d]{2,3}$
```

Cette Regex est déjà plus simple.

```
^[\d]{2, 3}\.
```

Première partie, le ^ utilisé pour signaler le début de la chaîne puis un bloc (qui sera répété) qui spécifie qu'il doit y avoir une suite de deux ou trois chiffres quelconques, le tout doit être suivi d'un point.

Ce bloc est répété (sauf pour le point) pour le reste de l'adresse IP, étant donné qu'elle est composée de blocs de deux ou trois chiffres.

Et enfin, la Regex est terminée par le symbole \$, pour qu'aucun autre caractère après ne soit accepté.

## II-A-3 - Tester la validité d'une adresse Internet

Comment tester la validité d'une adresse Internet ? Notez que cet exemple n'est pas adapté à un contexte réel et n'est là que pour illustrer de manière théorique l'utilisation des Regex.

Une adresse Internet doit être de la forme www.xxx.xxxx. Voici une des Regex possibles :

```
^[w]{3}\. [\w]+\\. [\w]{2,4}$
```

Ici nous avons :

```
^[w]{3}
```

qui signale trois w,

```
\. [\w]+\.
```

suivis d'un point puis de n'importe quel caractère répété au moins une fois et d'un autre point,

```
[\w]{2,4}$
```


suivis d'une extension (.fr, .com, .gouv) d'au moins deux caractères et au maximum de quatre caractères.

## III - Mise en œuvre des expressions régulières avec Qt

Une expression régulière avec Qt est représentée par une instance de la classe QRegExp.

### QRegExp dispose de trois constructeurs

- Un constructeur sans arguments qui construit une Regex vide.

- Un constructeur qui prend un QString représentant la Regex en argument.
- Un constructeur qui prend trois arguments : la Regex, un paramètre pour savoir si la Regex sera sensible à la casse, et le type de Regex utilisé (QRegExp::RegExp pour les Regex PERL, QRegExp::RegExp2 (ce qui revient à utiliser setMinimal (true)) de manière à réduire le contenu de la chaîne correspondant à la Regex au motif strict, en enlevant le texte entre deux répétitions du même motif, mais ce système est plus gourmand en mémoire, QRegExp::Wildcard pour un système simple voir  [cette page](#), QRegExp::FixedString revient à échapper tous les caractères spéciaux).

Pour reprendre notre exemple d'adresses mails, voici la manière de créer la Regex :

```
QRegExp mailExp ("^[\\w|\\.]+@[\\w]+\\. [\\w]{2,4}$");
```

Je vous rappelle que le code est entre parenthèses et qu'il est donc nécessaire d'échapper les slashes pour ne pas qu'ils soient considérés comme des séquences d'échappement.

Ensuite, pour vérifier si une chaîne de caractères correspond à une Regex, il nous faut utiliser la fonction membre *contains()* de la classe Qt :

```
bool QString::contains (const QRegExp & exp) const
```

Ainsi voici un exemple basique de programme permettant de récupérer l'adresse mail de l'utilisateur et de vérifier sa forme grâce à la Regex que nous avons créée.

#### regexqt.h

```
#ifndef REGEXQT_H
#define REGEXQT_H

#include <QDialog>

class QLabel;
class QLineEdit;
class QPushButton;

class regexqt : public QDialog
{
    Q_OBJECT

public:
    regexqt ();
    ~regexqt ();

public slots:
    void OnConfirm (QRegExp mailExp);

private:
    QLabel* label;
    QLineEdit* mailEntry;
    QPushButton* confirmButton;
    QRegExp mailExp;
};

#endif
```

Définition basique d'une classe Qt, héritant de QDialog, avec la macro Q\_OBJECT.

#### regexqt.cpp

```
#include <QtGui>
#include "regexqt.h"

regexqt::regexqt ()
{
    /* définition du titre de la fenêtre */
    setWindowTitle ("Validation");
```

## regexqt.cpp

```
/* création des contrôles */
label = new QLabel ("Entrez votre chaine de caracteres :");
Entry = new QLineEdit;
confirmButton = new QPushButton ("&Confirmer");
label->setBuddy (Entry);

/* connexion du signal clicked pour le bouton confirmer */
connect (confirmButton, SIGNAL (clicked ()), this, SLOT (OnConfirm ()));

/* création du layout */
QHBoxLayout* centralLayout = new QHBoxLayout;
centralLayout->addWidget (label);
centralLayout->addWidget (Entry);
centralLayout->addWidget (confirmButton);
setLayout (centralLayout);

/* création de la Regex */
QRegExp Exp ("^[\\w|\\.]+@[\\w]+\\. [\\w]{2,4}$");
mailExp = Exp ;
}

void regexqt::OnConfirm ()
{
    QString EntryText = Entry->text ();
    if (EntryText.contains (mailExp))
        QMessageBox::information (this, "Validation", "Valide !");
    else
        QMessageBox::critical (this, "Validation", EntryText + " n'est pas valide !");
    return;
}

regexqt::~regexqt ()
{
}
}
```

On crée l'interface et la Regex, puis l'on connecte le signal clicked pour le bouton confirmButton à la fonction regexqt::OnConfirm () qui vérifie la forme de l'adresse mail grâce à la Regex.

## main.cpp

```
#include <QApplication>
#include "regexqt.h"

int main (int argc, char *argv[])
{
    QApplication app (argc, argv);
    regexqt* regexUi = new regexqt;
    regexUi->show ();
    return app.exec ();
}
```

Création d'un objet de la classe regexqt et affichage de cet objet.

## III-A - Capturer du texte

Une autre partie importante des Regex est la faculté à capturer du texte. Pour chaque couple de parenthèses Regex (ici Qt) génère une variable interne contenant le texte, par exemple :

```
^(debut|begin)$
```

Après avoir vérifié que la chaîne de caractères y correspond, comment feriez-vous pour savoir s'il s'agit finalement de "debut" ou de "begin" ? C'est ce principe que l'on nomme généralement capturer du texte, la théorie est simple, il nous faut accéder à la variable créée en interne qui contient le contenu final des parenthèses, pour cela nous allons faire appel à la fonction membre cap de QRegExp, voici le prototype :



```
QString QRegExp::cap (int n = 0)
```

Cette fonction prend en argument le numéro du texte capturé, pour nous il s'agit du numéro 1 étant donné que c'est le seul couple de parenthèse que nous avons inséré dans la Regex, si vous envoyez 0 en argument à la fonction, elle vous retournera toute la chaîne de caractères (s'il a été validé).

Ainsi voici le code de la dernière fois un petit peu adapté (seul le fichier regexqt.cpp a changé) :

```
#include <QtGui>
#include "regexqt.h"

regexqt::regexqt ()
{
    /* définition du titre de la fenêtre */
    setWindowTitle ("Validation");

    /* création des contrôles */
    label = new QLabel ("Entrez votre chaine de caracteres :");
    Entry = new QLineEdit;
    confirmButton = new QPushButton ("&Confirmer");
    label->setBuddy (Entry);

    /* connexion du signal clicked pour le bouton confirmer */
    connect (confirmButton, SIGNAL (clicked ()), this, SLOT (OnConfirm ()));

    /* création du layout */
    QHBoxLayout* centralLayout = new QHBoxLayout;
    centralLayout->addWidget (label);
    centralLayout->addWidget (Entry);
    centralLayout->addWidget (confirmButton);
    setLayout (centralLayout);

    /* création de la Regex */
    QRegExp exp ("^(debut|begin)$");
    Exp = exp ;
}

void regexqt::OnConfirm ()
{
    QString EntryText = Entry->text ();
    if (EntryText.contains (Exp))
        QMessageBox::information (this, "Validation", "Valide, vous avez choisi : " + Exp.cap (1));
    else
        QMessageBox::critical (this, "Validation", EntryText + " n'est pas valide !");
    return;
}

regexqt::~regexqt ()
{
    delete Exp;
}
```

Vous disposez également de la fonction membre :

```
QStringList QRegExp::capturedTexts ()
```

qui renvoie tout le texte qui a été capturé sous forme d'une QStringList.

## III-B - Les autres fonctions membres de la classe QRegExp

### III-B-1 - La fonction isValid () const

Cette fonction dont voici le prototype :

```
bool QRegExp::isValid () const
```

renvoie false si la Regex donnée en argument n'est pas valide (une Regex non valide n'accepte aucune chaîne de caractères), true sinon.

### III-B-2 - La fonction errorString ()

Cette fonction :

```
QString QRegExp::errorString ()
```

retourne un texte explicatif d'une éventuelle erreur qui a pu se produire. S'il n'y a pas d'erreur, la fonction retourne "no error occurred".

### III-B-3 - La fonction exactMatch (const QString & str) const

La fonction :

```
bool QRegExp::exactMatch (const QString & str) const
```

renvoie true si la chaîne de caractères passée en argument est exactement équivalente à la Regex, par exemple, pour la Regex "a", la fonction renvoie true uniquement si la chaîne de caractères correspond à "a" exactement, elle renvoie false pour "aa", "ab", etc. Cela revient à rajouter "^" en début de Regex et "\$" en fin de Regex.

### III-B-4 - la fonction indexIn (const QString & str, int offset = 0, CaretMode caretMode = CaretAtZero) const

La fonction indexIn dont voici le prototype :

```
int QRegExp::indexIn (const QString & str, int offset = 0, CaretMode caretMode = CaretAtZero) const
```

cherche à valider une partie de *str* à partir de la position *offset*. Renvoie la position.

### III-B-5 - la fonction lastIndexIn const QString & str, int offset = -1, CaretMode caretMode = CaretAtZero) const

La fonction

```
int QRegExp::lastIndexIn (const QString & str, int offset = -1, CaretMode caretMode = CaretAtZero) const
```

ressemble beaucoup à la fonction indexIn à ceci près qu'elle effectue la recherche dans l'autre sens, en partant de la position offset qui par défaut vaut -1 (dernier caractère de la chaîne). Cette fonction renvoie la position de la première occurrence trouvée.



*Attention, rechercher à l'envers est plus lent que rechercher à l'endroit, ainsi si vous avez le choix, préférez la recherche à l'endroit.*

### III-B-6 - La fonction matchedLength () const

Cette fonction :

```
int QRegExp::matchedLength () const
```

renvoie la taille de la chaîne capturée, -1 si aucune chaîne n'a été trouvée.

### III-B-7 - La fonction numCaptures () const

Cette fonction dont voici le prototype :

```
int QRegExp::numCaptures () const
```

retourne le nombre d'expressions capturées.

### III-C - Accesseurs et mutateurs

La classe QRegExp respecte le principe d'accesseur/mutateur utilisé dans toutes les classes Qt, ce principe est simple : pour accéder à une variable appelée var, vous avez la fonction T class::var () et la fonction void class::setVar (T var) pour en définir la valeur.

#### Liste des accesseurs et mutateurs de la classe QRegExp

- Les fonctions Qt::CaseSensitivity QRegExp::caseSensitivity () const et void QRegExp::setCaseSensitivity (Qt::CaseSensitivity c) permettent respectivement d'accéder et de modifier la sensibilité à la casse de la Regex.
- Les fonctions QString QRegExp::pattern () const et void QRegExp::setPattern (const QString & pattern ) permettent respectivement d'obtenir la chaîne de la Regex et de la définir.
- Les fonctions PatternSyntax QRegExp::patternSyntax () const et void QRegExp::setPatternSyntax (PatternSyntax syntax) permettent respectivement d'obtenir le type de la Regex et de le définir.
- Le mutateur void QRegExp::setMinimal (bool minimal) permet de définir si la Regex est en mode "gourmand" (true) ou non (false), l'accesseur de *minimal* n'est plus utilisé avec Qt4 (il est remplacé par le test d'état *isMinimal*).

### III-D - Tests d'états de la classe QRegExp

La classe QRegExp possède également des tests d'états, qui peuvent être considérés comme un cas spécial des accesseurs pour les variables de type *bool*. Le test d'état de la variable *var* sera appelé *isVar* ().

#### Liste des tests d'états de la classe QRegExp

- Le test d'état bool QRegExp::isEmpty () const renvoie true si la Regex est vide, false sinon.
- Le test d'état bool QRegExp::isMinimal () const renvoie true si la Regex est en mode "gourmand", false sinon.
- Le test d'état bool QRegExp::isValid () const renvoie true si la Regex est syntaxiquement correcte, false sinon (dans ce cas *errorString* () renvoie la chaîne de l'erreur).

### IV - Réalisation d'un interpréteur

Les Regex se prêtent bien à ce genre de projet, aussi ici nous allons mettre nos connaissances en pratique pour réaliser un interpréteur pour créer un outil d'aide à la programmation.

#### IV-A - Cahier des charges

Cet interpréteur se présentera sous forme d'une boîte de saisie, où l'utilisateur pourra entrer différentes commandes qui généreront des "morceaux" de code C++, de manière à simplifier le travail du programmeur. Commençons par établir une liste des commandes reconnues :

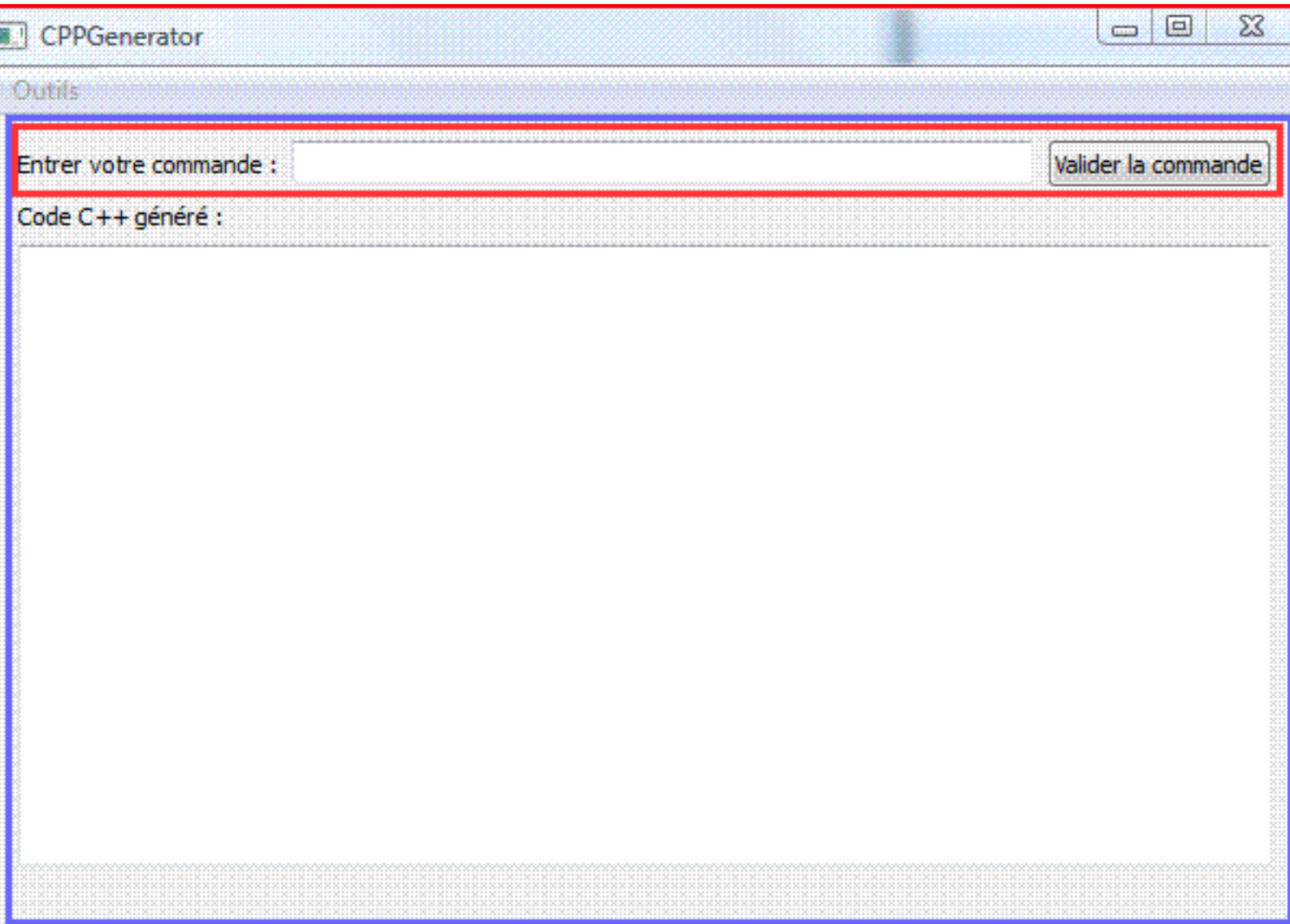
- un ensemble de commandes destinées à la gestion des fichiers, sous la forme FILE\_xxx ;
- un ensemble de commandes destinées à la gestion des chaînes de caractères, sous la forme STRING\_xxx.

## IV-A-1 - Présentation de l'interface graphique

Pour ne pas alourdir le tutoriel en codage de GUI, je vous propose d'utiliser le QtDesigner pour créer notre interface graphique (mais rien ne vous empêche de la coder à la main si vous n'êtes pas à l'aise avec le Designer), celle-ci sera relativement simple :

- deux labels d'information pour indiquer à l'utilisateur où entrer sa commande et où apparaît le code C++ généré, des QLabel ;
- une entrée de texte où l'utilisateur pourra entrer sa commande, nous utiliserons un QLineEdit ;
- une case à cocher si l'utilisateur souhaite que le programme génère également les includes nécessaires à la correcte compilation du code donné ;
- une sortie de texte où l'utilisateur pourra récupérer le code C++ généré, un QTextEdit ;
- une barre de menus, permettant de lancer la génération du code et d'afficher une fenêtre d'aide.

À l'aide du QtDesigner, l'interface est vite prête, vous devriez obtenir quelque chose comme cela :



### *L'interface de notre programme*

En rouge vous pouvez voir un gestionnaire de disposition horizontale qui sert à contenir le label et l'entrée de texte pour que l'utilisateur puisse saisir sa commande, et en bleu vous pouvez voir le layout principal qui est un gestionnaire de disposition vertical. Pour la bonne suite de la construction du projet, il est nécessaire de donner des noms intuitifs aux widgets de notre fenêtre, voici la liste de ceux que j'ai donnés à mes widgets, rien ne vous empêche d'en utiliser d'autres mais il vous faudra alors adapter le code :

- l'entrée de texte où l'utilisateur saisit sa commande s'appelle `commandEntry` ;
- le bouton de validation s'appelle `confirmButton` ;
- la case à cocher d'option s'appelle `includeCheckBox` ;
- la zone de texte éditable où apparaîtra le code généré s'appelle `textEdit` ;
- l'action pour valider la commande s'appelle `actionStartCommand`, celle pour l'aide `actionHelp` et celle pour le à propos `actionAboutQt` ;
- les noms des autres contrôles n'ont qu'une importance mineure étant donné que nous n'aurons pas besoin de les utiliser dans le code que nous écrirons.

## IV-A-2 - Structure du projet

Là encore je vais essayer de faire dans la simplicité, notre projet se composera donc de quatre fichiers :

- 1 main.cpp, fichier "principal" du programme ;
- 2 mainwindow.h, fichier contenant la déclaration de la classe mainWindow ;
- 3 mainwindow.cpp, fichier contenant l'implémentation de la classe mainWindow ;
- 4 mainwindow.ui, fichier généré par le Designer contenant l'interface graphique de notre programme.

## IV-B - Réalisation du projet

### IV-B-1 - mainwindow.h

À partir de ce point je suppose que vous avez au préalable correctement créé l'interface du programme à l'aide du designer. Je vous propose de commencer par coder la déclaration de notre classe MainWindow, ici rien de bien compliqué :

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QWidget ;
class QString ;

namespace Ui
{
    class MainWindowClass ;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public :
    MainWindow (QWidget *parent = 0) ;
    ~MainWindow () ;
    bool isValidCommand () ;
    void toGenerateCode () ;
    bool toParseCommand () ;
    bool toParseFileCommand () ;
    bool toParseStringCommand () ;

public slots :
    void onEnterCommand () ;
    void Help () ;

private :
    Ui::MainWindowClass *ui ;
    QString code ;
};

#endif // MAINWINDOW_H
```

Vous noterez l'utilisation de l'interface graphique créée par le Designer qui, je suppose, est générée dans le fichier ui\_mainwindow.h.

### IV-B-2 - mainwindow.cpp

Commençons en douceur par les includes en anticipant sur les classes dont nous allons nous servir :

```
/* Inclusion des classes Qt */
#include <QRegExp>
#include <QMessageBox>
#include <QWidget>
#include <QMainWindow>
```

```
/* Inclusion de nos fichiers */
#include "mainwindow.h"
#include "ui_mainwindow.h"
```

Passons au constructeur de notre classe MainWindow, là non plus rien de bien compliqué, on génère l'interface graphique à l'aide de la fonction automatiquement codée par le Designer puis on connecte les signaux de nos trois actions et du bouton pour valider la commande :

```
MainWindow::MainWindow (QWidget *parent)
: QMainWindow (parent), ui (new Ui::MainWindowClass)
{
    /* Construction de l'interface graphique */
    ui->setupUi(this) ;

    /* Connexion des signaux */
    connect (ui->confirmButton, SIGNAL (clicked ()), this, SLOT (onEnterCommand ())) ;
    connect (ui->actionStartCommand, SIGNAL (triggered ()), this, SLOT (onEnterCommand ())) ;
    connect (ui->actionAboutQt, SIGNAL (triggered ()), qApp, SLOT (aboutQt ())) ;
    connect (ui->actionHelp, SIGNAL (triggered ()), this, SLOT (Help ())) ;

    /* Mise en forme du texte sous forme de code */
    code = "<code>" ;
}
```

Codons à présent le slot qui sera appelé lorsque l'utilisateur cliquera sur le bouton, on vérifie si la commande est valide puis on appelle la fonction toGenerateCode () :

```
void MainWindow::onEnterCommand ()
{
    if (isValidCommand ())
        toGenerateCode () ;
    else
        return ;
}
```

La fonction isValidCommand :

```
bool MainWindow::isValidCommand ()
{
    if (ui->commandEntry->text () == "")
    {
        QMessageBox::critical (this, "Erreur", "Aucune commande n'a été entrée") ;
        return false ;
    }
    else
    {
        return toParseCommand () ;
    }
    return true ;
}
```

Ici c'est encore simple, il suffit de vérifier si l'utilisateur a entré une commande puis d'appeler la fonction toParseCommand (), qui regarde si la commande est de type FILE\_xxx ou STRING\_xxx, si la commande ne correspond à aucune de ces deux possibilités, on affiche un message d'erreur et on retourne false :

```
bool MainWindow::toParseCommand ()
{
    QRegExp FILECommand ("FILE_.+") ;
    QRegExp STRINGCommand ("STRING_.+") ;
    if (ui->commandEntry->text ().contains (FILECommand))
    {
        return toParseFileCommand () ;
    }
    else if (ui->commandEntry->text ().contains (STRINGCommand))
    {

```

```

        return toParseStringCommand () ;
    }
    else
    {
        QMessageBox::critical (this, "Erreur", "Commande invalide") ;
        return false ;
    }
}

```

Je vais coder avec vous la fonction toParseFileCommand, puis la fonction toParseMathCommand et je vous laisserais coder toParseStringCommand, commençons par établir une liste précise des commandes acceptées.

- FILE\_create:nom\_du\_fichier, pour créer un nouveau fichier, nous utiliserons la Regex **FILE\_create:(.+)**.
- FILE\_write:nom\_du\_fichier:texte\_à\_écrire, pour écrire du texte dans un fichier, nous utiliserons la Regex **FILE\_write:(.+):(.+)**.
- FILE\_append:nom\_du\_fichier:texte\_a\_ajouter, pour ajouter du texte à la fin d'un fichier, nous utiliserons la Regex **FILE\_append:(.+):(.+)**.
- FILE\_rename:ancien\_nom:nouveau\_nom, pour renommer un fichier, nous utiliserons la Regex **FILE\_rename:(.+):(.+)**.

```

bool MainWindow::toParseFileCommand ()
{
    QRegExp FILECreateCommand ("FILE_create:(.+)") ;
    QRegExp FILEWriteCommand ("FILE_write:(.+):(.+)") ;
    QRegExp FILEAppendCommand ("FILE_append:(.+):(.+)") ;
    QRegExp FILERenameCommand ("FILE_rename:(.+):(.+)") ;
    if (ui->commandEntry->text ().contains (FILECreateCommand))
    {
        QString filename = FILECreateCommand.cap (1) ;
        filename.replace ("\\", "\\\\") ;
        if (ui->includeCheckBox->isChecked ())
        {
            code += "#include <iostream><br>#include <fstream><br><br>" ;
        }
        code += "ofstream outStream (\"" + filename + "\", ios::out) ;" ;
        return true ;
    }
    else if (ui->commandEntry->text ().contains (FILEWriteCommand))
    {
        QString filename = FILEWriteCommand.cap (1) ;
        filename.replace ("\\", "\\\\") ;
        if (ui->includeCheckBox->isChecked ())
        {
            code += "#include <iostream><br>#include <fstream><br><br>" ;
        }
        code += "ofstream outStream (\"" + filename + "\", ios::out) ;<br>outStream <&&\"" +
        FILEWriteCommand.cap (2) + "\" ;" ;
        return true ;
    }
    else if (ui->commandEntry->text ().contains (FILEAppendCommand))
    {
        QString filename = FILEAppendCommand.cap (1) ;
        filename.replace ("\\", "\\\\") ;
        if (ui->includeCheckBox->isChecked ())
        {
            code += "#include <iostream><br>#include <fstream><br><br>" ;
        }
        code += "ofstream outStream (\"" + filename + "\", ios::out | ios::app) ;<br>outStream <&&\"" +
        FILEAppendCommand.cap (2) + "\" ;" ;
        return true ;
    }
    else if (ui->commandEntry->text ().contains (FILERenameCommand))
    {
        QString filename = FILERenameCommand.cap (1) ;
        filename.replace ("\\", "\\\\") ;
        QString filename_new = FILERenameCommand.cap (2) ;
        filename_new.replace ("\\", "\\\\") ;
        if (ui->includeCheckBox->isChecked ())

```



```

    {
        code += "#include <stdio.h><br><br>" ;
    }
    code += "if (rename (\"" + filename + "\", \"" + filename_new + "\") == -1) <br><br>
\\tcerr <&& \\\"Erreur lors du changement de nom du fichier \\\" ;<br>}" ;
    return true ;
}
else
{
    QMessageBox::critical (this, "Erreur", "Commande FILE_xxx invalide") ;
    return false ;
}
return false ;
}

```

Comme promis je vous laisse faire pour toParseString (), à présent codons le slot Help :

```

void MainWindow::Help ()
{
    QMessageBox::about
(this, "Aide", "Le programme CPPGenerator est un utilitaire d'aide à la conception de programme C+
+, il reconnaît 2 commandes :\"
        \"<br>FILE_create:nom_du_fichier pour créer un nouveau fichier\"
        \"<br>FILE_write:nom_du_fichier:text_a_écrire pour écrire du texte dans un fichier.\"

    \"<br>FILE_append:nom_du_fichier:text_a_ajouter pour ajouter du texte à la fin d'un fichier.\"
    \"<br>FILE_rename:ancien_nom:nouveau_nom pour renommer un fichier.\") ;
}

```

C'est sûrement la fonction la plus simple car elle se contente d'afficher une boîte d'informations avec les différentes commandes supportées. Ici, je n'ai mis que celle que je vous ai donnée, je vous laisse modifier le texte pour ajouter les vôtres.

La fonction toGenerateCode, elle aussi est relativement simple, maintenant que tout le travail est fait :

```

void MainWindow::toGenerateCode ()
{
    code += "</code>" ;
    ui->textEdit->append ("<u>Code généré pour la commande <b>" + ui->commandEntry->text () + "</b> :</
u><br>" + code + "<br>)" ;
    code = "<code>" ;
    ui->commandEntry->setText ("") ;
}

```

Le destructeur à présent, car il ne faut pas oublier de libérer la mémoire utilisée par l'interface :

```

MainWindow::~MainWindow ()
{
    delete ui ;
}

```

## IV-B-3 - main.cpp

Là encore, il n'y a rien à coder (ou presque) :

```

#include <QApplication>
#include "mainwindow.h"

int main (int argc, char *argv[])
{
    QApplication app (argc, argv) ;
    MainWindow w ;
    w.show () ;
    return app.exec () ;
}



```

```
}
```

## C - Idées d'améliorations

Rien ne vous empêche de multiplier les commandes, de manière à obtenir un outil très puissant en rajoutant des commandes pour la gestion GUI avec Qt par exemple, ou bien en générant le code complet du main.cpp correspondant à la requête et à le compiler pour créer un véritable petit langage.

## V - Conclusion

Si vous désirez d'avantages d'informations, je ne peux que vous recommander de consulter la documentation Qt en ligne de Qt Software :  <http://doc.trolltech.com/> et plus particulièrement la page consacrée aux **QRegExp**. Vous pouvez également visiter ce site très intéressant consacré aux expressions régulières,  <http://www.regular-expressions.info/>. À présent vous devriez être apte à utiliser les expressions régulières dans vos programmes Qt sans difficulté. Et je vous propose de conclure avec cette citation :

*Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. -- James Zawinski*

Certaines personnes lorsqu'elles sont confrontées à un problème pensent : "Je sais ! Je vais utiliser les expressions régulières." À présent elles doivent faire face à deux problèmes...