TP Qt : César

© 2012 tv <tvaira@free.fr> - v.1.0 - le 6 mars 2013

Sommaire

Manipulations **2**

Objectifs 2

Mise en situation 2

Travail demandé **3**

Itération 1 3

Itération 2 8

Il est fortement conseillé d'utiliser la documentation Qt de référence en français (http://qt.developpez.com/doc/) ou en anglais (http://qt-project.org/doc/).

Manipulations

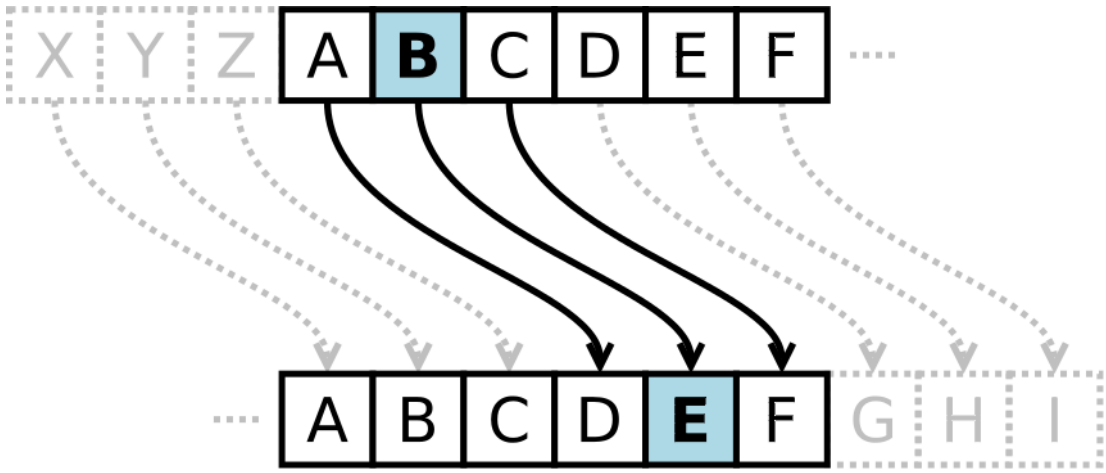
Objectifs

Les objectifs de ce TP sont :

- utilisation de quelques widgets et des classes spécialisées QString et QChar
- mise en oeuvre du mécanisme signal/slot

Mise en situation

Dans ce TP, on vous propose de développer une application graphique en Qt en utilisant des widgets et basée sur un TP C++ d'Éric Rémy. On va donc réaliser un programme qui crypte ou décrypte un message secret m avec une clé secrète k numérique suivant une méthode simple le chiffre de César (fr.wikipedia.org/wiki/Chiffre_de_Cesar).



On va développer l'application en deux itérations.

Remarque : un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un système partiel exécutable, testé et intégré (mais incomplet).

Le principe est simple : remplacer chaque lettre du message d'origine par la lettre qui se trouve k places plus loin (quand k est positif) dans l'alphabet. Par exemple, la lettre 'b' est remplacée par 'd' si k=2. La seule complication tient au fait que quand on arrive à la fin de la liste, ce qui dépasse se retrouve de l'autre côté (toujours avec k=2) :

- Clair → ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
- Crypté → CDEFGHIJKLMNOPQRSTUVWXYZAB cdefghijklmnopqrstuvwxyzab

Sur le même principe, si k est négatif, le décalage se produit dans l'autre sens.

 © 2012 tv <tvaira@free.fr>

L'opération peut être effectuée très simplement grâce à la suite d'opérations suivante :

```
si k<0 alors k=(k modulo 26) + 26. fin si.
soit c le code ASCII d'une les lettres du message, il faut la remplacer par (c - 'a' + k)
   modulo 26 + 'a' quand 'a' est le code ASCII de "a" (remplacez par 'A' pour coder les
   majuscules).
```

En effet, si k<0 alors k modulo 26 donne une valeur équivalente vis-à-vis du codage dans l'intervale [-25 ; 0], et si on lui rajoute 26, on arrive dans l'intervale [1 ; 26] c'est-à-dire qu'on a transformé notre décalage négatif en un décalage positif qui a le même effet. Par exemple, si le décalage était -28 (qui transforme 'C' en 'A'), -28 modulo 26 donne -2 (qui transforme lui aussi 'A' en 'A'), et -28 module 26 + 26 donne 24 (qui lui aussi transforme 'C' en 'A'), La deuxième étape suppose que k est positif (ce qui est toujours le cas grâce à la première ligne). On soustrait du code ASCII de c (par exemple, 98 pour 'b') la valeur du code ASCII de 'a' (97) afin d'obtenir la position de la lettre dans l'alphabet (puisqu'on commence à compter à 0, pour 'b', cela donne 1). Ensuite, on ajoute k pour décaler, et grâce au modulo, on reste dans [0 ; 25]. Il ne reste plus qu'à se remettre dans les lettres minuscules en ajoutant le code ASCII de 'a'.

Travail demandé

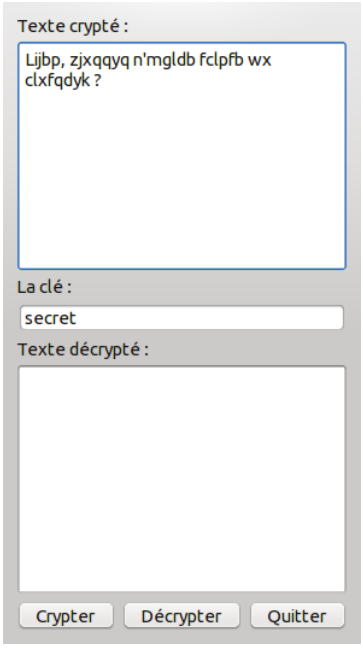
Itération 1

Dans cette première itération, on s'attachera à créer sa propre boîte de dialogue en créant une nouvelle classe **MyDialog** qui héritera de la classe **QDialog**. Une instance de cette classe représentera la fenêtre de l'application :

```
#include <QApplication>

#include "mydialog.h"

int main( int argc, char **argv )
{
   QApplication a( argc, argv );

   MyDialog w;
   w.show();

   return a.exec();
}
```

Code 2 – main.cpp

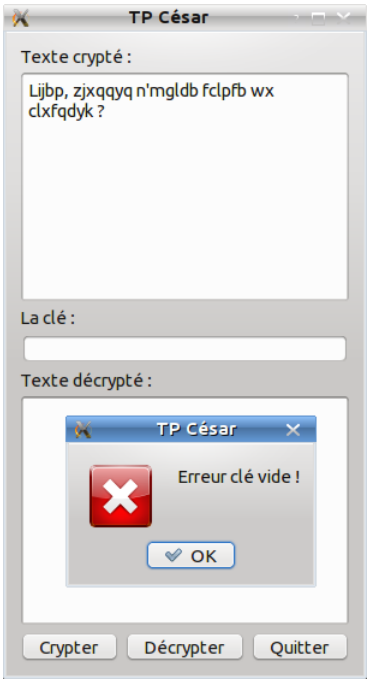
Et on obtiendra alors l'affichage suivant :



Le fonctionnement de l'application est assez simple :

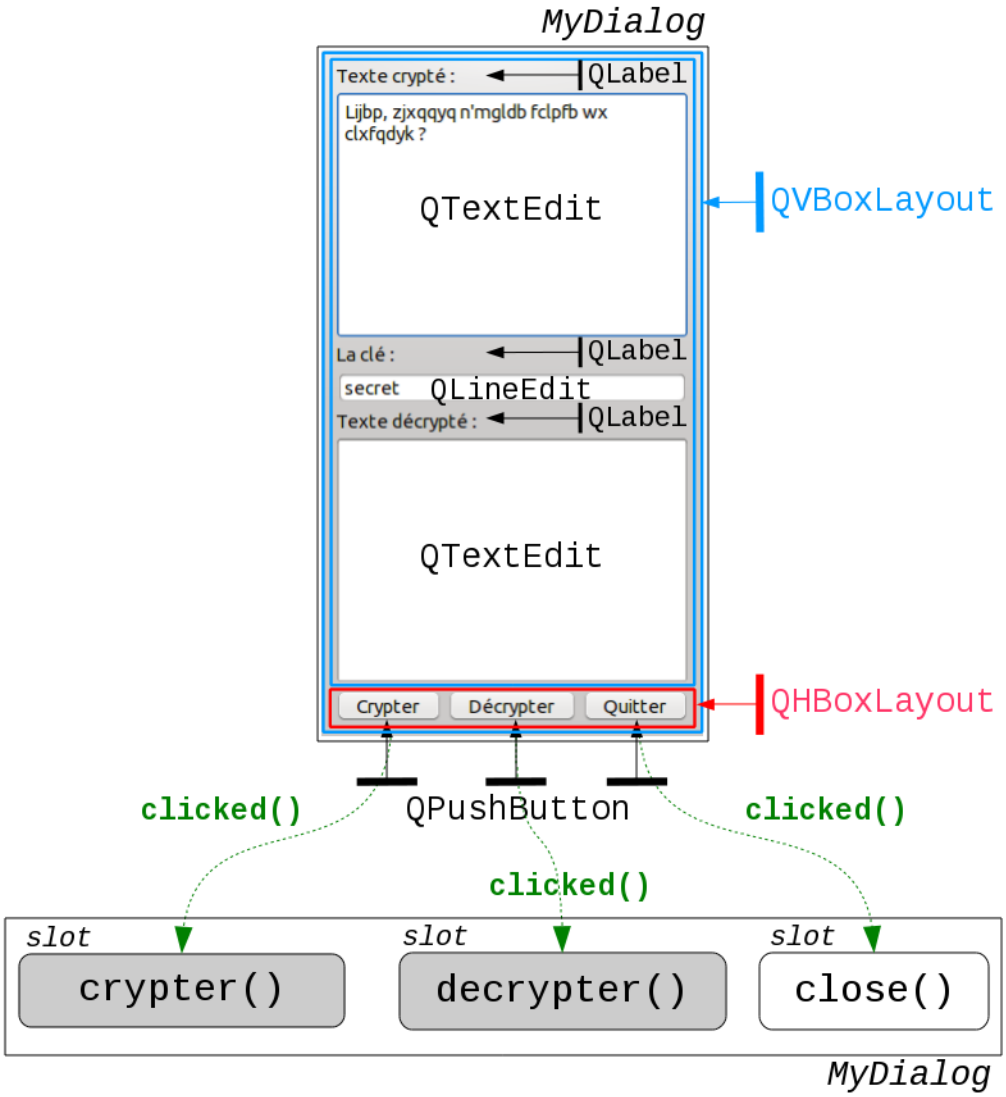
- le bouton Crypter assure le cryptage du texte saisi avec la clé fournie
- le bouton Décrypter assure le décryptage du texte saisi avec la clé fournie
- le bouton Quitter permet de terminer l'application

Si la clé saisie est vide, on affichera une boîte de dialogue d'erreur :



 © 2012 tv <tvaira@free.fr>

La classe `MyDialog` devra intégrer les différents éléments graphiques suivants :



Vous devez définir un positionnement de ces widgets avec des layouts afin de respecter l'affichage fourni dans la capture d'écran ci-dessus.

L'instanciation des widgets et leur positionnement se fera dans le constructeur de la classe **`MyDialog`** (voir Code 5).

Pour gérer les évenements et les actions entre l'utilisateur et l'application ou entre les widgets de l'application, Qt utlise un mécanisme d'évènements (signaux) et de récepteurs (slots). Chaque objet du programme (qui hérite de `QObject` et qui contient l'appel à la macro `Q_OBJECT` dans sa déclaration) peut émettre des signaux et proposer des slots de réception. Il suffit ensuite de connecter les signaux aux slots pour obtenir le déclenchement d'une action en présence d'un évènement. Ce mécanisme puissant permet aux différents objets de l'application de communiquer entre-eux de façon simple sans avoir à maintenir en permanence des références entre eux.

Dans notre situation, il faudra assurer au minimum la gestion des évèments et le déclenchement des actions représentées dans la capture d'écran ci-dessus.

La clé saisie dans un `QLineEdit` et les textes associées aux `QTextEdit` sont de type `QString`.

Question 1. Assurer la fabrication de l'application.

Afin de simplifier la fabrication, Qt offre l'utilitaire **`qmake`** qui permet de générer un fichier **`Makefile`** à partir d'un fichier projet (extension pro). Un fichier projet décrit les fichiers qui composent le programme (fichiers sources, headers, ...) et certaines options de construction (bibliothèques, ...). Pour ce TP, le fichier .pro généré par **`qmake`** sera le suivant :

```
#############################################################################
# Automatically generated by qmake (2.01a) dim. fvr. 5 16:14:41 2012
#############################################################################

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

# Input
HEADERS += mydialog.h
SOURCES += main.cpp mydialog.cpp
```

Code 3 – cesar.pro

Les différentes étapes pour fabriquer l'application demandée (sous GNU/Linux) sont :

Étape 1. Créer un répertoire **cesar**.

Étape 2. Copier les fichiers fournis (`main.cpp`, `mydialog.h` et `mydialog.cpp`) dans le répertoire **cesar**.

Étape 3. Compléter les fichiers fournis (`mydialog.h` et `mydialog.cpp`) : travail effectué aux questions 2 et 3.

Étape 4. Générer le fichier de projet `cesar.pro` en utilisant **`qmake`** dans le répertoire contenant les sources à compiler.

```
$ qmake -project
```

Étape 5. Générer le fichier `Makefile` en utilisant **`qmake`** dans le répertoire contenant les sources à compiler.

```
$ qmake
```

Étape 6. Fabiquer l'application en utilisant **`make`** ou `mingw32-make` sous Windows.

```
$ make
```

 © 2012 tv <tvaira@free.fr>

Étape 7. Tester l'application fabriquée.

```
$ ./cesar
```

Question 2. Compléter la déclaration de la classe **`MyDialog`**.

```
#ifndef MYDIALOG_H
#define MYDIALOG_H

#include <QtGui>

class MyDialog : public QDialog
{
   Q_OBJECT

   public:
      MyDialog( QWidget *parent = 0 );

   private:
      QLineEdit  *key;
      QTextEdit  *entree;
      QTextEdit  *sortie;
      QPushButton *bCrypter;
      QPushButton *bDecrypter;
      QPushButton *bQuitter;

      QChar cesar(QChar c, int cle);

   private slots:
      void crypter();
      void decrypter();
};

#endif
```

Code 4 – mydialog.h

Question 3. Compléter la définition de la classe **`MyDialog`**.

```
#include <QtGui>

#include "mydialog.h"

MyDialog::MyDialog( QWidget *parent ) : QDialog( parent )
{
   QLabel *texteCrypte = new QLabel(this);
   texteCrypte->setText(QString::fromUtf8("Texte crypte :"));

   // TODO
```

 © 2012 tv <tvaira@free.fr>

```
   setWindowTitle(QString::fromUtf8("TP Cesar"));
   setFixedHeight(sizeHint().height());
}

void MyDialog::crypter()
{
   // TODO
}

void MyDialog::decrypter()
{
   // TODO
}

QChar MyDialog::cesar(QChar c, int cle)
{
   // TODO
}
```

Code 5 – mydialog.cpp

Question 4. Fabriquer l'application et tester.

Itération 2

Une fois l'itération 1 validée, on passe à la seconde itération qui a pour but de finaliser l'application.

Remarque : dans un processus de développement itératif et incrémental, chaque développement s'ajoute et enrichit l'existant. Normalement, la nouvelle itération écrase la précédente et son numéro de version est incrémenté. Il est conseillé de conserver une copie fonctionnelle de l'itération 1. Dans certains cas, c'est une position de repli intéressante.

Question 5. Mise en oeuvre du code césar. Coder les méthodes `crypter()`, `decrypter()` et `cesar()`. Vous manipulerez les types `QString` et `QChar` fournis par Qt.

Nous allons utiliser une clé k différente pour chaque lettre du message. Pour obtenir cela, le plus simple est d'utiliser un mot de code (la clé) et d'utiliser le code ASCII de chacune de ses lettres pour donner chaque valeur de k.

Par exemple, pour crypter le message "Bonjour" avec la clé "cle", nous réalisons les opérations suivantes :
– répéter la clé autant de fois que nécéssaire sous le message, puis
– utiliser son code ASCII comme valeur de k.

Ainsi, on obtient :

– Texte en clair : Bonjour
– Clé : cleclec
– Texte crypté : Wskesrm

 © 2012 tv <tvaira@free.fr>

En effet, le code ASCII de 'c' est 99 et ('B'-'A'+99)%26+'A' donne 'W', le code ASCII de 'l' est 108 et ('o'-'a'+108)%26+'a' donne 's', le code ASCII de 'e' est 101 et ('n'-'a'+101)%26+'a' donne 'k', etc.

Le décryptage se fait sur le même principe mais en calculant : k = 26-code ASCII%26 ;

Remarques : la méthode `toAscii()` de la classe `QChar` renvoie la valeur ASCII du caractère. Et la méthode `at()` de la classe `QString` retourne un `QChar`.

Question 6. Trouvez la traduction du message suivant "Lijbp, zjxqqyq n'mgldb fclpfb wx clxfqdyk ?" en utilisant la clé "secret".

 © 2012 tv <tvaira@free.fr>