

TP Qt : Prise en main

Objectif

Le but de ce tp est une prise en main de la librairie Qt, et de ses outils associés, afin de créer des interfaces graphiques. Il est proposé de développer une calculatrice / convertisseur.

À tout moment, et avant d'appeler à l'aide, consulter la documentation de Qt disponible à l'adresse suivante :

- <http://doc.trolltech.com/4.4/index.html>

1 Fenêtre principale

1.1 Designer

Pour créer des widgets, fenêtres ou dialogues Qt met à disposition un logiciel permettant de dessiner les différents éléments de votre interface en se servant des widgets fournis par Qt. Lancer le designer Qt via la commande **designer** et créer une nouvelle **Form** de type “Fenêtre Principale”.

Ajouter sur le dialogue des boutons pour avoir une fenêtre qui ressemble à celle de la Figure 1. Dans cette fenêtre, vous devez utiliser les widgets suivants :

- 18 QPushButton¹.
- 1 QLineEdit en lecture seule.
- 1 Menu Fichier contenant un item Paramètres conversion ... et un item Quitter avec pour touches de raccourci la combinaison Ctrl+Q.
- 1 Menu Aide contenant un item À propos ... et un autre item À propos de Qt
- Les 10 boutons correspondant aux numéros de 0 à 9 et le bouton pour la virgule sont disposés dans un layout grille (QGridLayout).
- Les boutons des opérateurs et de conversion sont également dans un layout en grille (QGridLayout).
- Ces deux layouts sont disposés dans une layout horizontal (QHBoxLayout) avec un spacer horizontal les séparant.
- La fenêtre est dotée d'un layout vertical (QVBoxLayout).
- Un spacer vertical compresse les widgets vers le haut de la fenêtre.

Penser à donner un nom (par exemple **MainWindow**) à la fenêtre dessinée (champ **objectName** de l'objet **QMainWindow** dans l'éditeur de propriétés du designer).

1.2 Utilisation

1.2.1 Implantation de la fenêtre principale

Le designer Qt génère des fichiers xml (.ui) qu'il faut ensuite traduire en fichiers C++ (des .h). Pour cela, un utilitaire (**uic**) est fourni par Qt. Cet utilitaire génère un fichier qui porte le nom suivant **ui_nomfichier.h** où **nomfichier** est le nom du fichier .ui (sans son extension) généré par le designer. Dans ce fichier, une classe C++ est définie qui porte le nom donné au widget principal du fichier (si vous avez nommé votre objet de type **QMainWindow MainWindow**, cette classe sera **MainWindow**). Cette classe est utilisée pour construire dans l'application la fenêtre.

Pour cela, on définit une nouvelle classe qui hérite à la fois de **QMainWindow** et de la classe générée par **uic**.

Attention cette classe est encapsulée dans un namespace nommé **Ui**. Il faut donc préfixer son nom de **Ui::**.

Le fichier .h ressemblera à :

¹Une touche de raccourci pour les boutons numériques et les opérateurs peut être utile.



FIG. 1 – Fenêtre principale

```
#ifndef _QONVERTISSEUR_H_
#define _QONVERTISSEUR_H_

//Inclure le fichier contenant la definition de la fenetre
#include "ui_qonvertisseur.h"

class Qonvertisseur : public QMainWindow, private Ui::MainWindow
{
    Q_OBJECT //Necessaire a QT

public:
    Qonvertisseur(QWidget *parent = 0);
    ~Qonvertisseur();
};

#endif
```

Le fichier .cpp correspondant sera :

```
#include "qonvertisseur.h"

Qonvertisseur :: Qonvertisseur(QWidget *parent)
    : QMainWindow(parent), Ui::MainWindow()
{
    setupUi(this); //Permet d'initialiser les widgets
                    //(definie dans Ui::MainWindow)
}

Qonvertisseur :: ~Qonvertisseur() {}
```

Il existe d'autres méthodes pour utiliser des widgets construits dans le designer, consulter la documentation de Qt pour plus d'informations :
<http://doc.trolltech.com/4.4/designer-using-a-component.html>.

1.2.2 Fonction main

Pour pouvoir lancer notre programme, nous avons besoin d'une fonction `main` effectuant quelques initialisations et l'ouverture de la fenêtre principale de notre programme (via la fonction `show` définie dans `QMainWindow`). Dans un fichier .cpp, définissez cette fonction sur le modèle suivant :

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

int main(int argc, char *argv [])
```

```

{
    //Necessaire a Qt
    QApplication app(argc, argv);

    //Creation et ouverture de la fenetre
    Qonvertisseur mainwin;
    mainwin.show();

    //Lancement de la boucle principale de
    //gestion des evenements utilisateurs
    return app.exec();
}

```

1.3 Compilation : qmake

Pour compiler un programme Qt, plusieurs outils sont nécessaires : le compilateur **g++**, mais également certains outils propres à Qt comme **uic** ou **moc**².

Afin de simplifier la compilation, Qt offre l'utilitaire **qmake** qui permet de générer un fichier Makefile à partir d'un fichier projet (extension **.pro**). Ces fichiers projet décrivent les fichiers qui composent le programme (fichiers sources, ...) et certaines options construction du projet (librairies à inclures, type de projet à générer, ...). Pour votre projet, créer un fichier **.pro** sur le modèle suivant :

```

TEMPLATE = app
TARGET = Qonvertisseur
QT += core \
    gui
HEADERS += qonvertisseur.h
SOURCES += qonvertisseur.cpp \
    main.cpp
FORMS += qonvertisseur.ui

```

La variable **HEADERS** contient la liste³ des fichiers **.h** du projet, **SOURCES** contient la liste des fichiers **.cpp** du projet et **FORMS** la liste des fichiers **.ui** nécessaires à la construction du programme. **TARGET** définit le nom de l'exécutable généré. Pour plus d'informations sur les autres variables, consulter la documentation de **qmake** :

<http://doc.trolltech.com/4.4/qmake-manual.html>

Pour compiler le programme, taper la commande : **qmake fichier.pro** puis **make**.

Tester votre programme.

2 Gestion des événements utilisateurs : mécanisme de signaux/récepteurs

Principe de fonctionnement Pour gérer les événements et les actions entre l'utilisateur et le programme ou entre les widgets du programme, Qt utilise un mécanisme très puissant de *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. Ces signaux peuvent être captés par d'autres objets qui y sont connectés pour le signal donné. Ce mécanisme puissant permet aux différents objets de votre application de communiquer entre-eux de façon simple sans avoir à maintenir en permanence des références entre les différents objets. La contrepartie est qu'une étape de précompilation (avant l'appel de **g++**) est nécessaire. Cette étape est effectuée par le programme **moc** qui, à partir des fichiers **.h** et **.cpp** décrivant les classes, va générer un fichier **moc_*.cpp**.

qmake détermine automatiquement les fichiers nécessitant une précompilation et crée les règles de compilations adéquates dans le fichier Makefile. Pour plus de détails, je vous encourage vivement

²Nous verrons son utilité plus tard!

³Si la liste est sur plusieurs lignes, n'oublier pas le **\n** en fin de ligne (sauf pour la dernière) !

à lire la documentation de Qt explicitant ce point :
<http://doc.trolltech.com/4.4/signalsandslots.html>.

Mise en œuvre Afin de connecter deux objets entre-eux, vous devez utiliser la fonction statique `connect` dont le prototype est le suivant :

```
bool QObject::connect(const QObject* sender, const char* signal,
                      const QObject* receiver, const char* method,
                      Qt::ConnectionType type = Qt::AutoConnection
);
```

où `sender` et `receiver` sont deux pointeurs sur les objets à connecter. `signal` et `method` sont deux pointeurs représentant respectivement le signal et la fonction réceptrice concernés par cette connexion. La syntaxe de ces chaînes de caractères est un peu particulière et utilise deux macros :

- Pour le signal : `SIGNAL(nomSignal(typeArg1, typeArg2, ...))`
- Pour le récepteur : `SLOT(nomSlot(typeArg1, typeArg2, ...))`

Attention, le type des arguments du signal et du slot doivent être identiques, sinon la connexion ne pourra être effectuée. Un message d'erreur sera normalement généré lors de l'exécution.

Définition de signaux et de récepteurs Il est possible de définir ses propres signaux et récepteurs dans une classe. Il suffit de définir des méthodes de classe dont la visibilité est `signals` ou `{public | protected | private} slots` :

```
class C {
    Q_OBJECT
public:
    C();
    ~C();
public slots:
    void f1();
private slots:
    void f2(int a, float b);
signals:
    void s1(float b);
};
```

Les signaux ne nécessitent pas d'implantation (elle est créée dans le fichier générée par `moc`).

À tout moment, une fonction membre d'une classe fille de `QObject` peut émettre un signal en utilisant le mot clé `Qt emit` :

```
emit s1(2.0f);
```

Connexion et designer Il est possible de connecter des widgets directement à partir du `designer` via deux méthodes :

- En passant par “l’éditeur de Signal/Slot”.
- Ou en dessinant directement les connections entre les widgets (mode “édition de signal/slot”, accessible dans le menu “Edition”).

Connecter le signal `triggered` de l’item de menu `Quitter` du menu `Fichier` avec le slot `close` de la fenêtre principale de votre application.

Recompiler et tester.

3 Application

3.1 Boîtes de dialogue

3.1.1 Boîtes À propos

Qt offre des mécanismes pour créer facilement ce type de boîtes de dialogue.

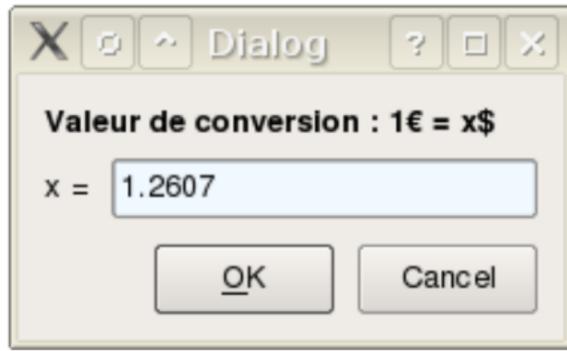


FIG. 2 – Dialogue des paramètres de conversion

Dans votre classe principale (ici `Qconvertisseur`), créer 2 slots privés associés aux signaux `triggered` des items de menu `Aide → À propos de Qt ...` et `Aide → À propos`

Dans le premier slot, ouvrir la boîte de dialogue “À propos de Qt...” grâce à la fonction statique `QApplication::aboutQt`. Faire de même avec une boîte de dialogue générée par la fonction statique `QMessageBox::about` dans le deuxième slot.

3.1.2 Paramètres de conversion

Créer un nouveau widget via le designer (de type “Dialogue avec boutons en bas”) qui devra ressembler à la capture de la Figure 2 (2 `QLabel` et 1 `QLineEdit`).

Créer la classe d’implémentation associée (qui dérive, cette fois, de la classe `QDialog`). Cette classe devra contenir une fonction permettant d’initialiser et de récupérer la valeur contenue dans la `QLineEdit`. Penser à vérifier que la valeur saisie est bien un nombre en utilisant, par exemple la fonction `toFloat` de la classe `QString`.

Créer le slot associé au signal `triggered` de l’item de menu `Fichier → Paramètres de conversion....` Dans ce slot, effectuer les actions suivantes :

1. Créer une boîte de dialogue de “Paramètres de conversion”.
2. Initialiser sa `QLineEdit` avec la valeur courante de conversion euro → dollar (qui peut être stockée dans une variable membre protégée de la classe `Qconvertisseur`).
3. Afficher la boîte de dialogue de façon modale (fonction `exec` définie dans `QDialog`).
4. Récupérer la valeur de conversion saisie et mettre à jour la variable membre de la classe `Qconversion` si l’utilisateur a cliqué sur `Ok`.

3.2 Actions sur les boutons

Boutons numériques Créer 10 slots pour chaque clique sur un bouton numériques (à connecter au signal `clicked` de chaque bouton).

L’action à effectuer dans ces slots est la suivante : récupérer la chaîne de caractère de la `QLineEdit` et y ajouter à la fin la valeur du bouton.

Bouton séparateur décimal Le comportement de ce bouton est similaire aux boutons numériques. Cependant, ce bouton ne doit plus être actif quand un point a été saisi ou quand la valeur affichée dans la `QLineEdit` contient un point. Il faut donc prévoir un mécanisme qui mémorise si ce bouton a été pressé dans l’expression actuelle et le désactiver le cas échéant.

Opérations Pour simplifier, on ne gère pas les priorités des opérations.

Vous aurez besoin de mémoriser au minimum le premier opérande et l’opérateur courant.

Attention à la division par 0.

Le bouton “=” permet de calculer et d’afficher le résultat.

Les conversions utilisent qu’un seul opérande et la valeur de conversion saisie dans la boîte de dialogue des “Paramètres de conversion” et affichent directement le résultat sur la `QLineEdit`.