

Introduction à la programmation Qt

Qt est une librairie de programmation graphique écrite en C++. Totalement Open Source elle est disponible sur les principales plates-formes du marché, de l'AIX au téléphone portable en passant par l'ordinateur de bureau et le PDA !

Cette bibliothèque est écrite avec une préoccupation majeure de portabilité. Qt est, en effet, multi-plate-forme par simple recompilation. Ceci fait gagner un temps précieux lors du développement d'applications multi-plates-formes. Google ne s'y est pas trompé lorsque le choix de Qt fut fait pour le développement de Google Earth. Dans cette série de 3 articles, nous allons développer une application complète avec Qt 4 en C++. Si le lecteur n'est pas familier de ce langage ce n'est pas rédhibitoire. En effet, il est possible de développer avec Qt dans quasiment n'importe quel langage. Trolltech propose Jambi, la version Java de son framework, la communauté quant à elle propose des binding pour les principaux langages de scripts tels que Perl (Perl Qt 4), Python (PyQt4) ou encore Ruby. Les concepts sont les mêmes pour tous ces langages et les 3 articles de cette série sont très facilement transposables.

Au commencement...

... Était le cahier des charges.

Dans un premier temps, donc, parlons de l'application que nous allons développer. Nommée QuickDesktopNotes, celle-ci permet une prise de notes rapide sur le bureau via un icône dans le " system tray ". Le but est d'avoir une application à la fois rapide et peu intrusive en terme d'occupation du bureau. Ajoutons au cahier des charges la nécessité de gérer les notes multiples, la sauvegarde de ces notes sous forme d'un flux RSS (pour l'interopérabilité) ainsi qu'une fenêtre de configuration pour gérer l'application.

En résumés nous implémenterons 2 fenêtres communicantes, un " tray icon " et un parser RSS. Le tout en moins de 500 lignes de code.

QuickDesktopNotes va nous permettre de nous familiariser avec les principaux concepts de programmation Qt (héritage, signaux/slots, etc.), les outils de développement (designer, assistant, qmake, etc.) et la programmation d'IHM (Interface Hommes Machines).

Quelques petites remarques avant d'entrer dans le vif du sujet :

- QuickDesktopNotes a été développé en partie sous Linux, en partie sous Windows XP.
- En de (très) nombreuses occasions il sera fait référence à la (volumineuse) documentation de Qt. Celle-ci est livrée avec Qt et est accessible via le logiciel " assistant ", qui permet une navigation simple et rapide dans l'API, les exemples et autres tutoriels disponibles.
- Pour la gestion de projet ainsi que pour la compilation de l'application, un shell est nécessaire, assurez-vous donc de posséder cet outil avant de commencer.

PNoteEditor : la fenêtre d'édition des notes.

Dans un premier temps, nous nous occuperons de la création de la fenêtre d'édition des notes. Ceci se fera selon le processus classique : nous allons dessiner la fenêtre dans le designer, puis hériter de la classe générée afin d'implémenter les fonctionnalités de notre fenêtre. En effet, nous ne voulons pas perdre de temps à réinventer la roue, nous

souhaitons par contre profiter de toutes les fonctionnalités d'une fenêtre d'édition classique (copier/coller, drag'n'drop, etc.). Commencez par créer un répertoire QuickDesktopNotes. Puis lancez le designer, vous devriez vous trouver en face de la fenêtre suivante : (Fig.1)

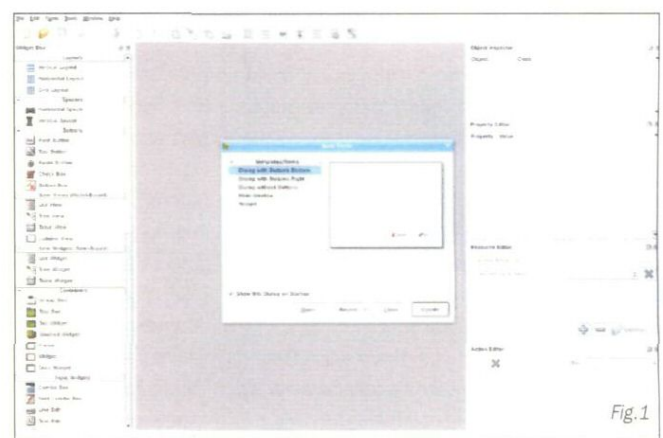


Fig.1



Fig.2

Choisissez de créer un nouveau widget en sélectionnant le template " widget ". Designer crée alors un formulaire vide de type QWidget. Modifiez sa propriété objectName pour la positionner à " PNoteEditor ". (Fig.2)

Ajoutez un QLineEdit au formulaire vide. Ceci se fait en glissant l'élément désiré depuis la " widget box " vers le formulaire. Modifiez ensuite la propriété " objectName " pour affecter le nom " title " à la place de " lineEdit ". Cet élément servira, comme son nom le laisse supposer, à saisir le titre d'une note. Cela sera utile pour identifier une note précise par la suite. (Fig.3)

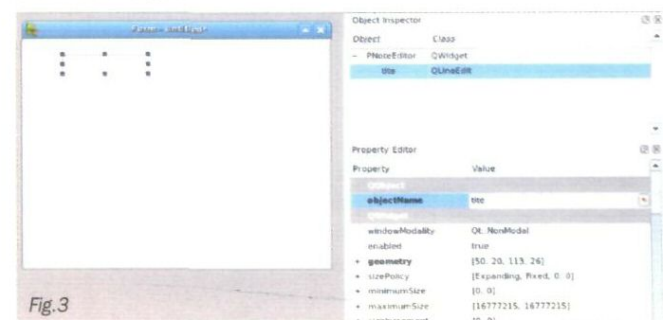


Fig.3

Il faut ensuite ajouter une zone de saisie de texte. Cette fonctionnalité est prise en charge par le widget QTextEdit, que l'on ajoutera de la même façon que précédemment. Nommez ce dernier " editor ". (Fig.4)

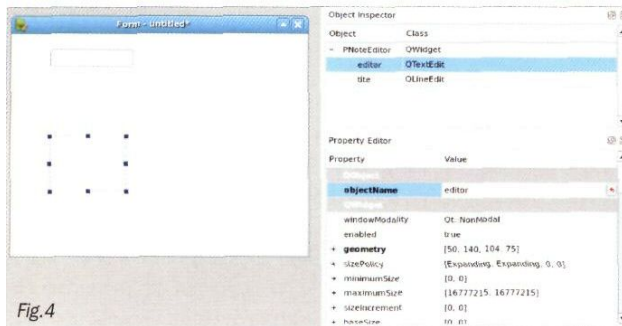


Fig.4

Une fois ces 2 éléments mis en place il reste à s'assurer d'une disposition dynamique des 2 éléments graphiques à l'intérieur de notre formulaire. Le mécanisme proposé par Qt pour gérer les dispositions est nommé QLayout. Il se décline en différentes versions comme les dispositions " box " ou " grid " (QBoxLayout ou QGridLayout). Le principe général des " layout " est basé sur la taille optimale d'un widget. En effet, chaque widget retourne une taille qu'il considère comme optimale pour lui-même (détenue par la propriété sizeHint). Les objets layout se servent de cette taille pour disposer les éléments graphiques sur le formulaire. Dans le cas de notre fenêtre, nous choisirons une disposition sur grille. Pour cela, sélectionnez le fond du widget (ni le QLineEdit ni le QTextEdit), puis cliquez sur l'icône " Layout in a Grid " dans la barre d'outils. Cela va disposer automatiquement les 2 éléments de façon optimale. (Fig.5)

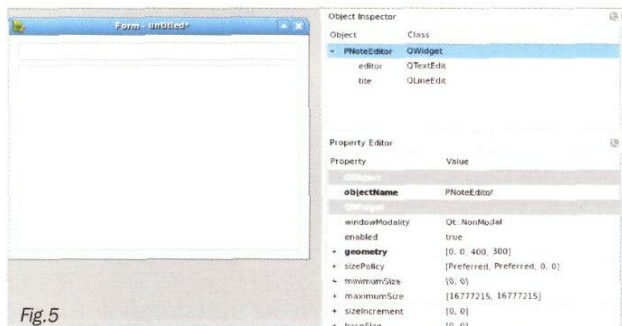


Fig.5

Il reste désormais à sauvegarder le formulaire dans le répertoire Quick-DesktopNotes sous le nom *pnoteeditor.ui*. Les fichiers .ui sont automatiquement compilés par l'uic (User Interface Compiler) en un fichier d'en-tête (.h) préfixé par ui_ (et déclarant l'espace de nom Ui::<objectName>, ici Ui::PNoteEditor), ce fichier contient toutes les définitions de code en ligne. Nous ne serons pas contraints d'appeler uic nous-mêmes car cela sera effectué automatiquement à la compilation. En l'état notre fenêtre est utilisable, il suffirait de s'en servir dans un programme avec un QWidget comme base, d'instancier la classe Ui::PNoteEditor généré par uic, puis d'appeler la méthode setupUi() sur le QWidget précédemment instancié. Le code ressemblerait à cela :

```
QWidget *widget = new QWidget;
Ui::PNoteEditor ui;
ui.setupUi(widget);
```

Cependant, bien que la fenêtre s'affiche et que les fonctionnalités d'édition de QTextEdit et QLineEdit soient bien présentes, il manque certains mécanismes que nous désirons avoir à disposition.

Premièrement, il serait très utile que les 2 éléments de saisie signalent les modifications qui sont apportées à leur contenu. Il est aussi souhaitable de fournir des accesseurs qui permettront de retirer ou de positionner les valeurs des 2 zones de saisies. En effet, cette fenêtre ne sera pas utilisée comme plusieurs éléments graphiques séparés mais comme un widget unique.

Nous allons donc mettre en place une solution plus évoluée que la simple instanciation d'un formulaire : nous allons créer une classe PNoteEditor qui va hériter de Ui::PNoteEditor.

Afin de fournir aussi les possibilités des widgets Qt, nous allons également hériter (publiquement) de QWidget. L'héritage de Ui::PNoteEditor sera lui privé, car nous ne désirons pas que la classe soit accédée autrement que par les accesseurs que nous allons proposer.

PNoteEditor : le code derrière la fenêtre.

Voyons dans un premier temps le fichier d'en-tête contenant la définition de la classe :

```
#ifndef PNOTEEDITOR_H
#define PNOTEEDITOR_H

#include <QWidget>
#include "ui_pnoteeditor.h"

class PNoteEditor : public QWidget, private Ui::PNoteEditor {
    Q_OBJECT
public:
    PNoteEditor( QWidget *parent=0);
    QString noteText();
    QString noteTitle();

signals:
    void noteTitleChanged(const QString&);
    void noteTextChanged(const QString&);

public slots:
    void setNoteText(const QString&);
    void setNoteTitle(const QString&);

private slots:
    void on_title_textChanged();
    void on_editor_textChanged();
};

#endif
```

Dans ce fichier nous commençons par nous prémunir contre l'inclusion multiple aux lignes 1 et 2, puis nous incluons la définition de la classe QWidget (ce qui est obligatoire car nous en héritons) ainsi que l'en-tête de Ui::PNoteEditor stockée dans le fichier ui_pnoteeditor.h. Notez au passage que le fichier ui_pnoteeditor.h n'existe pas encore, nous verrons plus loin comment faire pour qu'il soit généré automatiquement. Vient ensuite la déclaration de la classe PNoteEditor (l. 7) :

```
class PNoteEditor : public QWidget, private Ui::PNoteEditor {
    ...
};
```

PNoteEditor hérite simultanément de QWidget de façon publique (c'est-

Programmez!

à-dire que l'API de QWidget sera directement accessible depuis PNoteEditor ou d'une de ses instances) et de Ui::PNoteEditor de façon privée cette fois. Nous avons, ici, recours à l'héritage privé car la classe générée à partir de pnoteeditor.ui déclare toutes ses méthodes et attributs publics. Or ce n'est pas un comportement souhaitable et ce mécanisme n'est disponible que dans le but de simplifier la création de l'interface en permettant l'héritage de l'intégralité des variables, pointeurs et autres attributs de la classe Ui:: de base.

La première chose à faire lors de la création d'un objet Qt est l'insertion de la macro Q_OBJECT. Celle-ci permet l'intégration par le compilateur de méta-objet " moc " (pour Meta Object Compiler) de tout le code nécessaire aux objets Qt (tels que le mécanisme des signaux/slots, le système de propriétés et plus généralement tous les mécanismes d'introspection).

Ensuite, nous définissons les méthodes publiques, qui sont au nombre de 2 plus le constructeur. Celui-ci peut prendre un argument éventuel : son widget parent. Cela permet d'utiliser l'objet PNoteEditor dans un autre programme Qt. Les 2 méthodes sont noteTitle() qui retourne le contenu du QLineEdit (title) sous forme de QString, et noteText() qui retourne aussi une QString représentant le corps de la note renvoyé depuis le QTextEdit (editor).

Les méthodes permettant de positionner le contenu de ces 2 éléments sont déclarées comme des slots publics : setNoteText(const QString &) et setNoteTitle(const QString&) qui permettent de positionner respectivement le corps et le titre de la note.

Le mécanisme signal/slot de Qt est à la fois pratique et ingénieux, il permet à un objet Qt de sélectionner les événements importants, de les digérer et d'émettre un signal spécifiquement adapté au contexte. Prenons un exemple : l'événement transmis aux objets Qt lors de l'appui d'une touche de clavier (QKeyEvent) n'est pas toujours intéressant. Dans le cas d'un widget tel que QLineEdit, en revanche, cela peut être très important si la touche sur laquelle l'utilisateur vient d'appuyer modifie le texte contenu dans la zone de saisie. Le widget QLineEdit filtre donc les événements qui lui sont rapportés et quand il détecte l'appui sur une touche ayant un impact sur le texte de la zone de saisie, il émet le signal textChanged() qui informe les objets qui y sont connectés que le texte a changé dans le QLineEdit. Un signal ne fait rien en lui-même, il doit être connecté à un slot qui s'occupe de le gérer. Un slot est une fonction C++ classique et de ce fait peut être virtuel, public, privé, surchargé, etc. La seule différence qui existe entre un slot et une fonction C++ traditionnelle est qu'un slot peut être connecté à un signal. Dans ce cas de figure, le slot est appelé automatiquement à chaque fois que le signal est émis.

Un signal peut être connecté à un ou plusieurs slots, plusieurs signaux peuvent être connectés au même slot, enfin, un signal peut être connecté à un autre signal.

Pour connecter un signal à un slot il suffit de faire appel à la fonction statique QObject::connect (le prefix QObject n'est pas nécessaire si la macro Q_OBJECT a été placée dans la classe).

Exemple :

```
connect(widget, SIGNAL( unSignal() ), this, SLOT( unSlot() ) );
```

Pour qu'un signal soit correctement connecté à un slot, il faut que les arguments soient compatibles. Par exemple, le signal textChanged(const QString &) ne peut pas être correctement connecté au slot setPa-

ge(int) alors qu'il se connecte parfaitement au slot setNoteText(const QString &). Une exception à cette règle existe : la connexion à un slot avec moins de paramètres que le signal est possible mais pas encouragée. PNoteEditor déclare lui même 2 signaux : noteTitleChanged(const QString&) et noteTextChanged(const QString&). Ceux-ci sont émis à chaque fois que le texte de leur élément de saisie respectif change. L'émission des signaux à proprement parler est prise en charge par les 2 slots privés : on_title_textChanged() et on_editor_textChanged(). Ces derniers sont un peu spéciaux. En effet, nous pouvons constater que le nom de ces 2 méthodes respecte le même format on_<nom d'un élément graphique>_<quelque chose>(<paramètre(s)>). Ceci est une fonctionnalité particulièrement pratique introduite dans Qt 4 : la connexion par nom de slot !

Il n'est pas obligatoire, en effet, de connecter explicitement slots et signaux. Le moc ajoute, à la compilation, le code nécessaire à la connexion par nom de slots (voir QObject::connectSlotsByName dans l'assistant pour plus d'informations à ce propos).

Le mécanisme signal/slot est proposé par le système de méta-objets de Qt. Celui-ci permet d'écrire des classes communicantes sans qu'aucune de celles-ci ne sachent quoi que ce soit sur les autres. Ce système propose aussi un mécanisme d'introspection que nous mettrons en pratique dans la suite de ce dossier.

Voyons maintenant le code de la classe (stocké dans le fichier source pnoteeditor.cpp). Celui-ci n'est pas bien complexe. Tout d'abord il faut inclure le fichier d'en-tête comme dans n'importe quel programme C++.

```
#include "pnoteeditor.h"
```

Il est ensuite nécessaire de mettre en place le constructeur de la classe PNoteEditor :

```
PNoteEditor::PNoteEditor( QWidget *parent ) : QWidget(parent){
    setupUi(this);
}
```

Ce dernier ne fait rien d'autre que d'appeler le constructeur de la classe parent (QWidget) et de mettre en place l'interface graphique que nous avons dessinée dans le designer (setupUi() provient de Ui::PNoteEditor). A la suite de cela viennent les 2 slots privés. Ceux-ci sont appelés à chaque fois que le texte de l'éditeur (le QTextEdit editor) ou de la barre de titre (le QLineEdit title) change. Nous en sommes avertis car chacun de ces widgets émet le signal textChanged(), et grâce à la connexion par nom de slots, on_editor_textChanged() et on_title_textChanged() sont appelés à chaque émission du signal par leur widget respectif.

```
void PNoteEditor::on_editor_textChanged(){
    emit( noteTextChanged( editor->PlainText() ) );
}
void PNoteEditor::on_title_textChanged(){
    emit( noteTitleChanged( title->text() ) );
}
```

On remarquera l'utilisation du mot clé emit() pour l'émission d'un signal. C'est là tout le code que nous devons écrire pour profiter de ce puissant mécanisme. Le code des accesseurs est lui aussi très simple dans la mesure où il suffit de retourner ou de positionner le texte de la barre de

titre et de l'éditeur. Le widget QLineEdit possède une méthode de lecture de son contenu : text() et QTextEdit quant à lui implémente la méthode toPlainText(). Ces 2 méthodes retournent des QString, il suffit donc de retourner directement le résultat de leur appel respectif :

```
QString PNoteEditor::noteTitle(){
    return title->text();
}
QString PNoteEditor::noteText(){
    return editor->toPlainText();
}
```

Pour finir, il reste à écrire le code des 2 slots setNoteText(const QString&) et setNoteTitle(const QString&) :

```
void PNoteEditor::setNoteText(const QString& str){
    editor->setText(str);
}
void PNoteEditor::setNoteTitle(const QString& str){
    title->setText(str);
}
```

Là encore, la simplicité du code laisse rêveur !

Afin de tester ce code nous allons écrire un dernier fichier source : main.cpp. Celui-ci ne contient que le code de la fonction principale main() :

```
#include <QtGui>
#include "pnoteeditor.h"
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    PNoteEditor window;
    window.show();
    return app.exec();
}
```

Cette fonction crée une nouvelle application Qt puis instancie et affiche PNoteEditor. Le recours à QApplication nous permet, par exemple, d'ajouter le support des options de base des applications Qt comme "-style" ou "-title". Le code de cette première fenêtre étant désormais terminé il reste à le compiler. En avant !

Compilation et exécution d'un programme Qt :

Dans cette dernière partie, nous allons avoir besoin d'un terminal car les outils de compilation sont en lignes de commande.

La première chose à faire est de se rendre dans le répertoire QuickDesktopNotes dans lequel sont stockés tous les fichiers sources du programme :

```
cd <chemin d'accès>/QuickDesktopNotes/
```

Une fois dans ce répertoire l'affichage de son contenu devrait donner cela :

```
arno@Shai_Hulud:~/progz/Qt4/QuickDesktopNotes$ ls
pnoteeditor.h main.cpp pnoteeditor.cpp pnoteeditor.ui
```

Les utilisateurs des systèmes d'exploitation Windows doivent utiliser la

commande "dir" au lieu de "ls". Afin de créer les fichiers de compilation pour le projet, nous allons utiliser l'outil qmake fourni avec toutes les distributions de Qt :

```
qmake -project
```

Ceci crée le fichier (texte brut) QuickDesktopNotes.pro, je vous conseille de vous familiariser avec sa syntaxe car nous allons très vite le modifier à la main ! Une fois le fichier projet (multi-plate-forme) généré, il faut créer les directives de compilation propres à la plate-forme hôte en appelant qmake sur le fichier projet :

```
qmake QuickDesktopNotes.pro
```

Enfin, suivant votre environnement de compilation il suffit d'invoquer les commandes de compilation, dont le plus classique :

```
make
```

Une fois la compilation terminée, il ne reste plus qu'à lancer le programme et à admirer le résultat !

Dans le prochain article nous mettrons en place le "system tray icon" et nous utiliserons PNoteEditor comme éditeur de note.

D'ici là bon développement.

Pré-requis :

Pour toutes les plates-formes, la bibliothèque Qt-4.3.x (à l'écriture de ces lignes la version courante est la 4.3.4) est un pré-requis impératif à cet article.

Linux :

Quasiment toutes les distributions Linux proposent un paquetage Qt 4 installable via l'outil de gestion de paquetage de la distribution.

Mac OS X :

Trolltech fournit un paquetage (.dmg) précompilé dans la section "download" de son site. Il est fortement conseillé d'installer aussi le paquetage contenant les bibliothèques de debug (QDebug).

Windows :

Comme pour Mac OS X un paquetage (.exe) est proposé par la société Trolltech. Pour cette plate-forme en revanche le paquetage est proposé sous 2 formes : incluant le compilateur MinGW ou non. Si vous n'avez pas d'environnement de développement pré-installé, nous conseillons fortement cette version basée sur GCC. Cela garantit un passage d'une plate-forme à l'autre moins douloureux...

Tous ces paquetages plus les sources sont disponibles ici : <http://trolltech.com/downloads/>

■ Arnaud Dupuis

Consultant Uperto/ Groupe Devoteam