Le site francophone Qt sur l'Internet : **http://prog.qt.free.fr**

Je tiens à remercier particulièrement Nicolas Arnaud pour le travail de relecture qu'il a bien voulu réaliser.

Débuter avec Qt 3

Jean-Luc Biord

Septembre 2004

ISBN 2-9522843-0-X

Table des Matières

Introduction

Qt est une bibliothèque de programmation C++. Son utilisation repose donc sur la connaissance du langage de programmation. Cet ouvrage n'expose que les fonctionnalités de Qt sans s'attarder sur le langage. En effet le nombre de livres traitant du C++ est largement suffisant pour permettre un apprentissage de celui-ci. De plus, cela ne ferait qu'augmenter le nombre de pages au détriment de l'apprentissage de Qt lui-même. Néanmoins, lorsque c'est indispensable à la compréhension du sujet traité, quelques notions de C++ seront rappelées afin de ne pas noyer le lecteur dans de multiples problèmes de compréhension.

Trolltech a annoncé récemment la sortie de Qt 4. Un livre sur Qt 3 n'est quand même pas inutile. D'une part au moment où sortira ce livre, la version 3 en aura encore pour quelques mois, voir plus. D'autre part, la version 4 n'est pas une révolution. Même si certains concepts de programmation ont été repensés, cette version est la suite de la 3 et beaucoup de principes restent les mêmes. Quelqu'un qui est à l'aise avec Qt 3 s'en sortira avec la version 4.

Le code des exemples montrés dans ce livre est accessible en téléchargement sur le site http://prog.qt.free.fr, le site francophone de la programmation Qt.

Le forum du site propose une rubrique dédiée à cet ouvrage. Le lecteur pourra y trouver les éventuelles corrections apportées depuis son édition. Chacun pourra également poser des questions pour préciser des points qui n'auraient pas été assimilés.

1. Pour commencer

1.1. Les outils fournis par Qt

Qt est fourni avec un certain nombre d'outils en ligne de commande ou graphiques qui facilitent grandement le développement des applications.

Afin d'avoir un fil conducteur, nous allons utiliser un petit programme d'exemple qui nous servira à illustrer l'utilisation des différents outils. Différents fichiers présents dans le même répertoire nommé par exemple `hello` contiennent les sources de ce programme.

Voici tout d'abord `hello.ui` :

```
1    <!DOCTYPE UI><UI version="3.2"
stdsetdef="1">
2    <class>Hello</class>
3    <widget class="QDialog">
4        <property name="name">
5            <cstring>Hello</cstring>
6        </property>
7        <property name="geometry">
8            <rect>
9                <x>0</x>
10               <y>0</y>
11               <width>163</width>
12               <height>63</height>
13           </rect>
14       </property>
15       <property name="caption">
```

Figure 1.1. Notre petit dialogue.

```
16              <string>Hello World</string>
17          </property>
18          <widget class="QLabel">
19              <property name="name">
20                  <cstring>textLabel2</cstring>
21              </property>
22              <property name="geometry">
23                  <rect>
24                      <x>30</x>
25                      <y>20</y>
26                      <width>66</width>
27                      <height>20</height>
28                  </rect>
29              </property>
30              <property name="text">
31                  <string>Bonjour</string>
32              </property>
33          </widget>
34      </widget>
35      <layoutdefaults spacing="6" margin="11"/>
36      </UI>
```

C'est un peu brutal de montrer ici le contenu d'un fichier d'interface. Un fichier (.ui) est un fichier contenant du XML qui est généré par l'éditeur d'interface `Qt Designer`. Notez que le développeur n'a ja-

mais à manipuler directement des fichiers au format XML, c'est `Qt Designer` qui s'en charge. Mais comme nous ne connaissons pas encore `Qt Designer` et qu'il serait prématuré de l'utiliser, le contenu du fichier est montré afin de pouvoir tester avec lui les outils qui vont être présentés.

Notre programme contient une classe appelée `HelloImpl` dont voici le fichier d'en-tête `helloimpl.h`:

```
1    #ifndef HELLOIMPL_H
2    #define HELLOIMPL_H
3
4    #include <qdialog.h>
5    #include "hello.h"
6
7    class HelloImpl : public Hello
8    {
9        Q_OBJECT
10   public:
11        HelloImpl(
12             QWidget* parent = 0,
13             const char* name = 0);
14   signals:
15        void monSignal();
16
17   public slots:
18        void slotAMoi();
19
20   };
21   #endif
```

et le fichier d'implémentation `helloimpl.cpp`

```
1    #include "helloimpl.h"
2
3    HelloImpl::HelloImpl(
4              QWidget* parent,
```

```
5               const char* name)
6           : Hello(
7               parent,
8               name)
9   {
10  }
11
12  void HelloImpl::slotAMoi()
13  {
14  }
```

Enfin, un fichier `main.cpp` est nécessaire pour faire fonctionner l'ensemble :

```
1   #include <qapplication.h>
2   #include "helloimpl.h"
3
4   int main( int argc, char **argv )
5   {
6     QApplication a( argc, argv );
7
8     HelloImpl *hello = new HelloImpl(0);
9
10    a.setMainWidget( hello );
11    hello->show();
12    return a.exec();
13  }
```

Ces fichiers seront utilisés dans ce qui suit pour servir d'exemples aux outils de Qt. Notez que ces fichiers utilisent des notions qui seront abordées plus tard dans l'ouvrage. Il est peu important, pour l'instant, de les comprendre.

1.1.1. qmake

`qmake` permet de créer des fichiers Makefile à partir de fichier-

projets simples indépendants de la plate-forme.

L'un des outils les plus utilisés en C/C++ est `make`. Il est employé afin de construire les programmes exécutables en compilant si nécessaire chacun des fichiers sources. Par comparaison de date de création/mise à jour, il évite de recompiler des sources inutilement. C'est aussi `make`, qui va transformer les fichiers d'interface (.ui) en lançant l'utilitaire adéquat.

Un fichier Makefile ou (makefile) est un fichier contenant les informations nécessaires à la commande `make` et déterminant quelles actions doivent être faites pour mettre à jour un exécutable. Cet exécutable est assemblé en compilant un ou plusieurs fichiers sources. Le fichier `Makefile` doit se trouver dans le répertoire courant lorsqu'on appelle `make` à l'invite du shell.

Ceux qui ont déjà écrit des fichiers Makefile à la main savent à quel point la chose est difficile. Avec `qmake`, le développeur a seulement besoin de créer un fichier projet relativement simple puis de lancer `qmake` pour générer un fichier Makefile adapté au système hôte.

`qmake` utilise des informations stockées dans des fichiers projets (.pro). Un fichier projet contient des informations concernant l'application, c'est-à-dire les fichiers nécessaires pour compiler l'application et le type de configuration qui doit être utilisé.

La bonne utilisation de `qmake` et de Qt en général dépend de trois variables d'environnement :

QTDIR: Elle indique le répertoire d'installation de Qt. Avec Linux Mandrake par exemple, cette installation est effectuée dans `/usr/lib/qt3`. Cela peut être vérifié en entrant dans une console la commande suivante : *echo $QTDIR.*

PATH: Cette variable définit le chemin d'accès aux programmes exécutables. Un programme lancé uniquement par son nom sera cherché dans les répertoires contenus dans `PATH`. Sous Linux, si cette variable ne le contient pas déjà, ajoutez le chemin d'accès au répertoire des fichiers exécutables de Qt par : *export set $QTDIR/bin.*

QMAKESPEC: Elle doit être définie et contenir une combinaison indiquant la plateforme et le compilateur que vous employez sur votre système. Par exemple, si vous employez Windows et Visual Studio de Microsoft, vous placerez cette variable d'environnement à `win32-msvc`. Si vous employez Linux et g++, vous placerez cette variable d'environnement à `linux-g++`. Lorsqu'elle n'est pas définie, Qt utilise le lien symbolique `default` présent dans le répertoire `mkspecs` et qui pointe vers le répertoire désignant l'environnement de travail.

Si vous avez procédé à l'installation de Qt à partir des fichiers d'installations, les variables d'environnements ont été correctement positionnées et vous n'avez normalement rien à faire de plus.

`qmake` reconnaît dans ses fichiers projets plusieurs variables que voici:

TEMPLATE: Cette variable contient le nom du modèle à utiliser pour générer le projet. Elle peut avoir les valeurs suivantes:

- **app**: Crée un fichier Makefile pour construire l'application (valeur par défaut).
- **lib**: Crée un Makefile pour construire une bibliothèque.
- **vcapp**: Crée un projet d'application pour Visual Studio (Windows seulement).

TARGET: Cette variable spécifie le nom du programme ou de la bibliothèque à créer.

Par exemple:

```
TEMPLATE = app
TARGET = monappli
```

TRANSLATIONS: Cette variable spécifie les fichiers de traduction à inclure au projet.

Par exemple:

```
TRANSLATIONS =  monappli_de.ts \
                monappli_en.ts
```

INCLUDEPATH: Cette variable spécifie le chemin à ajouter dans le chemin de recherche des fichiers d'en-têtes et de bibliothèques.

Par exemple:

```
INCLUDEPATH = /home/brd/mabibliotheque
```

SOURCES: Cette variable définit les fichiers d'implémentations (.cpp) nécessaires au projet.

Par exemple:

```
SOURCES = mondialogue.cpp \
          login.cpp \
          mainwindow.cpp
```

HEADERS: Cette variable définit les fichiers d'en-têtes (.h) nécessaires au projet.

Par exemple:

```
HEADERS = mondialogue.h \
          login.h \
          mainwindow.h
```

FORMS ou **INTERFACES**: Cette variable spécifie le nom des fichiers (.ui) à inclure. Ces fichiers seront transformés par l'outil `uic` en fichiers C++.

Par exemple:

```
FORMS = mondialogue.ui \
        maconfig.ui
```

IMAGES: Cette variable définit les fichiers images à intégrer dans une collection d'images. `uic` confectionnera un fichier nommé par défaut `qmake_image_collection.cpp`. Dans ce fichier seront intégrées les images sous forme de tableaux de données.

Par exemple:

```
IMAGES = couper.png \
         copier.png \
         coller.png
```

CONFIG: Cette variable spécifie la configuration du projet et les options de compilation. Les valeurs suivantes contrôlent les options de compilation:

- **release**: Compile avec l'optimisation activée, ignoré si "debug" est spécifié.
- **debug**: Compile avec les options de débogage activées. Le programme pourra être tracé avec un outil de débogage comme `gdb` . Voir le chapitre "Techniques de débogage".
- **warm_on**: Le compilateur affichera plus d'avertissements que la normale, ignoré si "warm_off" est spécifié.
- **warm_off**: Le compilateur n'affichera que les avertissements importants.

Les options suivantes définissent le type d'application/bibliothèque:

- **qt**: La cible est une application ou bibliothèque Qt.
- **opengl**: La cible nécessite les en-têtes/bibliothèque OpenGL (ou Mesa). Les chemins vers ces bibliothèques sont automatiquement inclus au projet.
- **thread**: La cible est une application/bibliothèque multi-thread.

- **x11**: La cible est une application ou bibliothèque x11.
- **windows**: La cible est une application Win32.
- **console**: La cible est une application Win32 console.
- **dll**: La cible est une DLL.
- **staticlib**: La cible est une bibliothèque statique (seulement pour les bibliothèques).

Par exemple:

```
CONFIG += qt warm_on debug
```

Le fichier projet `hello.pro` contient les informations nécessaires à la compilation de notre programme. Nous indiquons le nom des fichiers sources, d'en-têtes ainsi que des informations concernant la configuration du programme.

```
1       TEMPLATE = app
2       INCLUDEPATH += .
3
4       HEADERS += helloimpl.h
5       INTERFACES += hello.ui
6       SOURCES += helloimpl.cpp main.cpp
```

Nos quatre fichiers y sont bien présents dans leur rubrique respective.

Dans un fichier projet, une variable peut être initialisée avec =

```
SOURCES = premier.cpp
```

SOURCES contient premier.cpp.

On peut lui rajouter du contenu par +=

```
SOURCES += deuxieme.cpp
```

SOURCES contient maintenant premier.cpp deuxieme.cpp.

De la même façon -= supprime de la variable la valeur transmise

```
SOURCES -= premier.cpp
```

SOURCES contient maintenant deuxieme.cpp.

Lorsque le fichier projet est créé, il est très facile de générer un fichier Makefile.

Les fichiers Makefiles sont fabriqués à partir des fichiers (.pro) comme ceci:

```
qmake -o Makefile monprog.pro
```

Les utilisateurs de Visual Studio peuvent générer des fichiers .dsp qui pourront être ouvert dans l'environnement de programmation (Voir la section "Développer avec Visual Studio"):

```
qmake -t vcapp -o monprog.dsp monprog.pro
```

Rappelons que le fichier projet est le même quelle que soit la plate-forme de développement.

Le fichier Makefile étant généré, la compilation peut être lancée par:

```
make
```

sous environnement Unix/Linux et:

```
nmake
```

avec Visual Studio.

D'autres usages intéressants de `qmake` sont :

```
qmake -project
```

qui crée un fichier projet portant le même nom que le répertoire courant. Dans ce fichier projet sont ajoutés tous les fichiers .cpp, .h et .ui présents dans le répertoire. Si nous exécutons cette commande dans le répertoire contenant nos quatre fichiers, un fichier projet (hello.pro) va

être créé. Le fichier projet montré plus haut a d'ailleurs été généré par cette commande.

```
qmake
```

Utilisé seul, `qmake` va créer un fichier Makefile à partir du fichier projet.

1.1.2. Qt Designer

`Qt Designer` est un outil pour concevoir et mettre en oeuvre des interfaces utilisateur construites avec Qt. C'est un constructeur d'interfaces, vous pouvez avec lui facilement créer des dialogues et disposer les objets tels qu'ils seront vus à l'exécution. Une interface générée avec `Qt Designer` est enregistrée dans un fichier ayant l'extension (.ui) et contenant du XML. Ce format n'est pas directement exploitable par un compilateur C++. Il est donc nécessaire de transformer notre fichier (.ui) avec un outil qui s'appelle `uic` (User Interface Compiler), il lit un fichier de définition d'interface (.ui) en XML généré par `Qt Designer` et crée le fichier d'entête (.h) et le source (.cpp) correspondants.

`Qt Designer` peut être lancé sous Linux en ouvrant un terminal puis en saisissant `designer`. Il est également possible d'y accéder, comme sous Windows, en choisissant l'option correspondante du menu "Démarrer" (menu K sous Linux Mandrake).

Reportez vous au chapitre **Utiliser Qt Designer** qui détaille son utilisation.

1.1.3. uic

`uic` est le compilateur d'interface utilisateur qui lit un fichier d'interface (.ui) en XML généré par `Qt Designer` et crée un fichier d'en-tête et source C++ correspondants.

Générer le fichier d'en-tête (.h) est effectué comme ceci :

```
uic hello.ui -o hello.h
```

La création de l'implémentation (.cpp) est faite ainsi :

```
uic hello.ui -i hello.h -o hello.cpp
```

Pour les fichiers d'interface (.ui) faits avec `Qt Designer`, il peut être intéressant de créer un fichier d'en-tête et un fichier d'implémentation contenant une classe dérivée de celle créée par `Qt Designer`. Cela peut-être fait avec `uic`. Voici la ligne de commande pour le fichier d'en-tête:

```
uic -subdecl HelloImpl hello.h hello.ui -o
helloimpl.h
```

`HelloImpl` représente le nom de la classe à créer (elle dérivera de la classe `Hello` faite avec `Qt Designer`).

et celle pour le fichier d'implémentation:

```
uic -subimpl HelloImpl helloimpl.h hello.ui
-o helloimpl.cpp
```

Notez que connaître la syntaxe de `uic` à peu d'intérêt. En effet lors de la génération du fichier Makefile par qmake, la compilation des interfaces (.ui) est prévue. Donc tout est automatisé et il y a peu d'occasions de lancer `uic` seul.

1.1.4. moc

`moc` (Meta Object Compiler) c'est à dire le compilateur de méta-objet, est le programme qui manipule les extensions C++ de Qt.

`moc` lit un fichier source C++. S'il trouve une ou plusieurs déclarations de classe qui contiennent la macro Q_OBJECT, il produit un autre fichier source C++ qui contient le code de méta-objet pour les classes qui emploient la macro Q_OBJECT. Entre autres, le code de méta-objet est exigé pour le mécanisme de signal/slot, et le système dynamique de propriété.

Le fichier source C++ produit par `moc` doit être compilé et lié avec

l'implémentation de la classe (il peut être également inclus par #include dans le fichier source de la classe).

`moc` est employé sur des classes qui ressemblent au fichier `helloimpl.h`:

```
1   #ifndef HELLOIMPL_H
2   #define HELLOIMPL_H
3
4   #include <qdialog.h>
5   #include "hello.h"
6
7   class HelloImpl : public Hello
8   {
9       Q_OBJECT
10  public:
11      HelloImpl(
12          QWidget* parent = 0,
13          const char* name = 0);
14  signals:
15      void monSignal();
16
17  public slots:
18      void slotAMoi();
19
20  };
21  #endif
```

Vous remarquez la ligne 9 qui contient la macro `Q_OBJECT`. Également le `signal` de la ligne 15 et le `slot` de la ligne 18. Ces derniers ne pourraient pas être fonctionnels sans compilation avec `moc`.

`signals:` et `public slots:` qui sont spécifiques à Qt seront transformés en code C++ par `moc`.

La déclaration des slots suit les mêmes règles que la déclaration des données membres et méthodes de classe :

- **public slots:** Les slots déclarés avec le mot clé `public` peuvent être appelés par n'importe quel objet ayant instancié la classe.
- **private slots:** Les slots `private` ne sont accessibles que par les membres de la même classe.
- **protected slots:** L'accès est autorisé uniquement pour les membres de la classe et ses descendantes.

`moc` doit être exécuté de la façon suivante :

```
moc helloimpl.h -o moc_helloimpl.cpp
```

Le résultat produit par `moc` doit être compilé et lié, comme tout autre code C++ dans votre programme, autrement la construction échouera dans la phase de compilation. Par convention, cela doit être fait d'une des deux manières suivantes :

1 **La déclaration de classe est trouvée dans un dossier d'entête (.h).**

Si la déclaration de la classe est stockée dans un fichier `helloimpl.h`, le résultat de la compilation par `moc` devrait être stocké dans un fichier appelé `moc_helloimpl.cpp`. Ce fichier sera compilé comme les autres et inclus lors de l'édition de liens.

2 **La déclaration de classe est trouvée dans un fichier d'implémentation (.cpp)**

Si la déclaration de classe est trouvée dans le fichier `helloimpl.cpp`, le résultat de la compilation par `moc` devrait être stocké dans un fichier appelé `helloimpl.moc`. Ce fichier devra être inclus à la fin de l'implémentation par:

```
#include "helloimpl.moc"
```

Notez que la méthode 1 est la méthode normale et que son utilisation

doit être privilégiée. En fait, n'utilisez jamais la deuxième méthode !

Une règle simple à retenir: Toute classe qui utilise des signaux et/ou des slots doit contenir la macro `Q_OBJECT` au début de la déclaration de la classe. Cette classe doit passer au préprocesseur objet `moc`, puis être compilée avec le reste du projet.

Notez que comme pour `uic`, lors de la génération du fichier Makefile par `qmake`, l'appel de `moc` est prévu. Tout est donc automatisé et il y a peu d'occasions de lancer `moc` seul.

Notez que les fichiers générés par `moc` (commençant par moc_) n'ont pas besoin d'apparaître dans le fichier projet (.pro).

1.1.5. Développer avec Visual Studio

Pour bénéficier de la barre d'outils dans Visual Studio, il faut bien sûr que Qt soit installé sur la machine et que Visual C++ soit correctement configuré.

Si la barre n'est pas accessible, choisissez le menu **Tools option Customize** et cochez l'option QMsDev.DSAddln.1 dans l'onglet **Add-ins and Macro Files** :

Redémarrez Visual Studio ou l'ordinateur si nécessaire, vous avez maintenant accès à la barre d'outils de Qt.

La barre d'outis dispose d'icônes dont voici les fonctionnalités:

- **"New Qt Project"** : Permet de créer un nouveau projet Qt dans l'environnement de programmation. Après avoir renseigné le nom du projet, un répertoire du même nom sera créé pour stocker le projet (à moins que vous ne changiez ce répertoire dans le champ Location). Application Type permet de choisir parmi deux modèles d'applications :

  Dialog : Au lancement, l'application affiche une fenêtre simple (class `QDialog`) qui contient éventuellement des widgets et ef-

Figure 1.2. La barre d'outils Qt dans Visual Studio.

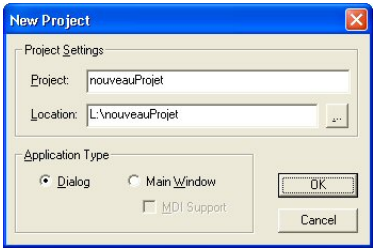


Figure 1.3. La création d'un projet.

fectue des traitements. Un exemple de ce type est la calculette de Windows.

Main Window : Il s'agit d'une fenêtre principale (class `QMainWindow`) qui dispose généralement d'un menu, d'une ou plusieurs barres d'outils, d'une barre d'état et d'un widget central. Elle peut être MDI (multiple document interface), c'est-à-dire qu'elle pourra contenir elle-même une ou plusieurs fenêtres ouvertes en même temps.

Après avoir validé la création, VC++ génère le projet et lance `Qt Designer` en chargeant un dialogue par défaut.

Ce dialogue a été ajouté au projet et les règles de compilation ont été définies. Chaque fois que le fichier d'interface (.ui) sera

modifié, `uic` sera lancé pour produire le fichier d'implémentation (.cpp) et d'en-tête correspondant (.h).

- **"New Qt Dialog"** : Cette icône permet d'ajouter un fichier d'interface au projet en définissant les règles de compilation.

- **"Qt Designer"** : Cette icône permet de lancer `Qt Designer`.

- **"Open Qt Project"** : Cette option doit être utilisée pour convertir un fichier projet Qt (.pro) en fichier projet de Visual C++.

  Un fichier (.dsp) est généré lors de la transformation qui pourra ensuite être ouvert par le menu "File/Open Workspace" puis en choisissant "Projects (.dsp)" dans la liste "Fichiers de type".

  Cette option est particulièrement utile pour importer un projet généré sous Linux. Les fichiers sources, d'en-têtes et surtout d'interfaces sont correctement intégrés et configurés.

- **"Write Qt Project"** : Cette icône permet de créer un fichier projet Qt (.pro) à partir d'un projet de Visual C++. Cette fois, c'est surtout utile pour exporter un projet vers Linux en utilisant les fichiers projets.

- **"Use Qt in current Project"** : Cette icône permet de transformer un projet "normal" en projet Qt. Cela rajoute deux DLL qtmain.dll et qt-mtxxx.lib au projet. La dernière DLL peut avoir un nom différent d'une version à l'autre de Qt. C'est pour cette raison que xxx doit être remplacé par votre numéro de version.

- **"Add MOC"** : Cette option demande au compilateur de lancer, lors de la compilation, l'utilitaire `moc` sur le fichier d'en-tête actuellement affiché dans l'éditeur. C'est indispensable lorsque la classe utilise des signaux et/ou des slots. Pour que la demande de compilation par `moc` soit prise en compte, veillez à bien afficher

le contenu du fichier d'en-tête en double-cliquant sur le nom de la classe dans le TreeView (l'explorateur de classes et de fichiers).

1.1.6. Utiliser la Documentation

La documentation de Qt est, comme on pourrait s'en douter, très bien réalisée.

Qt propose deux manières de consulter la documentation. La première est d'utiliser un navigateur ouvrant des pages au format Html. La deuxième est d'utiliser `Qt Assistant` fourni avec la distribution Qt. C'est un outil permettant de consulter en ligne la documentation de Qt. Toutes les classes et toutes les méthodes sont documentées correctement, avec une foule de détails et d'exemples. `Qt Assistant` présente un contenu identique à la documentation accessible par un navigateur Html avec quelques fonctionnalités supplémentaires. Le module de recherche par index est très puissant et permet de retrouver rapidement une classe ou une propriété.

`Qt Assistant` peut être lancé sous Linux en ouvrant un terminal puis en lançant `assistant`. Il est également possible d'y accéder, comme sous Windows, en choisissant l'option correspondante du menu "Démarrer" (menu K sous Linux Mandrake).

Il n'est pas possible de décrire entièrement la documentation tant elle est riche de contenu. Le lien "All Classes" référence l'ensemble des classes proposées par Qt. Un autre lien intéressant est "Grouped Classes" qui liste les classes groupées par centre d'intérêt.

Lorsque une classe est affichée, deux liens intéressants sont à suivre :

- **More**: Après une courte phrase décrivant la classe. "More" permet d'accéder à une description plus détaillée des fonctions fournies.

- **List of all member functions**: Ce lien fournit la liste complète des propriétés de la classe. En effet, lorsqu'une classe est affichée, seules les nouvelles propriétés sont affichées. C'est-à-dire que celles des classes dont elle hérite sont invisibles. Suivre ce lien

permet d'afficher la totalité des informations de la classe, héritées ou non.

1.1.7. Traducteur

Qt permet de fabriquer des applications gérant plusieurs langues. Ainsi les menus déroulants, les textes affichés ainsi que les boîtes de message peuvent apparaître dans différentes langues.

Pour qu'une chaine de caractères puisse faire l'objet d'une traduction, elle doit être accompagnée de la macro `tr()` dans le source du programme.

Par exemple:

```
QString message = tr("Veuillez entrer un nom");
QPushButton *pb = new QPushButton(
        tr("&Annuler"), this );
```

La traduction d'une application comporte trois phases:

1 Exécution de `lupdate` afin d'extraire du code de l'application les chaînes de caractères à traduire.

2 Utilisation de `Qt Linguist` pour traduire l'application.

3 Exécution de `lrelease` pour générer les fichiers binaires (.qm) que l'application chargera au lancement en utilisant un `QTranslator`.

`Qt Linguist` peut être lancé sous Linux en ouvrant un terminal puis en lançant `linguist`. Il est également possible d'y accéder, comme sous Windows, en choisissant l'option correspondante du menu "Démarrer" (menu K sous Linux Mandrake).

Le fichier projet (.pro) doit contenir le nom des langues à supporter par l'application (Voir la section `qmake` pour plus de détails).

Afin de créer les fichiers (.ts) contenant les textes à traduire, placez vous

dans le répertoire de votre projet. Exécuter ensuite:

```
lupdate -verbose monappli.pro
```

Notez que les fichiers d'interfaces (.ui) sont également traités par la commande.

Vous pouvez lancez `Qt Linguist` puis charger les fichiers générés (.ts) afin de les traduire.

Dans l'application, c'est une instance de la classe `QTranslator` qui charge des fichiers (.qm). Ceux-ci peuvent être générés dans `Qt Linguist` en choisissant le menu File/Release. Il est également possible d'effectuer la génération en ligne de commande en lançant:

```
lrelease -verbose monappli.pro
```

Votre application peut maintenant utiliser des fichiers de traductions en utilisant un `QTranslator`:

```
#include <qapplication.h>
#include "monappli.h"
#include <qtextcodec.h>
#include <qtranslator.h>

int main( int argc, char **argv )
{
        QApplication app( argc, argv );
        QTranslator appTranslator(0);
        appTranslator.load(QString("monappli_")
+
                QTextCodec::locale(),
                qApp->applicationDirPath());
        app.installTranslator(&appTranslator);
        monappli = new MonAppli(0, "MonAppli");
        app.setMainWidget(monappli);

        monappli->show();
        return app.exec();
}
```

1.2. Un premier programme

Nous allons maintenant écrire notre premier programme. Dans une console sous `Linux` ou dans l'explorateur de Windows créez un répertoire. Placez à l'intérieur un fichier nommé par exemple `dial.cpp` contenant le code suivant :

```
1   #include <qapplication.h>
2   #include <qpushbutton.h>
3
4   int main( int argc, char **argv )
5   {
6     QApplication a( argc, argv );
7
8     QPushButton *bouton =
9       new QPushButton("&Quitter", 0);
10
11    a.setMainWidget( bouton );
12    bouton->show();
13    return a.exec();
14  }
```

Les habitués du langage C/C++ reconnaîtront la traditionnelle fonction `main()` qui est le point d'entrée du programme. Avec Qt, qui est du vrai C++ elle est toujours indispensable. La manière de procéder avec Qt, comme nous allons l'étudier, consiste à créer une instance de la classe `QApplication`. Un dialogue ou une fenêtre principale est ensuite instancié puis affiché avec `show()`. Enfin, la boucle d'évènement est installée par le lancement réel de l'application qui s'effectue par `exec()` . Dans une "véritable" application, un dialogue ou une fenêtre principale serait sans doute affiché et les lignes 8 à 12 seraient remplacées par quelque chose comme :

Figure 1.4. Notre premier programme.

```
MaClasseDialogue *dialogue =
        new MaClasseDialogue(0);
a.setMainWidget( dialogue );
dialogue->show();
```

Dans une console placez vous dans le répertoire que vous venez de créer. Lancez ensuite

`qmake -project "&&" qmake "&&" make`.

Vous devriez obtenir un exécutable portant le même nom que le répertoire.

Notez que les utilisateurs de Windows devront lancer ces trois commandes indépendamment et remplacer make par nmake.

Etudions maintenant le contenu de notre fichier:

Les lignes 1 et 2 sont destinées à inclure les fichiers d'en-têtes nécessaires. Chaque fois qu'une classe Qt est utilisée, le fichier en-tête correspondant doit être inclus. Ce dernier porte le même nom que la classe, mais en minuscule. Si vous utilisez dans votre programme une ligne de saisie, c'est le widget `QLineEdit` qui est concerné et vous devrez donc inclure le fichier d'en-tête par: `#include<qlineedit.h>`. Qt installe dans un répertoire, souvent `/usr/lib/qt3/include` sous Linux, les fichiers d'en-tête et chaque classe possède le sien.

La ligne 6 crée un objet `QApplication` pour gérer les ressources de l'application. Notez que le constructeur de `QApplication` exige de recevoir les paramètres `argc` et `argv`. Un programme ne peut avoir qu'une seule instance de cette classe et elle est obligatoire pour les programmes graphiques. Elle doit être créée avant tout autre objet graphique. La classe `QApplication` est toujours accessible dans le programme grâce à un pointeur global nommé `qApp`.

La ligne 8 crée un `QPushButton`. Le premier paramètre, indique quel est le texte à afficher dans le bouton. Le & permet de souligner la lettre qui le suit qui sera alors utilisée comme raccourci clavier. Le paramètre 0, indique que ce dialogue ne possède pas de parent et que par conséquent il est autonome et ne dépend d'aucun autre.

La ligne 11 fait du bouton le widget principal de l'application. Quand l'utilisateur fermera le widget principal, l'application se terminera. Sans widget principal, le programme continuera à tourner en arrière-plan après la fermeture du dialogue.

La ligne 12 affiche le bouton.

La ligne 13 donne le contrôle de l'application à Qt. Le programme passe alors dans l'attente d'un évènement déclenché par l'utilisateur (clavier, souris).

1.3. Gestion des objets en mémoire

La classe `QObject` est la classe de base de tous les objets de Qt utilisant des signaux et/ou des slots.

Tous les objets Qt héritant de `QObject` possèdent un paramètre dans le constructeur appelé `parent`. Les `QObject` s'organisent dans des arbres d'objets. Quand vous créez un `QObject` avec un autre `QObject` comme parent, l'objet parent ajoute l'autre à sa liste d'enfants. Lorsque le parent sera détruit, il supprimera automatiquement ses enfants dans son destructeur. La gestion de la mémoire est, grâce à ce mécanisme grandement facilité et les risques de fuites-mémoire sont moindres.

En résumé, les seuls objets à détruire avec `delete` sont ceux ayant été

créés avec `new` et ne possédant pas de paramètre `parent` renseigné.

1.4. Signaux et Slots

Dans notre premier programme, nous pouvons remarquer que quelque chose ne fonctionne pas comme on pourrait s'y attendre. En effet, alors que le bouton est censé fermer la fenêtre, un clique sur celui-ci n'a aucun effet.

Le dispositif central du modèle objet de Qt est un mécanisme très puissant pour la communication d'objets faiblements couplés appelé des `signaux` et des `slots`.

Faiblement couplé signifie que l'émetteur d'un `signal` ne sait pas quel objet va le prendre en compte (il sera peut être ignoré). De la même façon, un objet interceptant un `signal` ne sait pas quel autre objet a émis le signal. Cette technique permet de relier entre eux des objets de types différents et offre une grande souplesse de développement.

Lorsqu'un objet veut signaler que quelque chose s'est passé le concernant, un bouton qui vient d'être cliqué par exemple, il émet un `signal` . Un `signal` est également émis par un objet lorsque son état a changé. L'objet émetteur ne sait pas comment ce `signal` sera interprété (il ne sera peut être pas pris en compte). L'action à mener, le code à exécuter est contenu dans un `slot`. Les `slots` peuvent être employés pour recevoir des signaux, mais ce sont également des fonctions normales membres de classes. Comme nous le verrons ultérieurement, il nous suffira de connecter (associer) le `signal` au `slot`. Le programme exécutera alors le code de la méthode définie comme `slot` lorsque le `signal` d'un objet sera émis. Par exemple, pour un bouton, le `signal` le plus utilisé est sans doute `clicked()` qui est émis lorsque l'utilisateur clique sur un bouton. Connecter un `signal` à un `slot` consiste à indiquer à Qt quel `slot` déclencher lorsqu'un `signal` est émis.

Voici en exemple la définition d'une classe contenant des signaux et des slots:

```
class Personne : public QObject
```

```
{
Q_OBJECT
public:
        Personne(
                QObject *parent=0,
                char *name=0 );

        int agePersonne() const { return age; }
public slots:
        void setAge( int );
signals:
        void ageChange( int );
private:
        int age;
};
```

Le fichier d'implémentation pourrait ressembler à ceci:

```
#include <qobject.h>
#include "personne.h"

Personne::Personne(
        QObject *parent,
        char *name )
        : QObject( parent, name )
{
}

Personne::setAge(int a)
{
        if ( a != age )
        {
                age = a;
                emit ageChange(a);
               }
}
```

Le `slot` `setAge` peut-être relié au `signal` d'un autre objet ou appelé comme une méthode normale. Lui-même émet un `signal` ayant comme paramètre un entier.

La connexion s'effectue grâce à la méthode `QObject::connect`. Modifiez notre premier programme montré au début du chapitre en rajoutant la ligne 10 qui va permettre d'assurer la connexion. La fonction connect attend deux paires de paramètres. Tout d'abord l'objet émetteur du `signal` ainsi que le `signal` à surveiller. Ensuite, l'objet récepteur ainsi que le `slot` qui doit être déclenché.

```
1       #include <qapplication.h>
2       #include <qpushbutton.h>
3
4       int main( int argc, char **argv )
5       {
6         QApplication a( argc, argv );
7
8         QPushButton *bouton =
9               new QPushButton("&Quitter", 0);
10        QObject::connect(
11              bouton, SIGNAL(clicked()),
12              &a, SLOT(quit()));
13        a.setMainWidget( bouton );
14        bouton->show();
15        return a.exec();
16      }
```

Nous pouvons à présent recompiler notre programme en lançant `make`. Inutile d'exécuter `qmake` puisque qu'aucun fichier n'a été ajouté.

Maintenant, lorsque l'utilisateur clique sur le bouton ou appuie sur la barre espace la fenêtre est fermée comme on s'y attend.

Le `signal` et le `slot` sont entourés respectivement par `SIGNAL()` et `SLOT()`. Ce sont deux macros qui sont présentes afin d'éviter des

saisies fastidieuses et seront transformées en C++ valide par l'utilitaire `moc` lors de la phase de compilation.

Nous avons connecté le `signal clicked()` au `slot quit()` de l'objet `QApplication`. Ce `slot` est bien entendu déjà programmé dans Qt et il suffit de l'appeler pour que la fenêtre se ferme. Nous verrons plus tard comment déclarer nos propres `slot` afin d'exécuter des traitements dans les programmes.

Un objet ne sait pas si quelque chose reçoit ses signaux et un `slot` ne sait pas s'il a des signaux qui lui sont reliés. Ceci permet de créer avec Qt des composants véritablement indépendants.

Vous pouvez relier autant de signaux que vous voulez à un simple slot, et un `signal` peut être relié à autant de `slot` que vous le désirez. Il est même possible de relier un `signal` directement à un autre signal. (Celui-ci émettra le deuxième `signal` immédiatement toutes les fois que le premier sera émis.)

Ensemble, les signaux et les slots composent un mécanisme de programmation composant puissant.

- **Un signal peut être connecté à plusieurs slots:**

```
connect(bouton, SIGNAL(clicked()),
        this, SLOT(slotMultiplie()));
connect(bouton, SIGNAL(clicked()),
        this, SLOT(slotFerme()));
```

  Lorsque le `signal` est émis, les slots sont appelés les uns après les autres dans un ordre qui peut-être différent de la déclaration.

- **Plusieurs signaux peuvent être connectés à un même slot:**

```
connect(bouton, SIGNAL(clicked()),
        this, SLOT(slotMAJ()));
connect(
        MaListBox,
        SIGNAL(selectionChanged()),
        this,
        SLOT(slotMAJ()));
```

Lorsque l'un des signaux est émis, le `slot` est exécuté. Si chaque objet émet son signal, le `slot` est exécuté deux fois.

- **Un signal peut être connecté à un autre signal:**

```
connect(bouton, SIGNAL(clicked()),
        autreBouton, SIGNAL(clicked()));
```

Lorsque le premier `signal` est émis, le deuxième l'est également.

- **Les connexions peuvent être supprimées:**

```
disconnect(bouton, SIGNAL(clicked()),
        this, SIGNAL(slotClic()));
```

C'est rarement nécessaire, car Qt détruit les connexions d'un objet lorsque celui-ci est supprimé.

Un `signal` et un `slot` connectés doivent avoir les mêmes types de paramètres. Ainsi :

```
connect(maClasse, SIGNAL(monSignal(int)),
        unSlot, SLOT(slotPerso(QString)));
```

est illégal et générera une erreur lors de l'exécution.

Valeurs dans le connect ?

Une fausse idée répandue est qu'il est possible de définir les valeurs à

envoyer avec un `signal` lors de la connexion, par exemple:

```
connect( &a, SIGNAL(ageChange(40)),
        &b, SLOT(majAgePersonne(int)) );
```

Ce n'est pas possible. Seulement les signatures des signaux sont utilisées dans l'appel de connexion. En émettant le signal, un paramètre peut être fourni, et seulement alors. La version correcte du code précédent serait :

```
connect( &a, SIGNAL(ageChange(int)),
        &b, SLOT(majAgePersonne(int)) );
```

et la valeur (40) serait indiquée en émettant le signal:

```
emit ageChange(40);
```

Résumé

Nous avons présenté dans ce chapitre les outils proposés par Qt. Un premier programme a été réalisé puis la gestion-mémoire de Qt a été étudiée. Cette gestion permet d'éviter les fuites-mémoires en permettant une suppression automatisée des objets alloués dans une classe. Enfin, les signaux et les slots ainsi que leurs utilisations ont été vus.

2. Dialogues

Ce chapitre présente la façon de créer des dialogues avec Qt. Un dialogue est une fenêtre qui peut contenir divers widgets : boutons, cases à cocher, listes déroulantes, champs de saisie et bien d'autres encore. Une application un peu importante contient généralement une fenêtre principale à partir de laquelle il est possible d'appeler des dialogues permettant de modifier les données gérées par le programme.

2.1. Sous-classer QDialog

`QDialog` est la classe de base des dialogues fenêtrés. Nous allons étudier dans cette partie la création d'un formulaire de création de signature héritant d'un `QDialog`. Cette petite application va être écrite entièrement à la main. C'est-à-dire que le concepteur d'interfaces, `Qt Designer` ne sera pas utilisé. Il sera étudié ultérieurement et nous verrons qu'avec lui il est plus facile et plus rapide de créer des interfaces.

Le code source est réparti dans deux fichiers: `signature.h` et `signature.cpp`. Voici `signature.h`

```
1    #ifndef SIGNATURE_H
2    #define SIGNATURE_H
3    #include <qdialog.h>
4    class QVBoxLayout;
5    class QHBoxLayout;
6    class QGridLayout;
7    class QGroupBox;
8    class QLabel;
9    class QLineEdit;
10   class QTextEdit;
```

Figure 2.1. Notre formulaire de signature.

```
11    class QPushButton;
```

Les deux premières lignes, associées à la ligne 38 permet de se prémunir contre les inclusions multiples du fichier d'en-tête.

Ce que nous faisons ensuite en ligne 3 est d'inclure la définition de la classe (`QDialog`) dont notre classe `Signature` dérive. Toutes les autres classes utilisées (`QPushButton`, `QLineEdit` etc.) sont référencées comme pointeurs, ainsi la seule chose que le compilateur doit savoir c'est que ce sont des classes (puisque les pointeurs ont la même taille en mémoire indépendamment de ce qu'ils pointent). Donc, au lieu d'inclure les déclarations de classe, une déclaration anticipée est utilisée lignes 4 à 11. Cela réduit le nombre de fichiers d'en-tête à compiler et donc ainsi le temps de compilation. Dès que nous aurons l'intention d'employer des membres des classes, leurs fichiers d'en-têtes devront être inclus dans le fichier d'implémentation de notre classe

(`signature.cpp`).

```
15   class Signature : public QDialog
16   {
17       Q_OBJECT
18   public:
19       Signature( QWidget* parent, const char*
name = 0);
```

Ensuite nous déclarons notre classe `Signature` qui hérite de `QDialog`. La macro `Q_OBJECT` présente au début de la déclaration de la classe est indispensable pour les classes qui définissent des `signaux` et des `slots`. Sans sa présence, vous obtiendriez une erreur de compilation similaire à celle-ci:

```
signature.h:35: Error: The declaration of the class
"Signature" contains signals or slots but no Q_OBJECT
macro.
```

```
21   private slots:
22       void slotCreer();
```

Nous déclarons un `slot` privé. Notez que `slots` qui est un mot clé spécifique à Qt sera interprété par `moc` afin de rendre le fichier d'en-tête compilable.

```
23   private:
24       QGroupBox* groupBox1;
25       QLabel* labelNom;
26       QLabel* labelPrenom;
27       QLabel* labelEmail;
28       QLineEdit* nom;
29       QLineEdit* prenom;
30       QLineEdit* email;
31       QLabel* labelSignature;
32       QTextEdit* textSignature;
33       QPushButton* creer;
34       QPushButton* fermer;
```

```
35    };
36    #endif // SIGNATURE_H
```

Les lignes 23 à 36 permettent de déclarer des objets privés à la classe. Ce sont des pointeurs vers des objets qui sont déclarés.

Voici maintenant dans le détail le fichier d'implémentation `signature.cpp` :

```
1     #include "signature.h"
2     #include <qpushbutton.h>
3     #include <qtextedit.h>
4     #include <qlabel.h>
5     #include <qgroupbox.h>
6     #include <qlineedit.h>
7     #include <qlayout.h>
8     #include <qstring.h>
9     #include <qmessagebox.h>
```

Notre code manipule des types d'objets dont les fichiers d'en-têtes doivent être inclus. C'est fait comme on le voit au début du fichier d'implémentation. Mais nous aurions pu le faire au début du fichier de déclaration de la classe (`signature.h`) au lieu d'effectuer des déclarations anticipées comme indiqué plus haut. En ligne 1 nous incluons la déclaration de notre classe `Signature` qui va être instanciée.

```
8     Signature::Signature(
9           QWidget* parent,
10          const char* name)
11        : QDialog( parent, name)
```

Le constructeur de la classe `Signature` reçoit deux paramètres, `parent` et `name` qui sont transmis au constructeur de la classe de base `QDialog` appelé. Les widgets héritent de `QWidget`, et lui même de `QObject`. Tous les objets héritant de `QObject` possèdent un paramètre `parent` dans le constructeur. Ainsi lorsque l'objet est construit, son parent met à jour sa liste d'objets enfants. Quand ce parent sera détruit, il supprimera automatiquement ses enfants. `setCaption` modifie le

nom affiché dans la barre de titre du dialogue. La macro `tr()` permet de gérer facilement plusieurs langues dans l'application (voir la section "Traducteur").

```
16   {
17     setCaption( tr("Signature") );
18
19     QGroupBox* details =
20        new QGroupBox("Details", this);
21     nom = new QLineEdit( details );
22     prenom = new QLineEdit( details );
23     email = new QLineEdit( details );
24     textSignature = new QTextEdit( this);
25     textSignature->setReadOnly( true );
26     fermer = new QPushButton(
27        "&Fermer", this);
28     creer = new QPushButton(
29        "&Creer", this );
30     labelNom = new QLabel(
31        "Nom", details);
32     labelPrenom = new QLabel(
33        "Prenom", details);
34     labelEmail = new QLabel(
35        "Email", details);
36     labelSignature = new QLabel(
37        "Signature", this);
```

Différents widgets sont créés, remarquez qu'ils renseignent également le paramètre `parent` en passant `this` qui est un pointeur sur l'instance de classe courante. Le widget `textSignature` est mis en lecture seule afin d'empêcher la saisie.

```
39     QGridLayout* SignatureLayout =
40        new QGridLayout( this, 1, 1, 11, 6);
41     QHBoxLayout* layoutBoutons =
42        new QHBoxLayout( 0, 0, 6);
```

```
43      layoutBoutons->addWidget( creer );
44      QSpacerItem* spacer =
45          new QSpacerItem( 151, 20,
46          QSizePolicy::Expanding);
47      layoutBoutons->addItem( spacer );
48      layoutBoutons->addWidget( fermer );
49      SignatureLayout->addLayout(
50           layoutBoutons, 3, 0 );
51      SignatureLayout->addWidget(
52           textSignature, 2, 0 );
53      SignatureLayout->addWidget(
54           labelSignature, 1, 0 );
55      details->setColumnLayout
56          (0, Qt::Vertical );
57      details->layout()->
58          setSpacing( 6 );
59      details->layout()->
60          setMargin( 11 );
61      QGridLayout* detailsLayout =
62          new QGridLayout( details->layout() );
63      detailsLayout->
64          setAlignment( Qt::AlignTop );
65      detailsLayout->
66          addWidget( labelNom, 0, 0 );
67      detailsLayout->
68          addWidget( labelPrenom, 1, 0 );
69      detailsLayout->
70          addWidget( labelEmail, 2, 0 );
71      detailsLayout->
72          addWidget( nom, 0, 1 );
73      detailsLayout->
74          addWidget( prenom, 1, 1 );
75      detailsLayout->
76          addWidget( email, 2, 1 );
77      SignatureLayout->
```

```
78          addWidget( details, 0, 0 );
```

Les widgets sont intégrés dans des `layouts`. Reportez-vous à la section **Disposer les widgets dans les dialogues** pour connaître leurs utilisations en détail.

Comme nous pouvons le constater si nous exécutons le programme, les layouts fournissent une méthode très puissante pour disposer les objets graphiques. Notamment parce que ceux-ci sont repositionnés lorsque la fenêtre est redimensionnée. Néanmoins, saisir le code correspondant demande une certaine gymnastique d'esprit afin de matérialiser les dispositions. Utiliser `Qt Designer` pour créer des dialogues est dans ce cas bien plus efficace. En effet, il permet de disposer graphiquement les layouts au même titre que les widgets. L'utilisateur peut même tester dans l'éditeur le comportement de sa fenêtre avant même de l'enregistrer. Tout ceci est expliqué dans le détail au chapitre **Utiliser Qt Designer**.

```
79      connect( fermer, SIGNAL( clicked() ),
80          this, SLOT( close() ) );
81      connect( creer, SIGNAL( clicked() ),
82          this, SLOT( slotCreer() ) );
83    }
```

Le `signal clicked()` du `QPushButton fermer` est connecté au `slot close()` de la classe de base `QDialog` .

Le `signal clicked()` du bouton `creer` est connecté au `slot slotCreer()` qui appartient à notre classe.

```
85    void Signature::slotCreer()
86    {
87      if ( nom->text().isEmpty() ||
88           email->text().isEmpty() )
89      {
90        QMessageBox::information
91        ( this, "Signature",
92        "Veuillez renseigner un nom\n"
```

```
93        "ainsi qu'une adresse de courrier." );
94        return;
95      }
96      QString message;
97      message = prenom->text()
98                + " "
99                + nom->text()
100                     + " peut etre contacte
a "
101                     + "l'adresse "
102                     + email->text();
103           textSignature->setText(
104             message);
105        }
```

Les `slots` sont déclarés et implémentés comme n'importe qu'elle autre méthode C++ comme le montre le code ci-dessus. Ce sont des méthodes qui peuvent être appelées de manière classique. Le fait de les déclarer comme `slot` permet en plus de les utiliser dans le mécanisme signaux/slots de Qt.

Dans le `slot slotCreer()`, nous vérifions tout d'abord si les champs de saisie `nom` et `email` possèdent un contenu. Dans le cas contraire, un message d'avertissement est affiché puis la fonction se termine (voir la section `QMessageBox`).

Un `QString` est ensuite utilisé pour fabriquer le message de signature. Ce message est ensuite affecté dans le `QTextEdit textSignature` à l'aide de la fonction `setText()`.

Nous avons également besoin d'un fichier `main.cpp`:

```
#include <qapplication.h>
#include "signature.h"

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

```
    Signature *signature = new Signature(0);

    a.setMainWidget( signature );
    signature->show();
    return a.exec();
}
```

Pensez à exécuter dans le répertoire du projet les commandes

```
qmake -project && qmake && make
```

pour créer un fichier projet puis un fichier Makefile et enfin de compiler le projet.

Vous aurez remarqué que notre formulaire est assez simpliste. Il est juste présent pour l'exemple et nécessiterait sans doute beaucoup d'améliorations pour être utile.

2.2. Communication entre dialogues

Ce nouvel exemple utilise deux dialogues, le premier, `Dialogue` possède un champ de saisie, un spinbox, un label et deux boutons. Le deuxième dialogue, `AutreDialogue` appelé par le premier contient pour sa part deux boutons et un champ de saisie. La déclaration de la classe `Dialogue` est enregistrée dans `dialogue.h`

```
1    #ifndef DIALOGUE_H
2    #define DIALOGUE_H
3
4    #include <qdialog.h>
5
6    class QPushButton;
7    class QLineEdit;
8    class QLabel;
9    class QSpinBox;
10   class AutreDialogue;
```

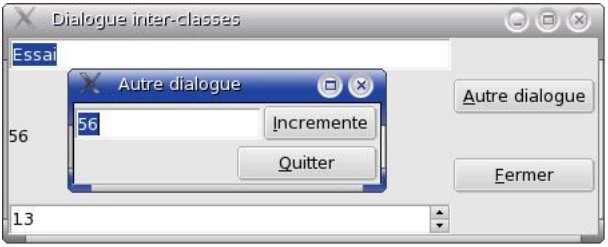


Figure 2.2. La communication entre dialogues.

```
11
12   class Dialogue : public QDialog
13   {
14         Q_OBJECT
15   public:
16         Dialogue(QWidget *parent,
17               const char *name = 0);
18   private slots:
19         void slotOuvreDialogue();
20         void slotIncremente();
21   public slots:
22         void slotMiseAJour(
23               const QString &texte);
24   private:
25         QPushButton *fermer;
26         QPushButton *ouvreDial;
27         QLineEdit *saisie;
28         QLabel *label;
29         QSpinBox *compteur;
30         AutreDialogue *autreDialogue;
31   };
32
```

```
33    #endif
34
```

Notre classe `Dialogue` déclare le `slot slotMiseAJour()` comme `public slots:`, ce `slot` sera donc accessible aux classes externes à `Dialogue`. Au même titre que pour les widgets, un pointeur vers la classe `AutreDialogue` est présent.

Voici maintenant l'implémentation présente dans le fichier `dialogue.cpp`.

```
1     #include <qpushbutton.h>
2     #include <qlayout.h>
3     #include <qlineedit.h>
4     #include <qlabel.h>
5     #include <qspinbox.h>
6     #include "dialogue.h"
7     #include "autreDialogue.h"
8     Dialogue::Dialogue(QWidget *parent,
9           const char *name)
10          : QDialog(parent, name)
11    {
12          setCaption(
13                tr("Dialogue inter-classes"));
14
15          saisie = new QLineEdit(this);
16          connect(
17                saisie,
18                SIGNAL(textChanged(
19                      const QString & )),
20                this,
21                SLOT(slotMiseAJour(
22                      const QString & )) );
```

Le `QLineEdit saisie` a son `signal textChanged(const QString &)` connecté au `slot slotMiseAJour(const QString &)`. Lors de l'exécution, dès que le texte du champ éditable sera modifié,

le `slot` sera appelé.

```
26          label = new QLabel(this);
27
28          compteur =
29               new QSpinBox(this);
30          compteur->setValue(0);
31
32          ouvreDial =
33               new QPushButton(
34               "&Autre dialogue", this);
35          connect(ouvreDial,
36               SIGNAL(clicked()),
37               this,
38               SLOT(slotOuvreDialogue()) );
39          fermer =
40               new QPushButton(
41               "&Fermer", this);
42          connect(fermer,
43               SIGNAL(clicked()),
44               this,
45               SLOT(close()) );
```

Un `QSpinBox` est ici alloué, il s'agit d'un widget muni d'un affichage et de deux flèches permettant d'augmenter et diminuer la valeur affichée. Ce compteur est mis à zéro avec `setValue(0)`.

```
48
49          QVBoxLayout *v1 =
50               new QVBoxLayout;
51          v1->addWidget(saisie);
52          v1->addWidget(label);
53          v1->addWidget(compteur);
54
55          QVBoxLayout *v2 =
56               new QVBoxLayout;
```

```
57          v2->addWidget(ouvreDial);
58          v2->addWidget(fermer);
59
60          QHBoxLayout *principale =
61               new QHBoxLayout(this);
62          principale->addLayout(v1);
63          principale->addLayout(v2);
```

Les widgets sont positionnés dans la fenêtre grâce aux layouts que nous connaissons maintenant bien.

```
65          autreDialogue =
66               new AutreDialogue(this);
67
68          connect(
69          autreDialogue,
70          SIGNAL(plus()),
71          this,
72          SLOT(slotIncremente()) );
73     }
```

Cette partie du code est assez intéressante. Nous allouons une instance de la classe `AutreDialogue` dont l'adresse est sauvegardée dans le pointeur `autreDialogue`. Ensuite nous connectons le `signal plus()` de cette classe au `slot slotIncremente()` de notre classe `Dialogue`. Il suffira ensuite que le `signal` soit émis pour que le `slot` soit appelé.

```
74     void Dialogue::slotOuvreDialogue()
75     {
76
77          autreDialogue->exec();
78     }
```

Le `slot slotOuvreDialogue()` est chargé d'afficher le dialogue alloué dans le constructeur ligne 65.

```
81   void Dialogue::slotMiseAJour
82        (const QString &texte)
83   {
84        label->setText( texte );
85   }
```

Le `slot slotMiseAjour(const QString &texte)` est connecté au `signal textChanged(const QString &texte)` du `QLineEdit`. Mais ce `slot` est public et nous verrons dans le code de `autreDialogue.cpp` qu'il est également connecté au `signal` d'un autre widget.

```
87   void Dialogue::slotIncremente()
88   {
89        compteur->setValue(
90             compteur->value()+1 );
91   }
```

Enfin le `slot slotIncremente()` se charge d'augmenter la valeur du `QSpinBox` et sera déclenché par l'émission du `signal plus()` de la classe `AutreDialogue`.

Voici la déclaration de la classe `AutreDialogue` présente dans `autreDialogue.h`.

```
1    #ifndef AUTREDIALOGUE_H
2    #define AUTREDIALOGUE_H
3
4    #include <qdialog.h>
5
6    class QPushButton;
7    class QLineEdit;
8
9    class AutreDialogue : public QDialog
10   {
11        Q_OBJECT
12   public:
```

```
13        AutreDialogue(
14             QWidget *parent,
15             const char *name = 0);
16   signals:
17        void plus();
18   private:
19        QPushButton *quitter;
20        QPushButton *incremente;
21        QLineEdit *saisie;
22   };
```

La classe `AutreDialogue` comme la précédente hérite de `QDialog`. Elle déclare le `signal` `plus()` en ligne 16 et 17. `signal` qui est spécifique à Qt devra être passé à `moc` afin d'être compris par le compilateur. Comme la macro `Q_OBJECT` est présente, cela ne posera pas de problème.

Et voici pour terminer le fichier `autreDialogue.cpp` qui contient l'implémentation de la classe `AutreDialogue`:

```
1    #include <qpushbutton.h>
2    #include <qlayout.h>
3    #include <qlineedit.h>
4
5    #include "autreDialogue.h"
6
7    AutreDialogue::AutreDialogue(
8         QWidget *parent,
9         const char *name)
10        : QDialog(parent, name)
11   {
12        setCaption(tr("Autre dialogue"));
13
14        saisie = new QLineEdit(this);
15        connect(
16             saisie,
17             SIGNAL(textChanged(
```

```
18              const QString & )),
19              parent,
20              SLOT(slotMiseAJour(
21              const QString & )) );
```

Le `signal` du `QLinedit` est connecté au `slot` `slotMiseAJour(const QString & )` du parent, c'est-à-dire à l'instance de la classe `Dialogue`. Ce `slot` à été déclaré `public slots:` ce qui nous donne la possibilité d'y faire référence.

```
22
23          quitter =
24               new QPushButton(
25               "&Quitter", this);
26          connect(quitter, SIGNAL(clicked()),
27               this, SLOT(close()) );
28
29          incremente =
30               new QPushButton(
31               "&Incremente", this);
32          connect(incremente,
33               SIGNAL(clicked()),
34               this,
35               SIGNAL(plus()) );
```

Le `signal` `clicked()` du `QPushButton incremente` est connecté au `signal` `plus()` déclaré dans cette même classe. C'est-à-dire que lorsque le `signal` du bouton sera émis, le `signal` `plus()` sera également émis. Ce `signal` a été connecté dans l'implémentation de la classe `Dialogue` en ligne 68. Cela démontre qu'une classe peut déclarer ses propres signaux au même titre que les objets fournis par Qt.

```
36          QHBoxLayout *h1 =
37               new QHBoxLayout;
38          h1->addWidget(saisie);
39          h1->addWidget(incremente);
```

```
40
41          QHBoxLayout *h2 =
42                  new QHBoxLayout;
43          QSpacerItem *spacer =
44                  new QSpacerItem(120, 21);
45          h2->addItem(spacer);
46          h2->addWidget(quitter);
47
48          QVBoxLayout *principale =
49                  new QVBoxLayout(this);
50          principale->addLayout(h1);
51          principale->addLayout(h2);
52      }
```

Enfin, nos amis les layouts se chargent de disposer harmonieusement les widgets dans la fenêtre. Remarquez l'apparition d'un nouvel objet, `QSpacerItem` en lignes 43 et 44. Cet objet permet d'insérer des espaces blancs à l'intérieur des layouts. Notre bouton, au lieu d'occuper toute la largeur de la fenêtre, est maintenant poussé à droite de celle-ci par le spacer.

2.3. QFileDialog

Qt dispose de la classe `QFileDialog` permettant de parcourir le système de fichiers afin de sélectionner un fichier ou un répertoire.

```
QString nom =
    QFileDialog::
        getOpenFileName
        (
                QString::null,
                "Texte (*.txt);;"
                "Tous (*.*)",
                this
        );
        if (
```

```
        nom.isEmpty() )
                return; // Selection annulee
```

Cette classe possède des méthodes statiques permettant de choisir un fichier à lire (`getOpenFileName`), ou à écrire (`getSaveFileName`). La première méthode ne renverra un nom que si le fichier existe alors que la seconde permet de désigner le nom d'un fichier n'existant pas encore sur disque. Il existe également la méthode `getOpenFileNames` permettant de sélectionner plusieurs fichiers et qui renvoie un `QStringList` contenant le ou les fichiers sélectionnés. La méthode `getExistingDirectory` permet de sélectionner un nom de répertoire existant et renvoie un `QString`. La méthode `getOpenFileName` utilisée dans l'exemple renvoie un `QString`. Celui-ci contient le nom du fichier choisi ou un `QString` vide si la sélection a été annulée. Cette méthode accepte plusieurs paramètres: le premier qui n'a pas été renseigné permet d'indiquer le répertoire dans lequel s'ouvre le dialogue. Le deuxième indique le filtre à appliquer au fichiers affichés.

Le code vérifie si `nom` possède un contenu avant de continuer le traitement.

2.4. QMessageBox

La classe `QMessageBox` fournit un dialogue modal avec un message court, une icône, et quelques boutons.

Un dialogue est modal lorsqu'il est bloquant jusqu'à ce que l'utilisateur le ferme. L'utilisateur ne peut pas agir sur d'autres fenêtres dans la même application tant que le dialogue reste ouvert.

`QMessageBox` propose une gamme de différents messages, répartis en quatre niveaux:

 **Question**: Pour les boîtes de message qui posent une question en tant qu'opération normale du programme.

```
    if ( QFile::exists( nomFichier ) &&
```

```
            QMessageBox::question(
            this,
            tr("Ecraser le fichier?"),
            tr("Un fichier %1 existe deja."
            "Voulez vous l'ecraser?")
            .arg( nomFichier ),
            tr("&Oui"), tr("&Non"),
            QString::null, 0, 1 ) )
    return false;
```

 **Information**: Pour les boîtes de message qui font partie d'opérations normales.

```
    QMessageBox::information( this, "Mon
    programme",
            "Impossible de trouver la recherche.\n"
            "Veuillez preciser vos criteres." );
```

Warning: Pour les boîtes de message qui indiquent à l'utilisateur des erreurs inhabituelles.

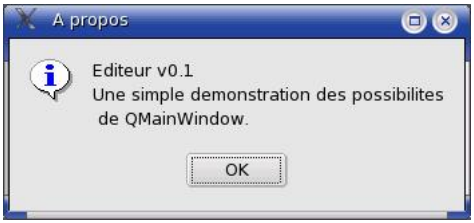
```
QMessageBox::warning( this, "Mon Application",
        "Impossible de se connecter au
serveur.\n"
        "Ce programme ne peut fonctionner
correctement "
        "sans le serveur.\n\n",
        "Retenter",
        "Quitter", 0, 0, 1 );
```

Critical: Pour les boîtes de message qui indiquent à l'utilisateur des erreurs critiques.

```
QMessageBox::critical( this, "Mon Application",
        QString("Une erreur interne vient de se
produire.\n") +
        "Veuillez contacter le support
technique\n"+
        "et communiquer l'erreur numero 12.\n"+
        "L'application va maintenant se
terminer.");
```

La boîte de message a une icône différente pour chacun des niveaux précédents. Il y a des fonctions statiques pour les cas les plus communs. `QMessageBox::about()` permet d'afficher des informations relatives au programme avec une icône appropriée.

```
QMessageBox::about(
        this,
        tr("A propos"),
        tr("Mon programme v0.1\n"
        "Une simple demonstration"
        " des possibilites\n de "
        "QMainWindow.")
        );
```



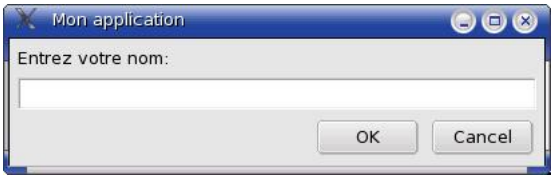
2.5. QInputDialog

La classe `QInputDialog` fournit un dialogue permettant de

récupérer une valeur saisie par l'utilisateur.

Quatre fonctions statiques sont fournies : `getText()`, `getInteger()`, `getDouble()` and `getItem()`. Toutes ces fonctions s'utilisent d'une façon similaire, par exemple:

```
bool ok;
    QString text = QInputDialog::getText(
            "Mon application", "Entrez votre
nom:", QLineEdit::Normal,
            QString::null, &ok, this );
    if ( ok && !text.isEmpty() ) {
        // l'utilisateur a saisi quelque chose et
a enfonce OK
    } else {
        // l'utilisateur n'a rien saisi ou a
enfonce Annuler.
    }
```



Résumé

Ce chapitre nous a permis d'étudier la création des fenêtres héritées de la classe `QDialog`. La communication entre deux dialogues a été également vue. Enfin, nous avons passé en revue quelques-uns des dialogues prédéfinis de Qt.

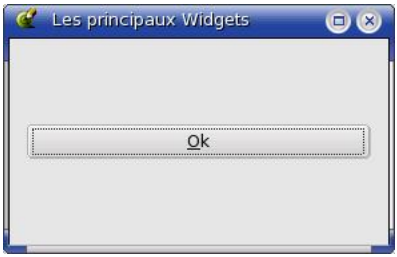
3. Les principaux Widgets

Dans ce chapitre sont présentés les principaux widgets de Qt. Il s'agit de ceux qui sont employés le plus souvent dans les dialogues.

Après une courte description, une figure présente le widget utilisé au sein d'une fenêtre. Le lecteur se reportera si nécessaire à la documentation fournie par Trolltech afin de connaître les détails de leurs utilisations et la façon de les créer.

3.1. QPushButton

Le widget `QPushButton` fournit un bouton de commande.



Le bouton-poussoir, ou le bouton de commande, est peut-être le widget le plus généralement utilisé dans toute interface utilisateur graphique. Enfoncez (cliquez) un bouton pour commander l'ordinateur afin d'effectuer une certaine action, ou pour répondre à une question. Les boutons typiques sont OK, Appliquer, Annuler, Fermer, Oui, Non et Aide.

Un bouton de commande est rectangulaire et affiche typiquement un texte décrivant son action. Un caractère souligné dans le texte (signi-

fié en le précédant avec un & dans le texte) indique un accélérateur, par exemple.

```
QPushButton *pb = new QPushButton( "&Ok", this );
```

Dans cet exemple l'accélérateur est Alt+O, et le texte affiché sera Ok.

Les boutons de commande peuvent afficher une étiquette textuelle ou un pixmap, et optionnellement une petite icône. Ceux-ci peuvent être spécifiés dans le constructeur et modifiés ultérieurement en utilisant `setText()`, `setPixmap()` et `setIconSet()`. Si le bouton est désactivé, l'apparence du texte ou pixmap et iconset sera modifiée dans le respect du style graphique afin d'afficher un bouton à l'aspect désactivé.

Un bouton émet le `signal clicked()` quand il est activé par la souris, la barre d'espace ou par un accélérateur du clavier. Connectez un `slot` à ce `signal` pour effectuer l'action du bouton. Les boutons fournissent également des signaux moins généralement utilisés, par exemple, `pressed()` et `released()`.

3.2. QRadioButton

Le widget `QRadioButton` fournit un bouton radio avec un texte ou un pixmap.

`QRadioButton` et `QCheckBox` sont tous deux des boutons d'options. C'est-à-dire qu'ils peuvent être sélectionnés (cochés) ou désélectionnés (décochés). Les classes diffèrent par la restriction des choix proposés à l'utilisateur. Les cases à cocher permettent de sélectionner plusieurs d'entre elles en même temps alors que les boutons radios ne permettent

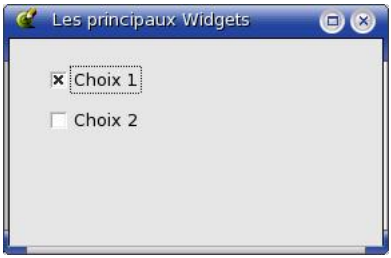
la sélection que de l'un d'entre eux à la fois. Dans un groupe de boutons radios, seulement un seul d'entre eux peut être sélectionné en même temps; si l'utilisateur sélectionne un autre bouton, le bouton précédemment coché est désélectionné.

Lorqu'un bouton est coché ou décoché, il émet le `signal toggled()`. Connectez un `slot` à ce `signal` si vous désirez déclencher une action chaque fois que l'état du bouton change. Autrement, l'utilisez `isChecked()` pour savoir si un bouton particulier est sélectionné.

Comme un `QPushButton`, un bouton radio peut afficher un texte ou un pixmap. Le texte peut-être défini dans le constructeur ou avec `setText()`; le pixmap peut être défini avec setPixmap().

3.3. QCheckBox

Le widget `QCheckBox` fournit une case à cocher avec un texte.



`QRadioButton` et `QCheckBox` sont tous deux des boutons d'options. C'est-à-dire qu'ils peuvent être sélectionnés (cochés) ou désélectionnés (décochés). Les classes diffèrent par la restriction des choix proposés à l'utilisateur. Les cases à cocher permettent de sélectionner plusieurs d'entre elles en même temps alors que les boutons radios ne permettent la sélection que l'un d'entre eux à la fois.

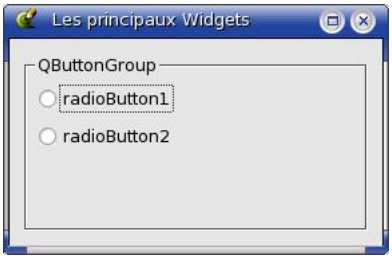
Un `QButtonGroup` peut être utilisé pour grouper visuellement les cases à cocher.

Lorqu'une case est cochée ou décochée, elle émet le `signal toggled()`. Connectez un `slot` à ce `signal` si vous désirez déclencher une action chaque fois que l'état de la case à cocher change.

Autrement, utilisez `isChecked()` pour savoir si une case à cocher particulière est sélectionnée.

3.4. QButtonGroup

Le widget `QButtonGroup` organise les widgets `QButton` dans un groupe.

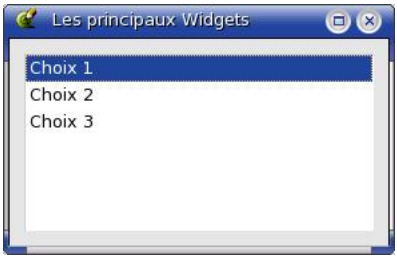


Un widget `QButtonGroup` rend facile le travail avec les boutons groupés. Chaque bouton dans un groupe de boutons possède un identifiant unique. Le groupe de boutons émet le `signal clicked()` avec cet identifiant quand un bouton dans le groupe est cliqué. Cela rend le groupe de boutons particulièrement utile quand vous avez plusieurs boutons semblables et que vous voulez relier le `signal clicked()` de tous ces boutons à un unique `slot`.

Un groupe exclusif (positionné avec `setExclusive()`) bascule tous les boutons enfoncés excepté celui qui vient d'être cliqué. Un groupe de boutons est, par défaut, non exclusif. Notez que tous les boutons radios insérés dans un groupe de boutons sont mutuellement exclusifs même si le groupe de boutons est non exclusif.

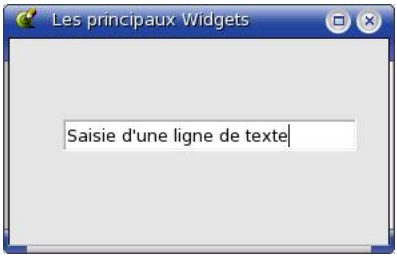
3.5. QListBox

Une `QListBox` permet à l'utilisateur de sélectionner une ou plusieurs entrées dans une liste d'éléments. Si la zone ne peut pas afficher l'ensemble des choix, l'utilsateur peut la faire défiler. Une zone de liste permet en général de sélectionner une chaîne de caractères, mais aussi des données graphiques. La liste est généralement remplie par `insertItem()`.



3.6. QLineEdit

Un `QLineEdit` est un widget fournissant un éditeur sur une seule ligne.



Une ligne éditable permet à l'utilisateur d'entrer et d'éditer une simple ligne de texte avec une collection complète de fonctions d'éditions,

incluant "Annuler" et "Refaire", couper et coller, et glissé et déposer.

Vous pouvez changer le texte par `setText()` ou `insert()`. Le texte est récupéré avec `text()`. Il peut-être aligné par `setAlignment()`.

Quand le texte est modifié, le `signal textChanged()` est émis. Lorsque la touche Entrée ou Retour en enfoncée le `signal returnPressed()` est émis.

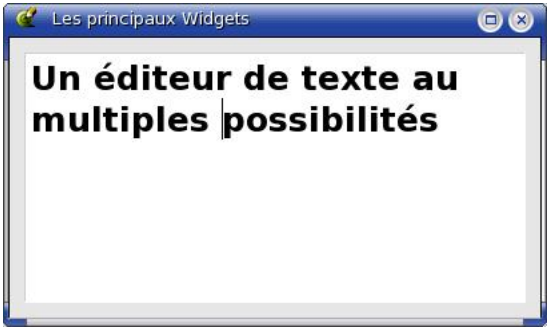
3.7. QSpinBox

La classe `QSpinBox` fournit un champ de saisie incrémentale permettant de sélectionner une valeur numérique dans un ensemble prédéfini. La valeur peut-être modifiée en cliquant sur les boutons haut/bas pour incrémenter/décrémenter la valeur affichée ou en tapant directement la valeur dans le champ de saisie.

A chaque fois que sa valeur change, le `QSpinBox` émet le `signal valueChanged()`. La valeur actuelle peut-être récupérée par `value()` et positionnée avec `setValue()`.

3.8. QTextEdit

`QTextEdit` est un éditeur/visualisateur évolué supportant le "rich text" formaté en utilisant les balises HTML. Il est optimisé pour travailler sur de grands documents et pour répondre rapidement à la saisie de l'utilisateur.



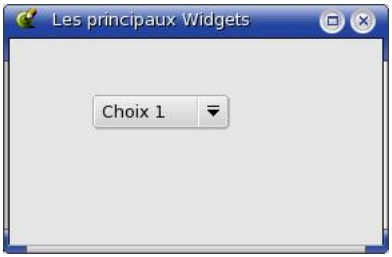
`QTextEdit` possède quatre modes d'opération:

- Editeur de texte simple (PlainText).
- Editeur Rich texte (RichText).
- Visualisateur de texte.
- Visualisateur de log.

`QTextEdit` est tellement riche de possibilités qu'un chapitre complet ne suffirait pas à le décrire.

3.9. QComboBox

Le widget `QComboBox` est la combinaison d'un bouton et d'une liste popup.



Une combo est un widget de sélection qui affiche l'élément courant et peut dérouler une liste d'éléments. Une combo peut être éditable dans ce cas l'utilisateur peut entrer des chaînes arbitraires.

Une entrée peut être ajoutée dans la combo par `insertItem()`. `currentText()` retourne le texte de l'élément sélectionné.

A chaque fois que le texte d'une combo éditable est modifié le `signal textChanged()` est émis.

3.10. QLabel

Le widget `QLabel` est utilisé pour l'affichage d'un texte ou d'une image. Aucune interaction avec l'utilisateur n'est prévue. L'apparence visuelle du label peut-être configurée de plusieurs façons.

Un QLabel peut contenir l'un des types de contenu suivants:

- Plain Text: En passant un `QString` à `setText()`.
- Rich text: En passant un `QString` contenant du "rich text" à `setText()`.
- Un pixmap: En passant un `QPixmap` à `setPixmap()`.
- Un QMovie: En passant un `QMovie` à `setMovie()`.
- Un nombre: En passant un entier ou un double à `setNum()`, ce qui convertit le nombre en texte.
- Rien: Appelé par `clear()`.

Résumé

Ce chapitre a permis de présenter rapidement les principaux Widgets de Qt. Tous les objets nécessaires à la création des fenêtres et dialogues graphiques sont fournis dans la bibliothèque. Le lecteur se reportera à la documentation afin d'obtenir une description détaillée de l'ensemble des widgets.

4. Disposer les widgets dans les dialogues

Chaque widget présent dans un dialogue doit avoir une position et une taille. Qt fournit plusieurs méthodes pour disposer les objets dans une fenêtre que nous allons étudier en utilisant pour l'exemple un dialogue proposant de saisir une fiche d'adhésion:

Voici le code correspondant, d'abord `adhesion.h`:

```
#ifndef ADHESION_H
#define ADHESION_H

#include <qdialog.h>

class QLabel;
class QLineEdit;
class QPushButton;
class QTextEdit;

class Adhesion : public QDialog
{
    Q_OBJECT
public:
     Adhesion(
          QWidget* parent = 0,
          const char* name = 0);

     QLabel* labelNom;
     QLineEdit* nom;
     QLabel* labelPrenom;
```

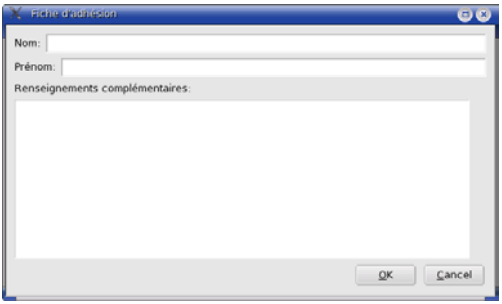


Figure 4.1. Notre fiche d'adhésion correctement disposée.

```
        QLineEdit* prenom;
        QPushButton* boutonOk;
        QPushButton* boutonAnnuler;
        QLabel* labelInfos;
        QTextEdit* infos;
    private slots:
        void slotSaisieValidee();
    private:
        void positionneWidgets();
    };

    #endif
```

Rien de nouveau dans la déclaration de la classe, excepté un nouveau type de widget: `QTextEdit`. Il fournit la possibilité d'éditer du texte dans une page avec de nombreuses possibilités, entre autres celle d'afficher au format *Rich text*. Il sera utilisé dans notre programme comme simple éditeur de texte qui est une de ses possibilités.

puis `adhesion.cpp`:

```
#include "adhesion.h"

#include <qpushbutton.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qtextedit.h>
#include <qmessagebox.h>

Adhesion::Adhesion(
          QWidget* parent,
          const char* name)
     : QDialog(
          parent,
          name)
{
     setCaption( "Adhesion" );

     labelNom =
          new QLabel("Nom:", this);

     nom =
          new QLineEdit( this );

     labelPrenom =
          new QLabel("Prenom:"
               , this);

     prenom = new QLineEdit( this );

     boutonOk = new QPushButton("&Ok", this );
     boutonOk->setAutoDefault( TRUE );
     boutonOk->setDefault( TRUE );

     boutonAnnuler =
          new QPushButton("&Annuler", this );
```

```
        boutonAnnuler->
              setAutoDefault( TRUE );

        labelInfos = new
              QLabel(
              "Renseignements"
              "complementaires:"
              , this );

        infos = new QTextEdit( this );

        connect( boutonOk,
              SIGNAL( clicked() ),
              this,
              SLOT( slotSaisieValidee() ) );
        connect( boutonAnnuler,
              SIGNAL( clicked() ),
              this,
              SLOT( reject() ) );
        positionneWidgets();
    }

    void Adhesion::slotSaisieValidee()
    {
        QMessageBox::information(
              this,
              "Adhesion",
              "Nouvel adherent enregistre" );
    }

    void Adhesion::positionneWidgets()
    {
    }
```

Ici également beaucoup de code bien connut. Une nouveauté malgré tout, l'usage de `QMessageBox::information` affichant un message

dans une boîte de dialogue.

enfin `main.cpp` nécessaire à tout programme Qt:

```
#include <qapplication.h>
#include "adhesion.h"

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

  Adhesion *adhesion = new Adhesion(0);

  a.setMainWidget( adhesion );
  adhesion->show();
  return a.exec();
}
```

Ces trois fichiers présents dans un répertoire nommé par exemple `adhesion` nous nous positionnons dans ce répertoire puis nous exécutons les traditionnels:

```
qmake -project && qmake && make
```

Si nous lançons le programme généré, l'affichage est complètement désordonné. En fait, les widgets ont été placés les uns au-dessus des autres et seul le dernier est visible. Cela produit ce résultat, car notre programme n'a pas positionné ses widgets. Voyons maintenant les différentes manières de procéder:

4.1. Coordonnées absolues

Le positionnement absolu est la manière la plus directe de disposer les widgets dans une fenêtre. Chaque objet reçoit une position x et y ainsi qu'une largeur et hauteur. Ce positionnement est effectué par la méthode `setGeometry()` pour chacun des widgets. Cette méthode attend les quatres paramètres cités plus haut.

C'est la méthode `positionneWidgets()`, laissée vide dans le source précédent qui reçoit le code de disposition:

```
66   void Adhesion::positionneWidgets()
67   {
68        labelNom
69             ->setGeometry(13,13,33,24);
70        nom
71             ->setGeometry(52,13,549,24);
72        labelPrenom
73             ->setGeometry(13,45,52,24);
74        prenom
75             ->setGeometry(71,45,530,24);
76        boutonOk
77             ->setGeometry(436,299,80,30);
78        boutonAnnuler
79             ->setGeometry(522,299,80,30);
80        labelInfos
81             ->setGeometry(11,77,592,17);
82        infos
83             ->setGeometry(12,101,590,190);
84
85    }
```

En exécutant le programme après recompilation, l'affichage est maintenant correct. Chaque widget est positionné à l'endroit demandé dans le dialogue.

Cette façon de faire présente néanmoins de nombreux inconvénients. Si vous essayez d'agrandir ou de réduire la taille de la fenêtre, les widgets restent à leur place. Si vous réduisez de trop la taille, certains vont même disparaîtrent de la fenêtre devenue trop petite. Une solution est d'affecter une taille fixe à la fenêtre:

```
84       setFixedSize(620, 350);
```

Ainsi maintenant la fenêtre ne peut plus être redimensionnée. Néan-

moins, des inconvénients subsistent, si par exemple le dialogue possède un label dont le contenu change au cours du programme. Il peut arriver que le texte à afficher soit trop large et déborde de la fenêtre qui ne pourra pas s'adapter.

Le défaut majeur est quand même de calculer les positions et tailles de chacun des widgets à la main. Si un nouvel objet doit être inséré, il faudra décaler tous les autres et recalculer leurs positions.

4.2. Position avec les layouts

La meilleure solution pour disposer les widgets dans une fenêtre est d'utiliser les layouts de Qt. Les layouts ajustent automatiquement la disposition des éléments du dialogue lorsque du texte ou que la taille de la fenêtre sont modifiés. L'affichage est également recalculé si un widget est caché ou supprimé dans le dialogue.

Qt fournit trois layouts: `QHBoxLayout` qui permet de disposer les widgets horizontalement de la gauche vers la droite. `QVBoxLayout` qui les aligne verticalement du haut vers le bas. Enfin `QGridLayout` permettant de les disposer dans une grille.

Les layouts peuvent contenir des widgets ou d'autres layouts. Les layouts ne sont pas des widgets et à ce titre sont invisibles lors de l'exécution du programme.

La même méthode `positionneWidgets()` est utlisée avec cette fois des layouts:

```
66   void Adhesion::positionneWidgets()
67   {
68
69         HlayoutNom = new QHBoxLayout;
70         HlayoutNom->
71               addWidget( labelNom );
72         HlayoutNom->
73               addWidget( nom );
```

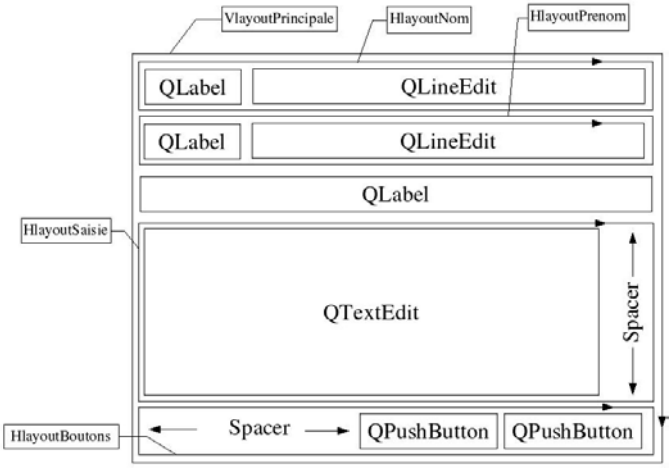


Figure 4.2. Les widgets et les layouts.

Un premier layout horizontal est créé. A l'intérieur sont placés, de la gauche vers la droite le `QLabel labelNom` puis le `QPushButton nom`.

```
75
76          HlayoutPrenom = new QHBoxLayout;
77          HlayoutPrenom->
78               addWidget( labelPrenom );
79          HlayoutPrenom->
80               addWidget( prenom );
81
82          VlayoutNomPrenom = new QVBoxLayout;
83          VlayoutNomPrenom->
84               addLayout( HlayoutNom );
85          VlayoutNomPrenom->
```

```
86              addLayout( HlayoutPrenom );
87
88          HlayoutSaisie = new QHBoxLayout;
89          HlayoutSaisie->
90              addWidget( infos );
```

Le layouts verticaux et horizontaux s'utilisent de la même façon.

```
91          QSpacerItem* spacer =
92              new QSpacerItem( 21, 210,
93              QSizePolicy::Minimum,
94              QSizePolicy::Expanding );
95          HlayoutSaisie->
96              addItem( spacer );
97
98          HlayoutBoutons = new QHBoxLayout;
99          QSpacerItem* spacer2 =
100                 new QSpacerItem( 20, 20,
101                 QSizePolicy::Expanding,
102                 QSizePolicy::Minimum );
103             HlayoutBoutons->
104                 addItem( spacer2 );
105             HlayoutBoutons->
106                 addWidget( boutonOk );
107             HlayoutBoutons->
108                 addWidget(
109                     boutonAnnuler
110                 );
111
112             QVBoxLayout*
113                 VlayoutPrincipale =
114                 new QVBoxLayout(this);
115             VlayoutPrincipale->
116                 addLayout(
117                     VlayoutNomPrenom
118                 );
```

```
119             VlayoutPrincipale->
120                  addWidget(labelInfos);
121             VlayoutPrincipale->
122                  addLayout(HlayoutSaisie);
123             VlayoutPrincipale->
124                  addLayout(HlayoutBoutons);
125             VlayoutPrincipale->
126                  setMargin(11);
127             VlayoutPrincipale->
128                  setSpacing(6);
129        }
```

Deux nouveaux objets, `QSpacerItem` sont créés lignes 91 et 99. `QSpacerItem` permet d'insérer des espaces vides dans les layouts. Il doit recevoir deux paramètres qui sont la largeur et hauteur du spacer. Puis deux règles optionnelles concernant le comportement d'affichage, la première horizontale et la seconde verticale.

Chaque widget de Qt ainsi que les spacers possèdent la méthode `void QWidget::setSizePolicy ( QSizePolicy )`. Elle définie la façon dont l'objet doit être affiché, notamment lorsqu'il doit être redimensionné.

La propriété `sizeHint()` retourne un `QSize` indiquant la taille recommandée pour le widget. Elle renvoie une taille invalide si le widget ne se trouve pas disposé dans un layout. Autrement elle indique la taille préférée pour ce widget dans le layout.

Le paramètre `QSizePolicy` peut avoir les cinq valeurs suivantes:

- **Fixed** : Signifie que le widget prend la taille renvoyée par `QWidget::sizeHint()`. Il ne peut jamais s'agrandir ou rétrécir, il garde toujours la même taille.

- **Minimum** : Signifie que `sizeHint()` est minimal, et suffisant. Le widget ne peut pas se rétrécir au-dessous de cette taille.

- **Maximum** : Signifie que `sizeHint()` est un maximum. Il ne

peut pas être agrandi au dessus de `sizeHint()`.

- **Preferred** : Signifie que `sizeHint()` est sa taille préférée. mais que le widget peut être rétréci ou agrandi si nécessaire.

- **Expanding** : Signifie que le widget peut se rétrécir ou s'agrandir et qu'il est particulièrement disposé à s'agrandir.

Voici quelques captures d'écran montrant le résultat de la modification de la règle d'affichage des widgets.

Le code est montré ci-dessous, les deux boutons sont mis dans une boîte horizontale `QHBox`. La règle d'affichage de `bouton1` est tout d'abord récupérée dans la variable `regle` de type `QSizePolicy`. La valeur concernant la politique horizontale est ensuite modifiée, puis la nouvelle `regle` est positionnée pour le bouton grâce à `setSizePolicy(regle)`. Les captures montrent le résultat de l'affectation des cinq valeurs possibles.

```
1     #include <qapplication.h>
2     #include <qpushbutton.h>
3     #include <qhbox.h>
4
5     int main( int argc, char **argv )
6     {
7           QApplication a( argc, argv );
8
9           QHBox *Hbox = new QHBox(0);
10          QPushButton *bouton1 =
11                new QPushButton(
12                      "Modifie",
13                      Hbox);
14          QPushButton *bouton2 =
15                new QPushButton(
16                      "Minimum",
17                      Hbox);
18          QSizePolicy regle =
```

```
19              bouton1->sizePolicy();
20          regle.setHorData(
21              QSizePolicy::Minimum);
22          bouton1->setSizePolicy(regle) ;
23
24          a.setMainWidget( Hbox );
25          Hbox->setCaption(
26              "setSizePolicy en action"
27              );
28          Hbox->setGeometry(30,30,250,30);
29          Hbox->show();
30          return a.exec();
31      }
32
```

Lors de la création, les widgets ont par défaut la politique d'affichage horizontale positionnée à `Minimum`. Les deux boutons ont la même taille dans la fenêtre.

Dans les schémas suivants, `Minimum` garde la valeur par défaut (`Minimum`), seul *Modifie* est changé par la méthode `setSizePolicy()`.

Valeur `Fixed` pour *Modifie*, la fenêtre est agrandie, mais la taille du bouton ne change pas.

Valeur `Minimum` pour *Modifie*, la fenêtre est réduite, mais le bouton ne diminue pas sa taille en dessous de `sizeHint()`.

Valeur `Maximum` pour *Modifie*, la fenêtre est agrandie mais le bouton n'augmente pas sa taille au delà de `sizeHint()`.

Valeur `Preferred` pour *Modifie*, la fenêtre est agrandie. Le bouton adapte sa taille comme pour `Minimum`.

Valeur `Expanding` pour *Modifie*, la fenêtre est agrandie. Le bouton, lorsque c'est possible, augmente sa largeur plutôt que de la diminuer.

En remplaçant dans un dialogue le bouton `Modifie` par un `QSpacerItem` comme vu plus haut, le ou les autres objets pourront être alignés à gauche ou à droite de la fenêtre. En fait, toutes les combinaisons sont possibles en plaçant judicieusement les spacers et en leurs af-

fectant une politique de disposition adéquate.

Résumé

Nous avons vu dans ce chapitre comment disposer les widgets dans les fenêtres en utilisant les deux méthodes possibles: par coordonnées absolues et avec les layouts.

La disposition avec les layouts est de loin la meilleure solution. Pour les dialogues comportant de nombreux objets graphiques, le développeur pourra utiliser **Qt Designer** avec lequel les délais de création sont grandement accélérés.

5. Fenêtres principales.

Nous allons étudier dans ce chapitre les fenêtres principales. `QMainWindow` fournit une fenêtre principale pour les applications. Elle donne la possibilité d'avoir un menu déroulant, une barre d'outils, une barre de message et d'ouvrir plusieurs fenêtres qui pourront être affichées simultanément. `QMainWindow` supporte les applications MDI (Multiple Document Interface).

Dialogue ou Fenêtre principale ?

On peut légitimement se demander s'il vaut mieux utiliser une classe héritée de `QDialog` ou préférer `QMainWindow`. La réponse dépend de l'application que l'on désire développer. De nombreuses "petites" applications pourront se contenter d'un dialogue. Ce qui ne signifie pas qu'elles soient limitées à un seul dialogue puisqu'un `QDialog` peut en ouvrir d'autres. Mais dès que l'application affiche plusieurs fenêtres, et surtout si celles-ci doivent être affichées en même temps, alors une fenêtre principale est plus adaptée. `QMainWindow` propose de plus des éléments "prêts à l'emploi" tels que menus ou barre d'outils permettant d'écrire des applications plus sophistiquées.

Nous allons voir la création d'un éditeur de texte MDI, c'est-à-dire que plusieurs documents éditables pourront être ouverts en même temps.

Notre application comporte trois classes : `FenetrePrincipale`, `Editeur` et `Recherche` héritants respectivement de `QMainWindow`, `QTextEdit` et `QDialog`. Voici tout d'abord le fichier d'entête

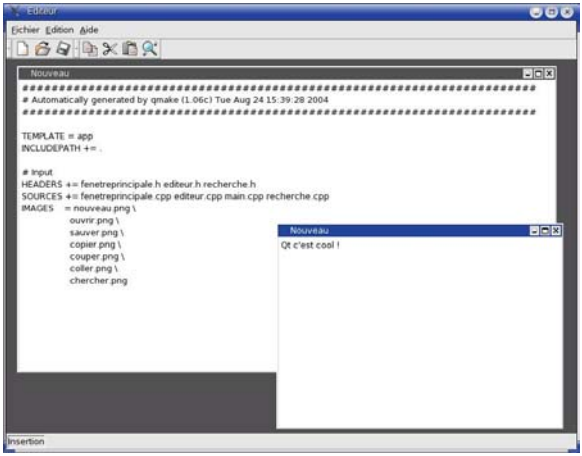


Figure 5.1. Notre éditeur de texte.

`editeur.h` :

```
1     #ifndef EDITEUR_H
2     #define EDITEUR_H
3
4     #include <qtextedit.h>
5
6     class QString;
7
8     class Editeur : public QTextEdit
9     {
10          Q_OBJECT
11    public:
12          Editeur(
13                QWidget *parent,
14                const char *name = 0);
```

```
15          void nouveau();
16          bool ouvrir(QString nom);
17          bool sauver();
18          bool sauverSous();
19          void chercher(
20               const QString &texte,
21               bool suite=false);
22     private:
23          QString nomFichier;
24     signals:
25          void insert();
26     protected:
27          void closeEvent(
28           QCloseEvent *e
29           );
30           void keyPressEvent (
31           QKeyEvent * e
32           );
33     };
34     #endif
35
```

Dans le chapitre concernant les dialogues nous avons étudié la manière d'afficher un dialogue fenêtré. En fait n'importe quel widget Qt peut-être utilisé comme fenêtre. Les exemples qui suivent utilisent un `QTextEdit` affiché de cette façon.

`Editeur` est une classe héritée de `QTextEdit` , une classe permettant facilement d'écrire un éditeur de document simple page. Cette classe possède plusieurs modes de fonctionnement, elle est ici utilisée comme simple éditeur de texte.

La classe définit le `signal insert()`. Ce `signal` sera émis chaque fois que l'utilisateur enfoncera la touche Insertion. Dans notre classe principale, il sera connecté à un `slot` qui se chargera d'afficher dans la barre de statut le mode de saisie (insertion ou remplacement).

Voici l'implémentation `editeur.cpp`

```
1    #include <qfile.h>
2    #include <qstring.h>
3    #include <qtextstream.h>
4    #include <qfiledialog.h>
5    #include <qmessagebox.h>
6
7    #include "editeur.h"
8
9    Editeur::Editeur(QWidget *parent,
10        const char *name)
11        : QTextEdit(parent, name)
12   {
13        setCaption(tr("Nouveau"));
14   }
15
```

L'implémentation commence par inclure les fichiers d'en-têtes Qt des classes utilisées dans le code. Le fichier d'en-tête de la classe `Editeur` est ensuite inclus afin d'avoir accès à la définition de la classe. Le constructeur de la classe reçoit en paramètre les habituels `parent` et `name` puis fait appel au constructeur de base de la classe. `setCaption()` permet de définir le titre de la fenêtre.

```
16   bool Editeur::ouvrir(QString nom)
17   {
18        QFile fichier(nom);
19        if( !fichier.open( IO_ReadOnly ) )
20             return false;
21        QTextStream stream(&fichier);
22        QString texte = stream.read();
23        setText( texte );
24        fichier.close();
25        nomFichier = nom;
26        setCaption( nomFichier );
```

```
27          return true;
28    }
29
```

La méthode publique `ouvrir` est appelée à partir de la fenêtre principale lorsque l'utilisateur a choisi d'ouvrir un fichier sur disque. Le nom du fichier est transmis à la méthode dans le `QString nom`.

La classe `QFile` se charge d'ouvrir le fichier en lecture seule (et quitte la fonction si l'ouverture échoue). Une variable `QTextStream stream` stocke le contenu du fichier texte dans un `QString` qui est ensuite affecté comme texte de notre classe `QTextEdit` en utilisant `setText()`. Le lecteur pourra se reporter au chapitre **Entrées/Sorties** pour plus de détails concernant `QFile` et `QTextStream`.

```
30    bool Editeur::sauver()
31    {
32          if ( nomFichier.isEmpty() ) {
33                sauverSous();
34                return true;
35          }
36          QFile fichier( nomFichier );
37          if ( !fichier.
38                open( IO_WriteOnly ) ) {
39                return false;
40          }
41          QTextStream stream( &fichier );
42          stream << text();
43          fichier.close();
44
45          setCaption( nomFichier );
46          setModified( false );
47          return true;
48    }
49
50    bool Editeur::sauverSous()
51    {
```

```
52          QString fichier = QFileDialog::
53               getSaveFileName(
54               nomFichier,
55               QString::null,
56               this );
57          if ( !fichier.isEmpty() ) {
58               nomFichier = fichier;
59               sauver();
60               return true;
61          } else {
62               return false;
63          }
64     }
65
```

Un `QFile` et un `QTextStream` sont également utilisés, mais cette fois pour enregistrer dans un fichier le contenu de la fenêtre de texte. Dans la méthode `sauverSous()`, la méthode statique `getSaveFileName()` de la classe `QFileDialog` est utilisée. Elle permet à l'utilisateur de sélectionner un fichier sur disque afin d'y enregistrer le contenu de la fenêtre. Si le dialogue de sélection a été annulé, la méthode renverra une chaîne vide dans `fichier`. C'est ce qui est testé en ligne 57 avant d'appeler la méthode `sauver()`.

`ouvrir()`, `sauver()`, et `sauverSous()` ne sont pas des `slot`. Ce sont des méthodes publiques de la classe qui sont appelées à partir de la fenêtre principale `fenetrePrincipale` lorsque les actions équivalentes de cette dernière sont déclenchées.

```
66     void Editeur::closeEvent(
67            QCloseEvent *e
68            )
69     {
70         if ( isModified() ) {
71          switch( QMessageBox::warning(
72               this,
73               "Sauvegarde",
```

```
74                  tr("Le fichier \"%1\""
75              " a ete"
76              " modifie.\nVoulez-vous"
77              " l'enregistrer ?")
78              .arg( caption() ),
79              tr("&Enregistrer"),
80              tr("&Abandonner"),
81              tr("A&nnuler") ) )
82         {
83         case 0:
84             {
85              sauver();
86              if ( !nomFichier.isEmpty() )
87                  e->accept();
88              else
89                  e->ignore();
90             }
91             break;
92         case 1:
93             e->accept();
94             break;
95         default:
96             e->ignore();
97             break;
98         }
99        } else {
100             e->accept();
101            }
102        }
103
```

L'évènement `closeEvent()` est réimplémenté afin d'intercepter la fermeture de la fenêtre (voir le paragraphe **Intercepter les évènements QEvent**). Ceci nous permet de proposer la sauvegarde du document si celui-ci a été modifié.

Tout d'abord, `isModified()` permet de savoir si le texte de l'éditeur a été modifié, sinon inutile de proposer une sauvegarde. `isModified()`, méthode appartenant à la classe `QTextEdit` renvoie `TRUE` si le document a été modifié par l'utilisateur, autrement retourne `FALSE`.

Si le texte a été modifié, la sauvegarde est proposée en utilisant la méthode statique `warning` de la classe `QMessageBox`. Elle permet d'afficher un message ainsi qu'au moins deux boutons (ici trois). Ainsi l'utilisateur peut choisir une option puis le programme récupère la valeur du bouton cliqué. La valeur 0 correspond au premier bouton, 1 au deuxième etc.

L'évènement `closeEvent()` intercepté peut être confirmé par `accept()`, c'est-à-dire que la fenêtre est effectivement fermée. Il peut également être annulé par `ignore()`. C'est utile dans notre application, car si l'utilisateur choisit `Annuler`, alors la fenêtre ne doit pas être fermée.

```
104        void Editeur::keyPressEvent (
105             QKeyEvent * e
106        )
107        {
108             if ( e->key()
109                  == Qt::Key_Insert )
110                  emit insert();
111             QTextEdit::keyPressEvent (
112                  e);
113        }
114
```

`keyPressEvent()` est également un évènement intercepté. Chaque fois que l'utilisateur appuie sur une touche du clavier, cette méthode est appelée. Nous vérifions ligne 108 si la touche enfoncée correspond à la touche insertion (`Qt::Key_Insert`). Si c'est le cas, le `signal insert()` est émis. Ce `signal` est connecté à un `slot` dans notre fenêtre principale. Ce `slot` met à jour le mode d'édition dans la barre de statut.

Enfin la méthode `keyPressEvent()` de la classe de base `QTextEdit` est appelée en lui passant en paramètre l'évènement reçu. Si nous ne faisions pas cela, les frappes clavier ne seraient plus prises en compte. Notre méthode fonctionnerait en "trou noir".

```
115       void Editeur::chercher(const QString
&texte,
116            bool suite)
117       {
118            if( !suite )
119                 setCursorPosition(0,0);
120            find(texte, false, false);
121       }
```

La méthode publique `chercher(const QString &texte)` est elle aussi appelée par la fenêtre principale et reçoit en paramètre le texte à rechercher. Elle vérifie seulement s'il s'agit d'une nouvelle recherche. Dans ce cas, le curseur est placé dans l'éditeur au début du texte. La méthode `find` de la classe `QTexteEdit` est ensuite appelée. Celle-ci se positionne sur le texte s'il est trouvé.

Lorsque l'utilisateur choisira de rechercher du texte dans une fenêtre d'édition, un dialogue lui sera présenté afin de saisir le texte de recherche. Voici le fichier d'en-tête du dialogue de recherche : `recherche.h`.

```
1    #ifndef RECHERCHE_H
2    #define RECHERCHE_H
3    #include <qdialog.h>
4    class QPushButton;
5    class QLineEdit;
6    class Recherche : public QDialog
7    {
8         Q_OBJECT
9    public:
10        Recherche(
11             QWidget *parent,
```

```
12              const char *name = 0);
13    signals:
14          void chercher(const QString &);
15    public slots:
16          void slotChercher();
17    private:
18          QPushButton *boutonChercher;
19          QPushButton *boutonAnnuler;
20          QLineEdit *saisie;
21    };
22    #endif
```

Rien de nouveau dans cette définition de classe. Notez la définition du `signal void chercher(const QString &)` qui sera émis lorsque le bouton `boutonChercher` sera enfoncé.

```
1     #include <qpushbutton.h>
2     #include <qlayout.h>
3     #include <qlineedit.h>
4     #include <qlabel.h>
5
6     #include "recherche.h"
7
8     Recherche::Recherche(QWidget *parent,
9           const char *name)
10          : QDialog(parent, name)
11    {
12          setCaption(tr("Recherche"));
13          QLabel *label =
14                new QLabel(
15                "Texte a chercher",
16                this);
17          saisie = new QLineEdit(this);
18
19          boutonChercher =
20                new QPushButton(
```

```
21              "&Chercher",
22              this);
23          connect(
24              boutonChercher,
25              SIGNAL(clicked()),
26              this,
27              SLOT(slotChercher()) );
28          boutonAnnuler =
29              new QPushButton("&Annuler",
30              this);
31          connect(
32              boutonAnnuler,
33              SIGNAL(clicked()),
34              this,
35              SLOT(reject()) );
36
37          QHBoxLayout *h1 = new QHBoxLayout;
38          h1->addWidget(label);
39          h1->addWidget(saisie);
40
41          QVBoxLayout *v1 = new QVBoxLayout;
42          v1->addLayout(h1);
43          QSpacerItem *spacerV =
44              new QSpacerItem(20,50);
45          v1->addItem(spacerV);
46
47          QVBoxLayout *v2 =
48              new QVBoxLayout;
49          v2->addWidget(boutonChercher);
50          v2->addWidget(boutonAnnuler);
51          QSpacerItem *spacer =
52              new QSpacerItem(20,50);
53          v2->addItem(spacer);
54
55          QHBoxLayout *principale =
```

```
56              new QHBoxLayout(this);
57          principale->addLayout(v1);
58          principale->addLayout(v2);
59          principale->
60              setMargin(11);
61          principale->
62              setSpacing(6);
63      }
64
```

Comme pour toutes classes héritées d'une classe Qt, le constructeur de base de la classe est appelé (ligne 10). La création des différents widgets du dialogue est effectuée puis ces derniers sont disposés dans des layouts. Le bouton `boutonChercher` est connecté au `slot slotChercher()` qui se chargera d'émettre le signal. `boutonAnnuler` est quant à lui connecté au `slot close()` de la classe `QDialog` chargé de fermer le dialogue.

```
65      void Recherche::slotChercher()
66      {
67          emit chercher(saisie->text());
68          close();
69      }
```

Le `slot slotChercher()` se contente d'émettre le `signal` avec `emit` en passant en paramètre le texte du champ de saisie.

```
1       #include <qapplication.h>
2       #include "fenetreprincipale.h"
3
4       int main( int argc, char **argv )
5       {
6         QApplication a( argc, argv );
7
8         FenetrePrincipale
9             *fenetrePrincipale =
10                new
```

```
11              FenetrePrincipale(0);
12
13      a.setMainWidget( fenetrePrincipale );
14      fenetrePrincipale->showMaximized();
15      return a.exec();
16    }
17
```

Chaque application a besoin de sa fonction `main()`. Elle est implémentée dans le fichier du même nom : `main.cpp`.

5.1. Sous-classer QMainWindow

`QDialog` et `QMainWindow` héritent tous deux de `QWidget`, la façon de créer une classe héritée d'un des deux types est donc similaire. Tout le code de cette application, pour les besoins du livre a été écrit à la main. Les dialogues ainsi que la fenêtre principale auraient pu être conçus avec **Qt Designer**. Le lecteur pourra se reporter au chapitre correspondant afin de connaître la manière de concevoir des applications avec cet outil. Voici tout d'abord le fichier d'entête `fenetreprincipale.h`

```
1     #ifndef FENETREPRINCIPALE_H
2     #define FENETREPRINCIPALE_H
3     #include <qmainwindow.h>
4     class QAction;
5     class QLabel;
6     class Editeur;
7     class QWorkspace;
8     class Recherche;
9     class FenetrePrincipale
10          : public QMainWindow
11    {
12          Q_OBJECT
13    public:
14          FenetrePrincipale(
```

```
15              QWidget *parent,
16              const char *name = 0);
17    private slots:
18          void slotAPropos();
19          Editeur* slotNouveau();
20          void slotOuvrir();
21          void slotSauver();
22          void slotSauverSous();
23          void slotCopier();
24          void slotCouper();
25          void slotColler();
26          void slotDialogueChercher();
27          void slotChercher(const QString &);
28          void slotPoursuivre();
29          void majBarreStatut();
```

La classe `FenetrePrincipale` hérite de `QMainWindow`, elle en aura donc les caractéristiques. Parmi les slots privés à la classe se trouve `slotNouveau()` qui renvoie un pointeur de type `Editeur`. Cette classe qui est détaillée plus loin hérite de `QTextEdit` et constitue la fenêtre d'édition de notre application.

```
33    protected:
34          void closeEvent(
35               QCloseEvent *e );
```

La fonction `closeEvent(QCloseEvent *)` qui est déclarée va nous permettre, en la ré-implémentant, d'intercepter l'évènement de fermeture de la fenêtre. Cette méthode sera décrite en détail en étudiant l'implémentation de la classe.

```
36    private:
37          void creeActions();
38          void creeMenus();
39          void creeBarreOutils();
40          void creeBarreStatut();
41          void nouveau();
```

```
42
43          QPopupMenu *menuFichier;
44          QAction *actionOuvrir;
45          QAction *actionNouveau;
46          QAction *actionSauver;
47          QAction *actionSauverSous;
48          QAction *actionQuitter;
49
50          QPopupMenu *menuEdition;
51          QAction *actionCopier;
52          QAction *actionCouper;
53          QAction *actionColler;
54          QAction *actionChercher;
55          QAction *actionPoursuivre;
56
57          QPopupMenu *menuAide;
58
59          QToolBar *barreFichier;
60          QToolBar *barreEdition;
61
62          QLabel* ligneCol;
63          QLabel* frappe;
64          QWorkspace *workspace;
65
66          Recherche* recherche;
67          bool insertion;
68          QString texteRecherche;
69    };
70    #endif
```

Quelques pointeurs sur des objets d'un nouveau type `QPopupMenu`, `QAction`, `QToolBar` et `QWorkspace` détaillés également dans l'implémentation. Voici justement l'implémentation `fenetreprincipale.cpp`

```
1     #include <qaction.h>
```

```
2     #include <qpixmap.h>
3     #include <qpopupmenu.h>
4     #include <qapplication.h>
5     #include <qmenubar.h>
6     #include <qmessagebox.h>
7     #include <qworkspace.h>
8     #include <qfiledialog.h>
9     #include <qstatusbar.h>
10    #include <qlabel.h>
11    #include "fenetreprincipale.h"
12    #include "editeur.h"
13    #include "recherche.h"
14
```

Nous incluons comme toujours les fichiers d'en-têtes des classes Qt utilisées ainsi que les deux classes `editeur.h` et `recherche.h` respectivement héritées d'un `QTextEdit` et d'un `QDialog`.

```
16    FenetrePrincipale::
17          FenetrePrincipale(
18          QWidget *parent,
19          const char *name)
20          : QMainWindow(parent, name)
21    {
22          setCaption(tr("Editeur"));
23          workspace =
24                new QWorkspace(this);
25          setCentralWidget(
26                workspace);
```

Le widget `QWorkspace` fournit une fenêtre de zone de travail qui peut contenir les fenêtres, par exemple pour les applications MDI. Notre programme étant MDI doit utiliser un `QWorkspace`.

Les applications MDI ont typiquement une fenêtre principale avec une barre de menu, une barre d'outils, une barre de statut et un widget cen-

tral qui est un `QWorkspace`. La zone de travail elle-même peut contenir une ou plusieurs fenêtres de document, dont chacune est un widget.

`QWorkspace` est un widget ordinaire de Qt et chaque fenêtre ouverte est un de ses enfants. Il a un constructeur standard qui prend un widget parent et un nom d'objet. La fenêtre parent est habituellement un `QMainWindow`, mais pas obligatoirement.

```
27          creeActions();
28          creeMenus();
29          creeBarreOutils();
30          creeBarreStatut();
31          recherche = 0;
32          insertion = true;
33      }
```

Dans le constructeur de notre classe, les méthodes de création des actions, menus, barre d'outils et barre de statut sont appelées.

L'utilisateur pourra, à partir de la fenêtre principale appeler une fenêtre de recherche. Cette classe `Recherche` ne sera allouée qu'une seule fois lors du premier appel, puis seulement affiché. Mettre 0 dans le pointeur `recherche` nous permettra, lors du premier appel de savoir que la classe n'a pas encore été instanciée.

Lors du lancement de l'application, le curseur est en mode insertion. Chaque appui sur la touche `Insert` va basculer ce mode entre le mode insertion et remplacement. Le mode actuel sera affiché dans la barre de statut en bas de la fenêtre et la variable `insertion` permet de savoir dans quel mode se trouve le curseur.

5.2. Création des actions.

La méthode `creeActions()` qui est en partie montrée ici est basée sur l'utilisation de la classe `QAction`. La classe `QAction` fournit une action abstraite d'interface utilisateur qui peut apparaître à la fois dans les menus et les barres d'outils.

Figure 5.2. La fenêtre de recherche.

```
35   void FenetrePrincipale::creeActions()
36   {
37        actionNouveau =
38        new QAction(
39             tr("&Nouveau"),
40             tr("Ctrl+N"),
41             this);
42        actionNouveau
43             ->setIconSet(
44             QPixmap::fromMimeSource(
45             "nouveau.png"
46             )
47             );
48        actionNouveau
49             ->setStatusTip(
50             tr("Creation d'un "
51             "nouveau document"));
52        connect(
53             actionNouveau,
54             SIGNAL(activated()),
55             this,
56             SLOT(slotNouveau()));
```

```
57
58          actionOuvrir =
59          new QAction(
60               tr("&Ouvrir"),
61               tr("Ctrl+O"),
62               this);
63          actionOuvrir
64               ->setIconSet(
65               QPixmap::fromMimeSource(
66               "ouvrir.png"
67               )
68               );
69          actionOuvrir
70               ->setStatusTip(
71               tr("Ouverture d'un"
72               " document existant"));
73          connect(
74               actionOuvrir,
75               SIGNAL(activated()),
```

Dans des applications GUI beaucoup de commandes peuvent être appelées par l'intermédiaire d'une option de menu, d'un bouton de barre d'outils et d'un accélérateur de clavier. Puisque la même action doit être effectuée indépendamment de la façon dont l'action a été appelée, et puisque le menu et la barre d'outils devraient être maintenus synchronisés, il est utile de représenter une commande comme action. Une action peut être ajoutée à un menu et à une barre d'outils et les maintiendra automatiquement synchronisés. Par exemple, si l'utilisateur appuie sur un bouton de barre d'outils "italique" l'entrée de menu "italique" sera automatiquement cochée.

Un `QAction` peut contenir une icône, un texte de menu, un accélérateur, un texte de statut, un "whats this text" et un tooltip. La plupart de ces derniers peuvent être placés dans le constructeur. Ils peuvent également être placés indépendamment avec `setIconSet()`,

`setText()`, `setMenuText()`, `setToolTip()`, `setStatusTip()`, `setWhatsThis()` et `setAccel()`.

Les actions sont ajoutées aux widgets (des menus ou des barres d'outils) en employant `addTo()`, et enlevées en utilisant `removeFrom()`. Une fois qu'un `QAction` a été créé il devrait être ajouté au menu approprié et barre d'outils puis être relié au `slot` qui effectuera l'action.

Dans le programme, chaque `QAction` est construit en lui passant en paramètres : le texte à afficher, le raccourci clavier et son objet parent qui est ici l'instance de la classe actuelle représentée par le pointeur `this`. Un icône lui est ensuite affecté qui sera affiché dans le menu et dans la barre d'outils.

Intégrer des images dans un programme Qt peut se faire de trois manières différentes :

1 Les images sont stockées dans des fichiers chargés au moment de l'exécution.

2 Les images sont intégrées dans le code source de l'application. C'est possible car les fichiers XPM sont des fichiers C++ valides.

3 En utilisant le mécanisme de collection d'images de Qt.

La solution de la collection d'images est intéressante car celles-ci sont intégrées au programme. Le temps de chargement de ces images est bien entendu grandement accéléré puisqu'elle sont chargées avec l'exécutable. Cela règle de plus les éventuels problèmes de chemin d'accès aux images. Une collection d'images est créée très facilement en rajoutant dans le fichier projet (*.pro) la rubrique IMAGES comme dans le fichier `editeur.pro`:

```
TEMPLATE = app
INCLUDEPATH += .

# Input
HEADERS += fenetreprincipale.h editeur.h
```

```
recherche.h
SOURCES += fenetreprincipale.cpp editeur.cpp
main.cpp recherche.cpp
IMAGES          = nouveau.png \
       ouvrir.png \
       sauver.png \
       copier.png \
       couper.png \
       coller.png \
       chercher.png
```

Lors de la compilation, `uic` construit un fichier nommé `qmake_image_collection.cpp` qui sera compilé et lié comme les autres fichiers sources. Ce fichier contient les images sous forme de tableaux de données.

L'accès aux images peut être réalisé dans le programme en utilisant la syntaxe `QPixmap::fromMimeSource("image.png")` sans être tributaire de fichiers sur disque qui pourraient être supprimés.

La connexion de ces actions est ensuite effectuée aux `slot` adéquats selon la syntaxe habituelle.

```
196             connect(
197                  actionPoursuivre,
198                  SIGNAL(activated()),
199                  this,
200                  SLOT(slotPoursuivre()));
201
202
203       }
```

5.3. Menu et barres d'outils

Les actions qui ont été précédemment créées peuvent maintenant être mises dans le menu déroulant de notre application:

```
205        void FenetrePrincipale::creeMenus()
206        {
207             menuFichier =
208                  new QPopupMenu(this);
```

Un `QPopuMenu` est tout d'abord créé. Un widget de menu popup est un menu de sélection. Ce peut être un menu déroulant dans une barre de menu ou un menu autonome de contexte (popup). Des menus déroulants sont affichés par la barre de menu quand l'utilisateur clique sur le titre de menu ou appuie sur la touche de raccourci correspondante. Employez `QMenuBar::insertItem()` pour insérer un menu de popup dans une barre de menu. Dans le cas des `QAction`, celle-ci doivent être insérées en employant `QMenuBar::addTo()`

```
209             actionNouveau->
210                  addTo( menuFichier);
211             actionOuvrir->
212                  addTo( menuFichier);
213             actionSauver->
214                  addTo( menuFichier);
215             actionSauverSous->
216                  addTo( menuFichier);
217             menuFichier->
218                  insertSeparator();
219             actionQuitter->
220                  addTo( menuFichier);
221
222             menuBar()->
223                  insertItem(
224                       tr("&Fichier"),
225                       menuFichier);
226             menuEdition =
227                  new QPopupMenu(this);
228             actionCopier->
229                  addTo( menuEdition);
```

Le `QPopupMenu` est ensuite accroché à la barre de menu accessible par le pointeur `menuBar()` de notre classe `QMainWindow`. C'est fait en utilisant la fonction `insertItem()` avec deux paramètres : le texte à afficher comme titre de menu et la pointeur `QPopupMenu` à insérer.

```
230             actionCouper->
231                  addTo( menuEdition);
232             actionColler->
233                  addTo( menuEdition);
234             menuEdition->
235                  insertSeparator();
236             actionChercher->
237                  addTo( menuEdition);
238             actionPoursuivre->
239                  addTo( menuEdition);
240             menuBar()->
241                  insertItem(
242                       tr("&Edition"),
243                       menuEdition);
244             menuAide =
245                  new QPopupMenu(this);
246             menuAide->
247                  insertItem(
248                  "A propos de l'editeur",
249                  this,
250                  SLOT(slotAPropos()),
251                  Key_F1
252                  );
253
254             menuAide->
255                  insertItem(
256                  "A propos de Qt",
257                  qApp,
258                  SLOT(aboutQt())
259                  );
260
```

```
261             menuBar()->
262                   insertItem(
263                         tr("&Aide"),
264                         menuAide);
265
266       }
```

Les deux dernières entrées ajoutées au popup `menuAide` ne sont pas des `QAction` et doivent être insérées par la méthode `insertItem()`. Cette dernière permet de passer en paramètres le texte à afficher, l'adresse de l'objet parent, le `slot` à connecter et enfin le raccourci clavier. Un `QAction` est inutile car nos entrées n'ont pas d'images et ne sont pas présentes dans la barre d'outils. Le `slot aboutQt()` accessible grâce au pointeur global `qApp` sur l'instance `QApplication` affiche une boîte de dialogue présentant quelques informations concernant Qt, notamment le numéro de version utilisé.

La barre d'outils est ensuite créée en utilisant, les actions comme pour le menu :

```
268       void FenetrePrincipale
269             ::creeBarreOutils()
270       {
271             barreFichier =
272                   new QToolBar
273                   ( tr("Fichier"),
274                   this);
```

La barre d'outils **Fichier** est instanciée, elle porte le nom "Fichier" et a comme parent `this`. Une barre d'outils est un panneau qui contient un ensemble de commandes, habituellement représentées par de petites icônes. Son but est de fournir l'accès rapide aux commandes ou aux options fréquemment utilisées. Dans un `QMainWindow`, l'utilisateur peut déplacer les barres d'outils à l'intérieur et entre les secteurs de dock. Une barre d'outils peut également être déplacée hors de n'importe quel secteur de dock pour flotter librement en tant que fenêtre autonome.

```
275             actionNouveau->
```

```
276                 addTo( barreFichier );
277             actionOuvrir->
278                 addTo( barreFichier );
279             actionSauver->
280                 addTo( barreFichier );
281             barreEdition =
282                 new QToolBar
283                 ( tr("Edition"),
284                 this);
285             actionCopier->
286                 addTo( barreEdition );
287             actionCouper->
288                 addTo( barreEdition );
289             actionColler->
290                 addTo( barreEdition );
291             actionChercher->
292                 addTo( barreEdition );
293         }
```

Chaque action créée précédemment est maintenant affectée à la barre d'outils. Notez que la syntaxe est la même que pour le menu.

5.4. Barre de statut

```
296         void FenetrePrincipale
297             ::creeBarreStatut()
298         {
299             frappe = new
300                 QLabel(this);
301             statusBar()->
302                 addWidget(frappe);
303             majBarreStatut();
304         }
```

La classe `QStatusBar` fournit une barre horizontale en bas de la fenêtre principale qui est utilisée pour présenter des informations de statut. La barre de statut peut présenter de l'information pouvant appartenir à l'une des trois catégories :

- **Temporaire**: occupe brièvement la majeure partie de la barre de statut.
- **Normal**: occupe une partie de la barre de statut et peut être caché par les messages temporaires.
- **Permanent**: n'est jamais caché. Utilisé pour des indications importantes de mode, par exemple, quelques applications mettent un indicateur de majuscule dans la barre de statut.

Notre barre de statut comporte un `QLabel` qui va être utilisé pour indiquer si le curseur de l'éditeur est en mode insertion ou remplacement. Ce widget est affiché en mode Permanent.

```
306        void FenetrePrincipale
307             ::majBarreStatut()
308        {
309             insertion = !insertion;
310             if( insertion )
311                  frappe->
312                       setText(
313                       tr("Insertion")
314                       );
315             else
316                  frappe->
317                       setText(
318                       tr("Refrappe")
319                       );
320        }
```

Chaque fois que ce `slot` est déclenché, il inverse la valeur du booléen `insertion` puis affiche "Insertion" ou "Refrappe" dans le la-

bel `frappe` de la barre de statut. Le `slot` est connecté au `signal insert()` de la classe `Editeur`. Ainsi dans l'éditeur, lorsque la touche insertion est enfoncée (voir l'implémentation de `editeur.cpp` pour plus de détails), le signal est émis déclenchant le `slot` de notre fenêtre principale.

5.5. Ouvrir des dialogues.

Dans cette section, nous allons étudier comment utiliser les dialogues.

5.5.1. Fenêtres documents MDI

```
322        Editeur *FenetrePrincipale
323             ::slotNouveau()
324        {
325             Editeur *editeur =
326                  new Editeur(workspace);
327             editeur->show();
328             connect(
329                  editeur,
330                  SIGNAL(insert()),
331                  this,
332                  SLOT(majBarreStatut())
333             );
334             return editeur;
335        }
336
```

Le `slot slotNouveau()` est connecté à l'action `actionNouveau` qui peut être déclenchée par la barre d'outils ou le menu correspondant. L'action à effectuer est d'ouvrir une nouvelle fenêtre dans l'éditeur. Une nouvelle instance de la classe `Editeur` qui hérite de `QTextEdit` est allouée, l'adresse est stockée dans le pointeur `editeur`. Elle a

comme parent le pointeur `workspace` de type `QWorkSpace` comme expliqué précédemment. La fenêtre est ensuite affichée par la méthode `show()`. Nous connectons le `signal` de la classe `Editeur` au `slot majBarreStatut()`. Pour terminer, en fin de méthode, le pointeur alloué est retourné. Notez que si le `slot` est déclenché par une action sur le menu ou la barre d'outils, le pointeur renvoyé sera ignoré.

5.5.2. Intercepter les évènements QEvent

```
373       void FenetrePrincipale::
374            closeEvent( QCloseEvent *e )
375       {
376           QWidgetList fenetres =
377                  workspace->windowList();
378           if ( fenetres.count() ) {
379            for (
380            int i = 0;
381            i < int(fenetres.count());
382            ++i )
383            {
384                QWidget *fenetre
385                       = fenetres.at( i );
386                if ( !fenetre->close() ) {
387                 e->ignore();
388                 return;
389                }
390            }
391           }
392           QMainWindow::closeEvent( e );
393       }
394
```

La classe `QEvent` est la classe de base de toutes les classes d'événement. La boucle principale d'événement de Qt

(`QApplication::exec()`) cherche des événements du système de fenêtre dans la file d'attente d'événement, les traduit en `QEvent` et envoie les événements traduits aux `QObject`. Les `QObject` reçoivent des événements en appelant leur fonction `QObject::event()`. La fonction peut être réimplémentée dans les sous-classes pour adapter l'événement et pour ajouter du traitement additionnel ; `QWidget::event()` est un bon exemple.

La classe `QEvent` est héritée par d'autres classes d'évènements plus spécialisées dont voici les principales:

- **QTimerEvent**: Cette classe permet de gérer les évènements de timer.
- **QMouseEvent**: Cette classe permet de gérer les évènements de la souris.
- **QKeyEvent**: Cette classe permet de gérer les évènements du clavier.
- **QResizeEvent**: Cette classe permet de gérer les évènements de redimensionnement des objets.
- **QCloseEvent**: Cette classe permet de gérer les évènements de fermeture des objets.

La classe `QMainWindow` elle-même une sous-classe de `QWidget` peut réimplémenter la fonction `QCloseEvent()` et c'est la seule façon d'intercepter la fermeture de la fenêtre effectuée par le bouton système "croix" de la fenêtre.

Dans notre application, la fenêtre principale récupère la liste de ses documents, puis leurs demandes de se fermer les uns après les autres. Lorsqu'une fenêtre document se ferme, l'évènement est intercepté, car cette classe réimplémente elle aussi l'évènement `closeEvent()`. La classe `Editeur` est ainsi en mesure de proposer la sauvegarde du document si celui-ci a été modifié.

5.5.3. Fenêtre MDI active.

```
396        void FenetrePrincipale
397             ::slotSauver()
398        {
399            Editeur* e =
400                   (Editeur*)workspace->
401                   activeWindow();
402            if ( e )
403             e->sauver();
404        }
405
406        void FenetrePrincipale
407             ::slotSauverSous()
408        {
409            Editeur* e =
410                   (Editeur*)workspace->
411                   activeWindow();
412            if ( e )
413             e->sauverSous();
414        }
415
```

Une fenêtre de document devient active quand elle obtient le focus du clavier. Vous pouvez également activer une fenêtre dans votre code en utilisant `setFocus()`. L'utilisateur peut activer une fenêtre en déplaçant le focus par l'une des manières habituelles, par exemple en cliquant une fenêtre ou en enfonçant la touche tabulation. La zone de travail émet un signal `windowActivated()` quand il détecte le changement d'activation, et la fonction `activeWindow()` renvoie toujours un pointeur de la fenêtre de document active.

Le pointeur `e` reçoit l'adresse de la fenêtre active et, si ce pointeur n'est pas nul, c'est-à-dire qu'une fenêtre est active appelle la méthode `sauver()` de la classe `Editeur`.

5.5.4. Dialogues hérités de QDialog.

```
416        void FenetrePrincipale
417             ::slotDialogueChercher()
418        {
419             if ( !recherche )
420             {
421                  recherche =
422                       new Recherche(this);
423                  connect(
424                  recherche,
425                  SIGNAL(
426                       chercher(
427                       const QString &
428                       )
429                  ),
430                  this,
431                  SLOT(
432                       slotChercher(
433                       const QString &
434                       )
435                  )
436                  );
437             }
438             recherche->show();
439             recherche->raise();
440             recherche->setActiveWindow();
441        }
442        void FenetrePrincipale
443             ::slotChercher(const QString
&texte)
444        {
```

```
445             Editeur *editeur =
446                    (Editeur *)
447                   workspace->
448                         activeWindow();
449             if( editeur )
450                   editeur->
451                         chercher(texte);
452             texteRecherche = texte;
453       }
```

Dans le constructeur de la classe, le pointeur `Recherche` a été mis à 0. Lors du premier appel de ce `slot`, `if(!recherche)` renvoie vrai c'est-à-dire que `recherche==0`. La classe `Recherche` est donc instanciée et son adresse est stockée dans `recherche` (lors des autres passages, `recherche`, contenant une adresse, l'affectation n'aura plus lieu). Le dialogue Recherche, lorsque l'utilisateur clique le bouton **Chercher**, émet le `signal chercher(const QString &)` passant en paramètre un `QString` contenant le texte à rechercher. Ce `signal` est connecté au `slot slotChercher(const QString &)`.

Le dialogue de recherche est ensuite affiché grâce à `show()` puis placé au dessus des autres fenêtres avec `raise()` et enfin rendu actif par `setActiveWindow()`.

Il existe deux façons principales d'afficher les dialogues :

- **show()**. Affiche le dialogue et ses enfants. Le dialogue est mis en mode non modal, c'est-à-dire que l'utilisateur pourra basculer sur les autres fenêtres de l'application sans avoir à fermer le dialogue. C'est utile par exemple dans notre fenêtre de recherche qui pourra rester ouverte alors que du texte est modifié dans l'éditeur.

- **exec()** Affiche le dialogue comme dialogue modal, bloquant jusqu'à ce que l'utilisateur le ferme. Les utilisateurs ne peuvent pas agir sur d'autres fenêtres dans la même application jusqu'à ce le dialogue soit fermé. La fonction renvoie un résultat qui peut être

`QDialog::Accepted` ou `QDialog::Rejected`.

La classe `QDialog` possède pour cela deux `slot`: `accept()` et `reject()`. Une bonne manière de procéder dans les dialogues modaux est de relier deux boutons, par exemple Ok et Annuler respectivement aux `slots accept()` et `reject()`. Chacun d'eux fermera le dialogue en renvoyant une valeur différente. Cette valeur peut-être interprétée comme ceci:

```
MonDialogueModal *dialogue =
          new MonDialogueModal(this);
     if( dialogue->exec() ==
          QDialog::Accepted )
     {
          QMessageBox::information(
               this,
               "Mon application ",
               "L'utilisateur a valide"
               " le dialogue." );
     }
     else
     {
          QMessageBox::information(
               this,
               "Mon application ",
               "Dialogue rejete." );
     }
```

Résumé

Nous avons vu dans ce chapitre la façon de créer les fenêtres principales ainsi que les menus et les barres d'outils. La classe `QAction` apporte dans ce domaine une grande puissance de développement puisqu'elle permet à l'aide d'une seule action de piloter les menus déroulants et les barres d'outils. Nous avons également étudié la façon d'afficher des dialogues dans la fenêtre principale.

6. Les Entrées/Sorties

Beaucoup d'applications doivent à un moment ou un autre manipuler des données stockées dans des fichiers. Qt, dans un souci de portabilité à implémenté des classes permettant d'accéder aux fichiers de façon indépendante de la plateforme.

Un fichier n'a d'utilité dans un programme que lorsque l'on peut y lire et écrire des informations. Pour ce faire, il est intéressant d'utiliser les flux de données.

Un flux de données est un flux binaire d'information codée qui est indépendant à 100% du logiciel d'exploitation de l'ordinateur principal, du CPU et de l'ordre des octets. Par exemple, un flux de données qui est écrit par un PC sous Windows peut-être lut par un Sun SPARC exécutant Solaris. Il existe plusieurs types de flux de données : des flux basés sur des fichiers, sur des sockets réseau, sur la console de l'application. Mais dans tous les cas, leurs utilisations seront similaires. Ce sont les concepts d'héritage et d'interfaces qui permettent d'obtenir cette abstraction.

Fichiers et flux de données étant par nature appelés à être utilisés conjointement, nous présenterons ce que sont les flux de données puis nous étudierons les deux classes ensembles au travers de plusieurs exemples.

6.1. Fichiers

La classe `QFile` est une unité d'entrée/sortie qui opère sur les fichiers pour la lecture et l'écriture des fichiers binaires et des fichiers textes. Elle peut être employée seule ou plus commodément avec un `QDataStream` permettant de manipuler des données binaires ou un `QTextStream` destiné aux textes. Voici un premier exemple de l'utili-

sation de `QFile` employé seul :

```
1       #include <qapplication.h>
2       #include <qfile.h>
3       #include <qstring.h>
4
5       #include <iostream>
6       using namespace std;
7
8       int main( int argc, char **argv )
9       {
10        QApplication a( argc, argv );
11        QString nomFichier = "qfile.cpp";
12        if( argc > 1 )
13              nomFichier = argv[1];
14        QFile fichier( nomFichier );
15
16        if( !fichier.open( IO_ReadOnly ) )
17        {
18          cout << "Impossible d'ouvrir "
19              "le fichier " << nomFichier
20              << endl;
21          return 0;
22        }
23
24        QString ligne;
25        while( !fichier.atEnd() )
26        {
27           fichier.readLine(ligne, 512);
28           qDebug("%s", ligne.latin1());
29        }
30        fichier.close();
31
32        return 0;
33      }
```

Les paramètres en ligne de commande passés à la fonction `main()` sont ici utilisés afin de déterminer le nom du fichier à ouvrir. Dans tous les programmes écris en C ou C++, `argc` est au moins égal à 1 car le nom du programme est toujours le premier argument transmis dans les paramètres. Dans l'exemple, si `argc` est supérieur à 1, l'argument supplémentaire est récupéré dans `nomFichier`, sinon `nomFichier` garde sa valeur attribuée dans la déclaration.

L'objet `QFile` est déclaré en passant à son constructeur le nom du fichier à ouvrir. Les classes C++ de Qt qui sont bien écrites proposent généralement plusieurs méthodes d'arriver à un résultat. Ainsi pourrions-nous écrire :

```
QFile fichier;
...
if( sexe == 1 )
        fichier.setName( "hommes.txt" );
else
        fichier.setName( "femmes.txt" );
```

N'appelez pas la fonction `setName()` si le fichier est déjà ouvert. Le nom du fichier à ouvrir peut contenir un chemin absolu comme `"/home/durand/toto.txt"`, un chemin relatif comme `"../durand/toto.txt"` ou pas de chemin du tout comme `"toto.txt"`. Le séparateur / fonctionne avec tous les systèmes d'exploitation supportés par Qt.

Lorque l'objet `QFile` est créé, le fichier peut être ouvert afin d'y effectuer des traitements. La fonction `open()` sert à l'ouverture du fichier et renvoie `true` lorsque l'ouverture s'est terminée avec succès, autrement retourne `false`. Un mode d'ouverture doit être passé en paramètre à la fonction `open()`, il doit avoir une ou plusieurs des valeurs suivantes :

- **IO_ReadOnly**: Ouvre le fichier en mode lecture seulement.
- **IO_WriteOnly**: Ouvre le fichier en mode écriture seulement. Si ce mode est employé avec un autre mode, par exemple IO_ReadOnly ou IO_Raw ou IO_Append, le fichier n'est pas tronqué ; mais s'il

est utilisé seul (ou avec IO_Truncate), le fichier est tronqué.

- **IO_ReadWrite**: Ouvre le fichier en mode lecture/écriture, équivalent à (**IO_ReadOnly** | **IO_WriteOnly**).
- **IO_Append**: Ouvre le fichier en mode ajout. (vous devez employer (**IO_WriteOnly** | **IO_Append**) pour mettre le fichier en écriture et pour le placer en mode ajout.). Ce mode est très utile quand vous voulez écrire dans un fichier de log car l'index est placé à la fin du fichier.
- **IO_Truncate**: Tronque le fichier, c'est-à-dire le vide de son contenu et se place au début.
- **IO_Translate**: Permet des retours chariot et la traduction de linefeed pour des fichiers textes sous Windows.
- **IO_Raw**: Accès brut (sans cache) au fichier.

La fonction `atEnd()` utilisé dans l'exemple renvoie `true` si la fin du fichier est atteinte. Retourne `false` s'il reste des données à lire dans le fichier. Une boucle peut donc être exécutée, qui se terminera lorsque la fin du fichier sera atteinte. Dans cette boucle c'est la fonction `readLine()` qui est utilisée pour lire une à une les lignes du fichier et les stocker dans `ligne`. Le deuxième paramètre est obligatoire et indique la longueur maximum de la chaîne à lire. Nous verrons plus loin que `QTextStream` ne possède pas cette contrainte et qu'il est préférable de l'utiliser.

La fonction `qDebug` est utilisée afin d'afficher le contenu de la ligne lue (Voir le chapitre "Techniques de débogage").

Pour fermer le fichier la fonction `close()` est employée. Elle n'a pas d'effet si le fichier n'est pas ouvert.

6.2. Flux de données

Voici une petite présentation des flux de données avec un exemple

n'employant pas de classe Qt :

```
1       #include <iostream>
2
3       int main()
4       {
5         char nom[512];
6         std::cout << "Entrez votre nom : ";
7         std::cin >> nom;
8         std::cout << "Bonjour "
9               << nom << std::endl;
10        return 0;
11      }
```

Les flux de donnée sont utilisables grâce au fichier `iostream`[1] inclus au début du fichier source. Cette bibliothèque fournit quatre objets initialisés au début du programme :

- **cin**: Gère les entrées depuis l'entrée standard, le clavier ou la redirection d'un autre flux. >> (lire depuis) est l'opérateur de lecture associé.
- **cout**: Gère les sorties vers la sortie standard qui est l'écran. << (écrire dans) est l'opérateur d'écriture associé.
- **cerr**: Gère les sorties vers l'unité d'erreur standard, par défaut l'écran.
- **clog**: Gère les messages d'erreur transmis vers la sortie standard, par défaut l'écran.

Le programme commence par afficher un message d'invite sur le sortie standard. La variable `nom` reçoit la saisie du clavier qui se termine lorsque la touche Entrée est enfoncée. La ligne 8 permet de constater

[1]En rajoutant `using namespace std;` en ligne 2, nous pourrions écrire cin et cout à la place de std::cin et std::cout.

que différentes sorties peuvent être enchaînées sur la même ligne de commande et que même le contenu de variables peut être redirigé.

Voici maintenant un exemple semblable employant cette fois la classe de flux texte `QTextStream` :

```
1       #include <qapplication.h>
2       #include <qtextstream.h>
3       #include <qstring.h>
4
5       int main( int argc, char **argv )
6       {
7         QApplication a( argc, argv );
8         QString chaine;
9         QTextStream flux(chaine,
10              IO_WriteOnly );
11
12        flux << "Resultat " << 2+3;
13        qDebug("Combien font 2+3 ? %s",
14              chaine.latin1() );
15        return 0;
16      }
```

Le résultat produit sur la console est : `Combien font 2+3 ? Resultat 5`.

Le classe `QTextStream` de Qt fournit des fonctions pour lire et écrire dans un `QIODevice` qui est la classe de base des dispositifs d'entrée/sortie. `QTextStream` à un fonctionnement très similaire aux classes iostream du C++ standard que nous venons de voir. Elle peut écrire et lire dans des chaînes (`QString`), dans des tableaux d'octets (`QByteArray`) et dans des fichiers (`QFile`). Elle permet également d'ouvrir un flux en recevant un pointeur de type `QIODevice`, ce qui signifie qu'elle peut traiter également les objets `QSocket` et `QBuffer` qui en héritent. Lorsque `QTextStream` est utilisé avec un `QString`, le mode d'ouverture doit être précisé. Les modes d'ouvertures sont détaillés dans la section consacrée aux fichiers, car identiques pour les

deux classes. Le flux est redirigé dans le `QString chaine` qui contient après l'exécution de la ligne 12 : `"Le resultat est 5"`.

En plus des caractères et des chaînes, `QTextStream` supporte les types numériques du C++ qu'il converti depuis et vers des chaînes de caractères. Ainsi dans l'exemple précédent, le résultat de l'addition de 2+3 a été inséré dans le flux sous forme de chaînes de caractères.

```
1       #include <qapplication.h>
2       #include <qtextstream.h>
3       #include <qstring.h>
4
5       #include <iostream>
6
7       using namespace std;
8
9       int main( int argc, char **argv )
10      {
11        QApplication a( argc, argv );
12        QString chaine;
13        QTextStream fluxEcriture(chaine,
14              IO_WriteOnly );
15
16        fluxEcriture << "Resultat: " << 2+3;
17        cout << "Combien font 2+3 ? "
18              << chaine << endl;
19
20        QTextStream fluxLecture(chaine,
21              IO_ReadOnly );
22        QString c;
23        int n;
24        fluxLecture >> c >> n;
25        qDebug("c :%s et n:%d",
26              c.latin1(), n);
27        return 0;
28      }
```

Cet autre exemple utilise deux flux. Le premier (`fluxEcriture`) ouvert en écriture redirige une chaîne de caractères et le résultat de l'addition dans le `QString chaine`. Le deuxième flux est créé en lecture sur `chaine`. Ce flux est redirigé dans un `QString` et dans un entier. Cela démontre que `QTextStream` sait reconnaître les types de données et les convertir depuis une chaîne de caractères (`chaine`) vers un type différent (ici l'entier `n`).

6.3. Utiliser ensemble Fichiers et Flux de données.

Voici maintenant les classes `QFile` et `QTextStream` utilisées conjointement. Pour les besoins de l'exposé un fichier `qfile.cpp` contenant le code suivant est créé dans un répertoire:

```
1       #include <qapplication.h>
2       #include <qfile.h>
3       #include <qtextstream.h>
4       #include <qstring.h>
5
6       #include <iostream>
7       using namespace std;
8
9       int main( int argc, char **argv )
10      {
11        QApplication a( argc, argv );
12
13        QFile fichier( "qfile.cpp" );
14
15        if( !fichier.open( IO_ReadOnly ) )
16        {
17          cout << "Impossible d'ouvrir "
18              "le fichier." << endl;
19          return 0;
20        }
21
```

```
22        QTextStream stream( &fichier );
23        QString ligne;
24        while( !stream.atEnd() )
25        {
26           ligne = stream.readLine();
27           cout << ligne << endl;
28        }
29        fichier.close();
30
31        return 0;
32      }
```

Pensez à exécuter dans ce répertoire les commandes `qmake -project && qmake && make`. Ce programme permet d'ouvrir un fichier puis d'afficher son contenu ligne par ligne dans la console. Voyons maintenant en détail le contenu du source :

Un `QFile` est construit en lui transmettant le nom du fichier à traiter. Puis ce fichier est ouvert en lecture seule grâce au paramètre `IO_ReadOnly`. Si l'ouverture échoue, un message d'erreur est affiché sur la sortie standard puis le déroulement du programme est interrompu en ligne 19.

Un `QTextStream` est ensuite créé avec le fichier ouvert. C'est ici le mode d'ouverture du fichier et non le `QTextStream` qui détermine les opérations autorisées. Comme le fichier est ouvert en lecture seulement, une tentative d'écriture avec par exemple : `stream << "toto" ;` ne fonctionnerait pas et le message d'erreur `QFile::writeBlock: Write operation not permitted` serai affiché[1]. Une fois le flux initialisé, une boucle lit le fichier du début à la fin. Elle se termine lorsque `atEnd()` retourne `TRUE` indiquant que la fin du fichier est atteinte. Chaque ligne lue est stockée dans le `QString ligne` puis redirigée sur la sortie standard. `readLine()` supprime lors de la lecture le

[1]Notez que les messages d'erreurs ne sont visibles que dans une console. Même avec un programme graphique, il est donc utile, en phase de développement, de le lancer à partir d'une console.

retour chariot de fin de ligne, il doit donc être rajouté lors de l'affichage grâce à `endl`.

Lorsque les données à lire ou écrire dans un fichier ne sont pas des lignes de texte, mais des blocs de données, la classe `QDataStream` devra être employée. Elle permet de créer des fichiers de données brutes contenant des enregistrements de blocs-mémoires. Cette notion qui dépasse le cadre de l'initiation n'est pas traitée dans cet ouvrage. En revanche, le lecteur trouvera dans la documentation de Trolltech toutes les explications nécessaires.

Résumé

Nous avons vu dans ce chapitre comment manipuler les fichiers en lecture ou écriture en utilisant les flux de données.

7. Quelques classes intéressantes

7.1. QString

`QString` permet de manipuler des chaînes de caractères encodées en Unicode. L'utilisation de l'Unicode permet l'encodage des alphabets du monde entier (ou presque). Nous n'allons étudier dans QString que les propriétés les plus utiles, c'est-à-dire celles qui servent chaque jour.

Voici quelques exemples d'utilisation de `QString` :

- Un `QString` vide est déclaré, sa propriété `isEmpty()` est alors à `true`. Ensuite la chaîne de caractère "Durand" lui est affecté:

```
QString nom;
nom = "Durand";
```

- Il reçoit ici une valeur transmise dans le constructeur:

```
QString nom("Dupond");
```

- Il est possible de concaténer une chaîne de caractères et un `QString`:

```
QString nom("Dupond");
QString affichage = "Monsieur " + nom;
```

- Conversion de types numériques vers un `QString`:

```
QString total;
int i=5, j=3;
total = QString().setNum(i+j);
```

Une autre possibilité est d'utiliser la fonction statique `number`. Ainsi aucun objet `QString` n'est alloué en mémoire pour la conversion.

```
QString total;
int i=5, j=3;
total = QString::number(i+j);
```

- Et d'un `QString` vers des types numériques:

```
QString pi = "3.1415926";
double PI;
PI = pi.toDouble();
```

- Fabriquer un `QString` en lui transmettant des arguments:

```
QString nom = "Durand";
QString prenom = "Paul";
int age = 36;
QString affichage = QString( "Monsieur %1 %2:
%3 ans" )
        .arg( nom )
        .arg( prenom )
        .arg( age );
```

`affichage` contient "Monsieur Durand Paul: 36 ans".

- Chercher une occurrence d'une chaîne ou d'un caractère dans un `QString`:

```
QString chaine( "canard" );
        int i = chaine.find( QRegExp("an"), 0
```

```
);    // i == 1
```

- Deux `QString` peuvent être comparés avec l'opérateur ==:

```
QString premier( "Durand" );
        QString deuxieme ( "Dupond" );
        if ( premier == deuxieme ) // Ici
c'est faux
        {
                ...
        }
        else
        {
                ...
        }
```

- Extraction de sections de la chaîne:

```
QString chiffre;
int i=0;
QString ligne( "1 5 6 8 9 7 10" );
do
{
        chiffre =
                ligne.section( ' ', i, i );
        int n = chiffre.toInt();
        i++;
} while( !chiffre.isEmpty() );
```

  Dans cet exemple, ligne contient des nombres séparés du suivant par un espace. Chaque partie de la chaîne est extraite et mise dans chiffre. Chiffre est ensuite converti en entier et affecté à n. On boucle tant que chiffre contient un QString valide.

- Contrôler si un QString est vide ou nul:

```
QString a( "" );
a.isEmpty();        // TRUE
a.isNull();         // FALSE

QString b;
b.isEmpty();        // TRUE
b.isNull();         // TRUE
```

La classe `QString` dispose de dizaines de fonctions différentes afin de travailler avec les chaînes de caractères. Cette petite liste représente celles qui sont utilisées le plus souvent. L'étude de la documentation permettra une fois de plus d'obtenir la liste complète des fonctions.

7.2. Les modèles de classes

La QTL (Qt Template Library) est un ensemble de modèles fournissant des conteneurs d'objets. Elle est équivalente à la STL et peut être utilisée si cette dernière n'est pas disponible sur votre plate-forme de développement.

7.2.1. QValueList

La classe `QValueList` permet de créer et manipuler des listes de valeurs. L'emploi des modèles permet de gérer des objets les plus variés allant des types de données simples aux classes complexes. L'insertion dans la liste est effectuée par copie de l'élément inséré.

```
1       #include <qvaluelist.h>
2
3       int main( int argc, char **argv )
4       {
5               typedef QValueList<int>
6                       ListeEntier;
```

```
7               ListeEntier liste;
8               liste.append( 5 );
9               liste.append( 15 );
10              liste.append( 28 );
11              ListeEntier::iterator it;
12              for ( it = liste.begin();
13                      it != liste.end(); ++it
)
14                      qDebug("Valeur :%d",
15                              (*it));
16      }
```

Dans cet exemple `QValueList` est employé sur un type simple. Le type `ListeEntier` est d'abord défini comme liste d'entiers. La liste nommée `liste` est ensuite déclarée. Elle va recevoir dans les trois lignes suivantes des entiers. La ligne 10 crée un itérateur sur notre liste. Un itérateur est destiné à recevoir un pointeur sur un élément de liste.

Une boucle est installée, elle permet grâce à l'emploi de l'itérateur de parcourir la liste du premier au dernier élément. `it` reçoit d'abord le premier élément (`liste.begin()`). A chaque passage de boucle, on vérifie si la fin de liste n'est pas atteinte (`liste.end()`). L'itérateur est ensuite incrémenté ce qui a pour effet de le faire pointer sur l'élément suivant de la liste. Enfin, la valeur pointée par `it` est affichée à l'écran (`(*it)`).

```
1       #include <qvaluelist.h>
2       class Personne
3       {
4       public:
5               Personne() :
6                       _nom(0), _age(0) {};
7               Personne(QString n,
8                       int a):_nom(n), _age(a)
{};
```

```
9               int age() { return _age; };
10              void setAge(int a) { _age =
a; };
11              QString nom() { return _nom; }
12              void setNom(QString n) { _nom =
n; };
13      private:
14              QString _nom;
15              int _age;
16      };
17
18      int main( int argc, char **argv )
19      {
20              typedef QValueList<Personne>
ListePersonne;
21              ListePersonne liste;
22              liste.append( Personne("Durand",
32 ) );
23              liste.append( Personne("Dupond",
25 ) );
24              Personne pers;
25              pers.setNom( "Martin");
26              liste.append( pers );
27              pers.setAge( 50 );
28              ListePersonne::iterator it;
29              for ( it = liste.begin(); it !=
liste.end(); ++it )
30                      qDebug("Nom :%s Age
:%d",
31
(*it).nom().latin1(),
32                                  (*it).age() );
33      }
```

Cet autre exemple est un peu plus évolué puisqu'il gère une liste d'objets de type classe `Personne`. Cette classe possède des données (`_nom` et `_age`) ainsi que des méthodes publiques renvoyant ces valeurs et permettant de les mettre à jour.

Comme dans le premier exemple, une liste est déclarée (`ListePersonne`) puis deux objets sont insérés dans cette liste.

Les lignes 22 à 25 permettent de mettre en avant que les objets sont dupliqués lorsqu'ils sont insérés dans la liste. En effet, le code de la ligne 25 (`setAge`) modifie l'objet `pers` qui est maintenant différent de celui inséré dans la liste. Lors de l'affichage, l'âge de Martin est resté à 0.

```
Nom :Durand Age :32
Nom :Dupond Age :25
Nom :Martin Age :0
```

Pour modifier les éléments à l'intérieur de la liste, il est possible d'utiliser l'itérateur.

```
1       ListePersonne::iterator it;
2               it = liste.begin();
3               (*it).setNom( "Joe" );
```

7.2.2. QValueVector

`QValueVector` permet de créer et manipuler des tableaux dynamiques (vecteurs). L'insertion dans le vecteur est effectuée par copie de l'élément inséré.

```
1       #include <qvaluevector.h>
2       class Personne
3       {
4       public:
5               Personne() : _nom(0), _age(0)
{};
```

```
6               Personne(QString n, int
a):_nom(n), _age(a) {};
7               int age() { return _age; };
8               void setAge(int a) { _age =
a; };
9               QString nom() { return _nom; }
10              void setNom(QString n) { _nom =
n; };
11      private:
12              QString _nom;
13              int _age;
14      };
15
16      int main( int argc, char **argv )
17      {
18              typedef QValueVector<Personne>
ListePersonne;
19              ListePersonne vecteur(2);
20              vecteur[0] = (
Personne("Durand", 32 ) );
21              vecteur[1] = (
Personne("Dupond", 25 ) );
22              Personne pers;
23              pers.setNom( "Martin");
24              vecteur.append( pers );
25              pers.setAge( 50 );
26              ListePersonne::iterator it;
27              for ( it = vecteur.begin();
28                      it != vecteur.end();
29                      ++it )
30                      qDebug("Nom :%s Age
:%d",
```

```
31
(*it).nom().latin1(),
32                                  (*it).age() );
33      }
```

Dans cet exemple une classe `Personne` est manipulée dans le vecteur. Ligne 19, un vecteur de deux éléments est créé. Les deux affectations dans le vecteur sont effectués lignes 20 et 21 en utilisant les indices des éléments du vecteur à modifier. Un élément de vecteur comme un élément de liste (`QValueList`) peut-être ajouté en employant `append()`. Et comme pour `QValueList` les objets affectés ou insérés le sont par recopie comme le montre la sortie du programme. En effet, l'age de Martin est resté à 0.

```
Nom :Durand Age :32
Nom :Dupond Age :25
Nom :Martin Age :0
```

7.2.3. QPtrList

La classe `QPtrList` est un modèle de classe permettant de manipuler des listes. Définissez une instance de classe QPtrList<X> pour créer une liste qui fonctionne avec des pointeurs sur X (X*).

```
1       #include <qstring.h>
2       #include <qptrlist.h>
3
4       class Personne
5       {
6       public:
7               Personne() : _nom(0), _age(0)
{};
8               Personne(QString n, int
a):_nom(n), _age(a) {};
```

```
9               int age() { return _age; };
10              void setAge(int a)
11                      { _age = a; };
12              QString nom() { return _nom; }
13              void setNom(QString n)
14                      { _nom = n; };
15      private:
16              QString _nom;
17              int _age;
18      };
19
20      int main()
21      {
22              QPtrList<Personne> liste;
23              liste.append(
24                      new Personne("Durand",
32 )
25              );
26              liste.append(
27                      new Personne("Dupond",
25 )
28              );
29              Personne *pers = new Personne;
30              pers->setNom( "Martin");
31              liste.append( pers );
32              pers->setAge( 50 );
33              Personne *personne;
34              for ( personne = liste.first();
35                      personne;
36                      personne = liste.next()
)
37                      qDebug("Nom :%s Age
:%d",
```

```
38
personne->nom().latin1(),
39                                  personne->age()
);
40      }
```

La même classe `Personne` que précédemment est utilisée. Un objet `QPtrList` sur un type `Personne` est tout d'abord créé.

Ce type de liste gère des pointeurs. Pour cette raison, les deux paramètres transmis à la fonction `append()` lignes 21 et 22 sont des pointeurs renvoyés par `new`. Nous aurions également pu utiliser:

```
Personne *nouvellePersonne =
        new Personne("Marchand", 23);
liste.append( nouvellePersonne );
```

L'instance de classe pour le nom "Martin" est insérée dans la liste puis l'age est modifié. Comme la liste gère des pointeurs, cette modification est prise en compte. En effet, l'adresse pointée par `pers` est la même que celle insérée dans la liste. C'est vérifié dans la sortie effectuée par le programme:

```
1       Nom :Durand Age :32
2       Nom :Dupond Age :25
3       Nom :Martin Age :50
```

8. Utiliser Qt Designer

Ce chapitre présente `Qt Designer`, le concepteur d'interface de Qt. Le lecteur ayant consulté les autres chapitres de ce livre à remarqué que `Qt Designer` n'y a jamais été utilisé. En effet, chacune des interfaces montrées en exemple a été construite directement en étant codée dans l'application. C'est justifié par le fait que lorsque l'on est en mesure de réaliser des interfaces en les codant entièrement, l'utilisation de `Qt Designer` devient un jeu d'enfant. L'inverse n'est pas vraie, en effet il est possible d'élaborer des programmes complets sans, par exemple, bien connaître le principe de fonctionnement des layouts. C'est pour cette raison que `Qt Designer` n'a jamais été employé et que seul un chapitre lui est consacré. Le lecteur va vite réaliser, en lisant ce chapitre, que son emploi accélère fortement le temps de développement en fournissant la possibilité de créer rapidement des interfaces complexes et agréables à utiliser.

Dans les versions précédentes de Qt (2.x), `Qt Designer` était uniquement un concepteur d'interface permettant de rendre agréable la conception des interfaces utilisateurs par l'emploi d'un programme graphique. `Qt Designer` lit et écrit des fichiers ayant l'extension (.ui) et contenant du XML. Le compilateur d'interface utilisateur `uic` est employé pendant le processus de construction d'une application pour produire du code C++ à partir de ces descriptions XML (voir la section uic).

Avec Qt 3.x, `Qt Designer` est plus qu'un simple éditeur de dialogue. Il permet, en plus de nouvelles caractéristiques comme la capacité de créer des fenêtres principales et des actions :

- La gestion de projet pour la partie interface utilisateur de votre application.
- L'édition de code permettant d'entrer le code des slots directement. Ce code est enregistré dans des fichiers (.ui.h) et supprime la nécessité de créer des sous-classes à partir des classes d'interfaces utilisateur générées par `Qt Designer`. Néanmoins, la possibilité de créer des sous-classes dans votre programme existe toujours.
- Le chargement dynamique des formes permet le chargement des fichiers (.ui) lors de l'exécution du programme. Cependant, ce point ne sera pas étudié dans cet ouvrage.

Deux façons d'utiliser Qt Designer.

1 En concepteur d'interface seulement. Des fichiers projets peuvent être créés, des interfaces rajoutées et les slots à exécuter sont définis avec cette méthode. En revanche aucun code n'est saisi dans `Qt Designer`. Le développeur devra ajouter à son programme une sous-classe de celle générée par le couple `Qt Designer` et `uic`.

2 La deuxième méthode, vous l'aurez compris, consiste à rentrer tout ou partie du code de la fenêtre d'interface dans `Qt Designer`. Dans ce cas, un fichier (.ui.h) est généré pour y stocker le code. Ce fichier qui porte le même préfixe que l'interface (par exemple `dialogue.ui` et `dialogue.ui.h`) est compilé en même temps que le projet. Cette manière de procéder, si elle permet d'économiser la création d'une classe dérivée, est moins souple que la première. Le constructeur et le destructeur de la classe ne sont pas accessibles dans `Qt Designer`. Deux fonctions, respectivement `init()` et `destroy()` sont appelées en lieu et place du contructeur/destructeur mais ne permettent pas, par exemple, de définir des paramètres ayant des valeurs par défaut.

Néanmoins, les deux méthodes sont opérationnelles et choisir l'une ou l'autre est l'affaire de chacun.

Voici une présentation rapide des fenêtres présentes dans l'éditeur:

- **"Toolbox"** propose, rangés par catégories, l'ensemble des widgets pouvant être disposés dans les interfaces. Lorsqu'une fenêtre est ouverte, disposer un nouveau widget s'effectue en cliquant sur le type désiré puis sur la fenêtre ce qui a pour effet de rajouter ce

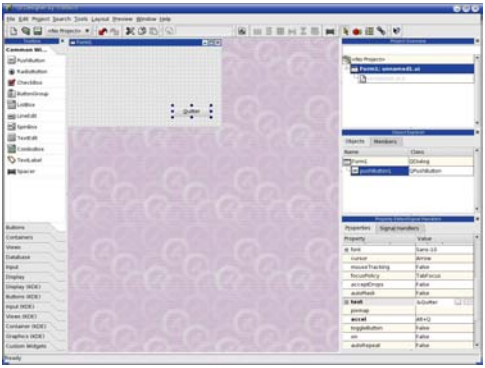


Figure 8.1. Qt Designer en création de dialogue.

widget à la fenêtre.

- **"Project Overview"** contient le nom de votre fichier projet ainsi que le nom des fichiers appartenant au projet. Un clic droit sur le nom d'un fichier permet de le supprimer du projet ou de l'ouvrir pour l'éditer.

- **"Property Editor/Signal Handlers"** et surtout son onglet **"Properties"**. Il permet de modifier les propriétés de la forme ou du widget sélectionné. Ne sont présentes dans cette fenêtre que les propriétés modifiables des objets. Chaque type de widget propose donc une liste différente adaptée à ses caractéristiques.

- **"Object Explorer"** présente dans un arbre les widgets de la fenêtre en cours d'édition. Sont affichés le nom et le type du widget. En cliquant sur un objet dans l'explorateur, celui-ci est sélectionné dans le dialogue édité ce qui peut être utile lorsque les widgets sont très proches les uns des autres. Un clique droit permet d'effectuer

un copier, couper ou coller de l'objet dans la fenêtre.

8.1. Création et gestion de projets

Le gestion de projet avec `Qt Designer` propose de créer puis gérer le contenu des fichiers projets (.pro). Ces fichiers au contenu identique à ceux créés à la main doivent être interprétés par l'utilitaire `qmake` afin de produire des fichiers Makefile adaptés à la plate-forme de développement. Une section ayant été consacrée à l'utilisation de `qmake`, nous ne reviendrons pas sur le sujet.

Nous allons maintenant générer un nouveau projet :

La création s'effectue en choisissant le menu "File|New" puis "C++ Project" dans le boîte de dialogue affichée. Dans le dialogue "Project Settings" donnez un nom à votre projet. Je choisis de le nommer `monprojet`. De préférence, placez ce nouveau fichier dans un répertoire sur disque spécialement créé pour le projet. Les options de l'onglet C++ conviennent à une application standard et ne seront pas étudiées ici. Reportez-vous à la section traitant de `qmake` pour plus de détails les concernant.

Les entrées du menu "Project" permettent d'agir sur les projets :

- **Active project** permet de changer de projet actif, ce qui signifie qu'il est possible de gérer simultanément plusieurs projets en basculant de l'un à l'autre.
- **Add File…** donne la possibilité d'ajouter au projet des fichiers existants. Ce sont généralement des fichiers d'en-têtes (.h), des fichiers d'implémentations (.cpp) ou des fichiers d'interfaces (.ui). Ce choix de menu ne permet pas de créer de nouveaux fichiers qui devront l'être par le menu "File|New".
- **Image collection…** est utilisé pour créer une collection d'images (voir la section correspondante dans ce chapitre).

- **Database connections…** qui ne sera pas étudié dans cet ouvrage permet de gérer les connexions aux bases de données.
- **Project Setting…** quant à lui sert à modifier les options du projet courant. On y retrouve les mêmes options que lors de la création du projet.

Notez qu'il est possible de créer ou ouvrir un fichier d'interface (.ui) sans pour cela générer ou ouvrir de fichier projet.

8.2. Dialogues

Cette partie va nous permettre d'étudier la création de dialogues avec `Qt Designer`. L'intérêt majeur de `Qt Designer`, outre d'éviter de fastidieuses saisies de code est de permettre de visualiser immédiatement le comportement de ses dialogues, notamment en redimensionnant ceux-ci pour tester leurs comportements.

8.2.1. Nouveaux dialogues

Choisissez le menu "File|New" et ajoutez un `Dialog` à votre projet. Vérifiez pour cela que la liste déroulante affiche bien le nom de votre projet. Un dialogue ayant été ajouté au projet, son nom doit être présent dans la fenêtre "Project Overview".

Un dialogue vide s'affiche dans `Qt Designer`. Nous pouvons déjà changer si nous le désirons ses dimensions avec la souris et saisir quelques champs dans la fenêtre Property Editor:

Les champs `name` et `caption` peuvent être renseignés : Le premier, comme pour les objets permet de donner un nom à la forme. Le deuxième modifie le texte affiché dans la barre de titre de la fenêtre.

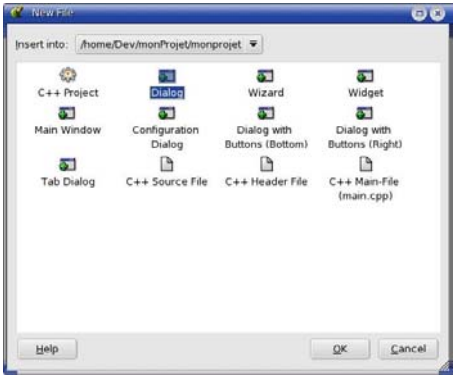


Figure 8.2. Création d'un fichier dans `Qt Designer`.

Donner un nom de classe à la forme.

Les formes créées dans `Qt Designer` sont, lors de la phase de compilation, transformées par `uic` en classes C++. Le nom de la classe qui sera produite est par défaut le même que la propriété `name` du dialogue. Il est néanmoins souhaitable de vérifier ce nom dans "Edit|Form Settings" et de le modifier[1] le cas échéant.

8.2.2. Placer des widgets

Poser de nouveaux widgets dans le dialogue est assez simple. Il suffit de cliquer une première fois sur le type d'objet choisi dans la fenêtre "Toolbox" puis sur le dialogue (le curseur à une forme de croix). Le

[1]Comme pour les noms d'objets, donnez de préférence un nom parlant à votre classe.

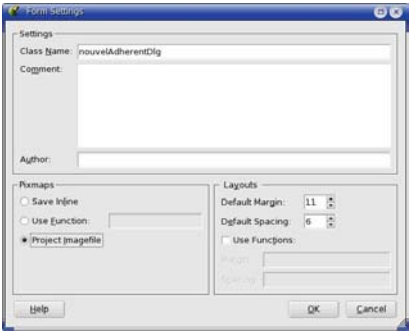


Figure 8.3. Modification du nom de la classe générée.

nouveau widget s'affiche dans le dialogue à l'endroit cliqué. Toutes ses propriétés peuvent être changées dans la fenêtre "Property Editor".

Une propriété présente pour tous les objets est `name`. Ce nom permet, dans les programmes de faire référence à l'objet. Le développeur pourra conserver, pour certains objets le nom attribué par `Qt Designer`. Par exemple, un label affichant un titre qui ne sera jamais modifié peut garder son nom `textLabel1` attribué par défaut. En revanche, si par exemple, un bouton doit être relié à un `slot` dans votre programme, il sera plus parlant d'avoir dans ce dernier:

```
connect(boutonValider, SIGNAL(clicked()),
        this, SLOT(slotEnregistre()));
```

plutôt que:

```
connect(pushButton2, SIGNAL(clicked()),
        this, SLOT(slotEnregistre()));
```

Ainsi le bouton devrait avoir sa propriété `name` renseignée avec `boutonValider`.

L'onglet Container de la fenêtre "Toolbox" propose une catégorie particulière de widget. En effet, ils sont destinés à contenir d'autres widgets.

Placez par exemple sur le dialogue un ButtonGroup. Tout en vérifiant qu'il est sélectionné, mettez à l'intérieur deux boutons. Ces deux boutons ont comme parent le ButtonGroup. Cela peut être vérifié de deux façons :

1 Dans l'arbre d'objets de la fenêtre "Object Explorer" où les deux boutons sont bien enfants du conteneur.

2 En déplaçant le conteneur sur le dialogue. Vous constatez que les boutons se déplacent avec lui.

Ainsi en changeant les propriétés du conteneur, on peut agir sur les widgets qu'il contient. Par exemple si la propriété `enabled` du `ButtonGroup` est positionnée à `false`, c'est l'ensemble, contenu compris qui est désactivé.

Notre dialogue étant toujours vide (supprimez éventuellement les widgets précédents), placez à l'intérieur un label, un lineEdit, un buttonGroup avec à l'intérieur deux radioButton. Enfin deux boutons, comme sur la figure ci-dessous. Voici un tableau récapitulatif des objets de

Figure 8.4. Notre dialogue Adhérent.

notre dialogue ainsi que les propriétés à modifier.

Type	name	caption	Class Name
QDialog	AdhesionDlg	Inscription d'un adhérent	inscriptionDlg
QLabel	labelNom		
QLineEdit	nom		
QGroupBut-ton	groupSexe		
QRadioBut-ton	masculin		
QRadioBut-ton	feminin		
QPushButton	boutonValider		
QPushButton	boutonAnnu-ler		

La propriété `checked()` d'un des deux boutons radios doit être mise à `True`.

Sauver le dialogue si ce n'est encore fait dans le répertoire du projet `monprojet` et nommez-le par exemple `adhesionDlg.ui`.

Vous pouvez tester le dialogue en choisissant le menu "Preview|Preview Form". Si vous tentez de redimensionner la fenêtre, les widgets restent à leur place, car ils n'ont pas été disposés dans des layouts.

8.2.3. Disposer les objets dans des Layouts

`Qt Designer` propose l'utilisation de layouts pour organiser la disposition des widgets.

Qt fournit trois layouts: `QHBoxLayout` qui permet de disposer les widgets horizontalement de la gauche vers la droite. `QVBoxLayout` qui les aligne verticalement du haut vers le bas. Enfin `QGridLayout` permettant de les disposer dans une grille.

Les layouts peuvent contenir des widgets ou d'autres layouts. Les layouts ne sont pas des widgets et à ce titre sont invisibles lors de l'exécution du programme.

Nous allons voir dans cette section la disposition des widgets grâce aux layouts en mode graphique. Pour plus de détails reportez-vous à la section "Position avec les layouts". Les trois types de `layouts`

Figure 8.5. La barre d'outils Layout.

sont présents dans la barre d'outils ainsi que d'autres éléments, dont le `QSpacerItem`.

Sélectionnez maintenant le label et la ligne de saisie. Vous pouvez les sélectionner à la souris au lasso ou l'un après l'autre en enfonçant la touche `Shift`. Cliquez maintenant sur le bouton représentant le layout horizontal. Un rectangle rouge est dessiné autour des deux widgets indiquant qu'ils sont maintenant intégrés dans un layout.

Placez maintenant un spacer horizontal à droite du groupe de boutons radio puis en sélectionnant le groupe et le spacer insérez autour un layout horizontal.

Ajoutez un spacer vertical au dessus des QPushButton puis regroupez les trois widgets dans un layout vertical.

Terminez enfin les dispositions en cliquant sur la forme puis en y insérant un layout grille. Vous devriez obtenir un résultat similaire à la figure ci-dessous: En affichant un nouvel aperçu du dialogue par `Ctrl+T` ou par le menu "Preview|Preview Form", celui-ci se comporte correctement lorsqu'il est agrandi ou réduit.

Vous pouvez jouer avec la propriété `sizeType` du spacer afin de tester

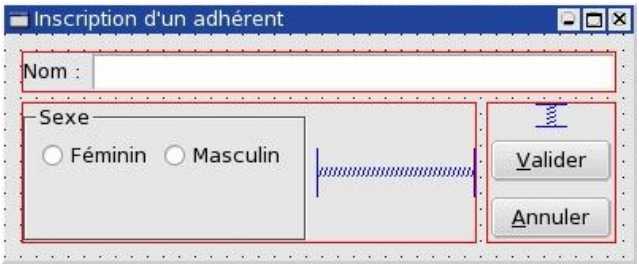


Figure 8.6. Les layouts positionnés.

son comportement en aperçu avec les différentes valeurs.

Tout ce que nous venons de faire aurait pu être réalisé en le codant entièrement dans l'application comme dans les chapitres précédents. Il faut néanmoins reconnaître qu'il est plus agréable et rapide d'utiliser `Qt Designer`.

8.2.4. Relier Signaux et Slots

Nous allons étudier dans cette partie comment relier à des slots les signaux émis par nos widgets. Ces slots peuvent être de deux types:

- Le `slot` appartient à une classe Qt. Par exemple, le `slot reject()` que nous allons utiliser dans notre exemple appartient à la classe `QDialog`. La seule chose que nous avons à faire est de le connecter au `signal` d'un objet.

- Nous désirons faire référence à un `slot` que nous allons coder nous-mêmes. Avant de le connecter dans `Qt Designer`, nous avons besoin de le définir. C'est à dire indiquer son nom, le type de sa valeur de retour ainsi que de ses paramètres. De plus, si nous avons choisi d'entrer le code des slots en utilisant des fichiers (.ui.h), nous saisirons le code dans `Qt Designer`.

Notre dialogue comporte deux boutons, le premier, `boutonValider` va déclencher un `slot` "personnalisé". Le deuxième `boutonAnnuler` le `slot` `reject()` de la classe `QDialog`.

Commençons par définir notre `slot` personnalisé : La boîte

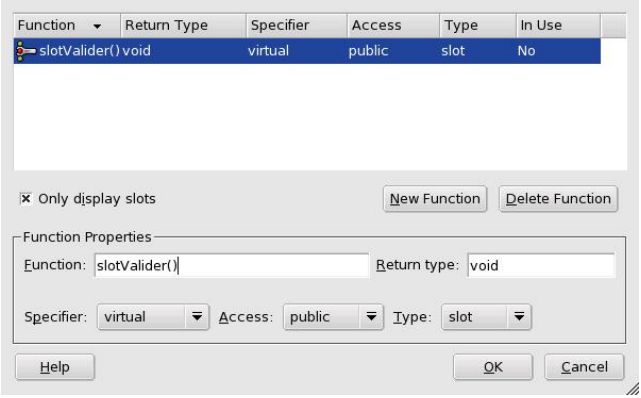


Figure 8.7. La définition des slots.

de dialogue de création des slots s'ouvre en choisissant le menu "Edit/Slots…". En cliquant sur "New Function" une ligne est insérée. Changez maintenant son nom pour `slotValider()` et les autres propriétés comme sur la figure ci-dessus. Validez ensuite le dialogue.

Notre `slot` défini, nous pouvons à présent effectuer la connexion aux signaux. Nous avons pour ce faire deux méthodes à notre disposition:

1 En ouvrant le dialogue de connexion accessible dans le menu "Edit/Connections…". Une fenêtre vide apparaît, chacun des slots doit être créé en cliquant sur le bouton "New" puis en renseignant les quatre paramètres de la connexion (objet émetteur, `signal` à

surveiller, objet receveur, `slot` à exécuter).

2 En cliquant sur le bouton "Connect Signal/Slots…" de la barre d'outils "Tools" ou encore en appuyant sur F3. Lorsque c'est fait, le curseur de la souris fait apparaître une croix au dessus des widgets. Cliquez sur le bouton `boutonValider` puis, en maintenant le bouton de la souris enfoncé, tirez le curseur en dehors du dialogue. La même fenêtre que précédemment s'affiche pour connecter les objets. Néanmoins, cette fois l'émetteur et le receveur ont été renseignés respectivement avec le nom du widget cliqué et avec celui du dialogue. Il ne reste plus qu'à choisir le `signal` à surveiller et le `slot` à déclencher.

Pour le bouton `boutonAnnuler`, recommencez en employant indifféremment la méthode 1 ou 2 et en lui connectant le `signal` `reject()`.

Figure 8.8. La connexion des signaux et des slots.

Après validation, testez de nouveau le dialogue avec `Ctrl+T`. Si vous cliquez sur le bouton Annuler, le dialogue se ferme. Cela signifie que les

slots appartenant aux classes Qt sont opérationnels en mode prévisualisation dans `Qt Designer`.

Le `slot slotValider()` est maintenant connecté, mais son implémentation n'ayant pas été faite, aucun code n'est exécuté. C'est maintenant qu'il faut décider d'opter pour la saisie des slots dans `Qt Designer` ou de créer une sous-classe dans votre programme. Nous allons maintenant étudier les deux méthodes.

8.2.5. L'éditeur de code de Qt Designer

L'onglet "Members" de la fenêtre "Object Explorer" permet de définir et de modifier les variables et fonctions membres de la classe, y compris les slots. Le `slot` que nous avons défini plus haut, `slotValider()` y est donc présent. Un clic droit sur celui-ci affiche un menu popup permettant de choisir "Goto implementation". Confirmez la création du fichier `adhesionDlg.ui.h` si la question est posée. Nous pouvons à présent entrer le code à exécuter dans le `slot` :

```
#include <qmessagebox.h>
#include <qstring.h>
#include <qradiobutton.h>
#include <qlineedit.h>

void AdhesionDlg::slotValider()
{
QString Sexe;
if( masculin->isChecked() )
        Sexe = "Monsieur ";
else
        Sexe = "Madame ";
QMessageBox::information( this, "Fichier
adhesion",
Sexe+nom->text()+" inscrit(e)" );

}
```

Dans ce slot, nous testons quel bouton radio est sélectionné afin d'afficher un message adapté. Une boîte d'information est ensuite affichée pour confirmer l'adhésion de la personne. Afin de pouvoir compiler et exécuter le programme, nous avons besoin d'un fichier `main.cpp` que voici:

```
#include <qapplication.h>
#include "adhesionDlg.h"

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

  AdhesionDlg *adhesion = new AdhesionDlg(0);

  a.setMainWidget( adhesion );
  adhesion->show();
  return a.exec();
}
```

Pensez à rajouter ce fichier `main.cpp` au projet grâce au menu "Project/Add Files…".

Lancez dans une console, ouverte dans le répertoire du projet : `qmake && make` afin de générer un fichier Makefile puis de le compiler. La compilation devrait se dérouler sans souci. Vous pouvez donc exécuter le programme obtenu. Tout est maintenant fonctionnel : les widgets suivent le redimensionnement de la fenêtre et les deux boutons `boutonValider` et `boutonAnnuler` valident ou annulent la saisie de l'adhésion. Avec cette méthode, la classe obtenue après compilation, `adhesionDlg` est directement instanciée dans la fonction `main()`. Si nous désirons ajouter des slots dans notre fenêtre, nous devrons le faire de nouveau dans `Qt Designer`.

8.2.6. Sous-classer une forme.

Avec cette autre manière de procéder, aucun code n'est saisi dans `Qt`

Figure 8.9. Notre dialogue en exécution.

`Designer`. Reprenons comme exemple la forme `adhesionDlg.ui` créée dans `Qt Designer` mais sans code saisi et donc sans fichier `adhesionDlg.ui.h`. La propriété `name` et le nom de classe contiennent `adhesionDlg`. Lors de la phase de compilation, `uic` transformera le fichier d'interface (.ui) et fabriquera deux fichiers : `adhesionDlg.h` pour la définition de classe :

```
#ifndef ADHESIONDLG_H
#define ADHESIONDLG_H

#include <qvariant.h>
#include <qdialog.h>

class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class QLabel;
class QLineEdit;
class QButtonGroup;
class QRadioButton;
class QPushButton;
```

```
class adhesionDlg :
        public QDialog
{
    Q_OBJECT

public:
    adhesionDlg(
            QWidget* parent = 0,
            const char* name = 0,
        bool modal = FALSE,
        WFlags fl = 0 );
    ~adhesionDlg();

    QLabel* labelNom;
    QLineEdit* nom;
    QButtonGroup* groupSexe;
    QRadioButton* masculin;
    QRadioButton* feminin;
    QPushButton* boutonValider;
    QPushButton* boutonAnnuler;

public slots:
    virtual void slotValider();

protected:
    QGridLayout* adhesionDlgLayout;
    QHBoxLayout* layout10;
    QHBoxLayout* layout16;
    QVBoxLayout* layout17;

protected slots:
    virtual void languageChange();

};
```

```
#endif // ADHESIONDLG_H
```

et le fichier d'implémentation `adhesionDlg.cpp` dont nous ne voyons ici que le `slot` `slotValider` :

```
void adhesionDlg::
slotValider()
{
    qWarning( "adhesionDlg::slotValider():"
            "Not implemented yet" );
}
```

Vous constatez que le `slot` contient du code dans la classe produite par `uic`. Il sert uniquement à avertir le développeur que la classe n'a pas été implémentée réellement et que c'est à lui de le faire. C'est pour cette raison que dans la définition, le `slot` a été déclaré `virtual` ce qui va nous permettre de le remplacer. Pour sous-classer `adhesionDlg` nous devons créer deux fichiers indépendants, un pour contenir la définition de notre sous-classe, l'autre l'implémentation. Pour tester cette méthode, vous pouvez créer un nouveau répertoire et y copier uniquement les fichiers `adhesionDlg.ui` et `main.cpp`. Ajoutez un fichier projet (.pro). Dans votre répertoire, créez à présent deux nouveaux fichiers[1] :

`adhesion.h`:

```
#include "adhesionDlg.h"

class Adhesion : public adhesionDlg
{
        Q_OBJECT
public:
        Adhesion(QWidget * parent = 0, const char
* name = 0);
private slots:
```

[1] `uic` permet de générer en ligne de commande l'en-tête et l'implémentation des classes dérivées (Voir section uic dans le chapitre Outils).

```
        void slotValider();
};
```

Il s'agit de la définition de la classe héritée. La classe `adhesionDlg` est héritée dans notre nouvelle classe `Adhesion`. Cela signifie qu'elle hérite de ses propriétés et peut en rajouter des nouvelles.

et `adhesion.cpp`:

```
#include <qmessagebox.h>
#include <qstring.h>
#include <qradiobutton.h>
#include <qlineedit.h>

#include "adhesion.h"

Adhesion::Adhesion(QWidget * parent,
        const char * name)
        : adhesionDlg(parent, name)
{
}

void Adhesion::slotValider()
{
QString Sexe;
if( masculin->isChecked() )
        Sexe = "Monsieur ";
else
        Sexe = "Madame ";
QMessageBox::information( this, "Fichier
adhesion",
Sexe+nom->text()+" inscrit(e)" );

}
```

Notre `slot slotValider` remplace celui de la classe de base et c'est maintenant lui qui sera exécuté.

Dans le constructeur de la classe, nous faisons appel au constructeur de la classe de base `adhesionDlg`.

Pensez à modifier votre fichier projet qui devrait contenir :

```
TEMPLATE = app
INCLUDEPATH += .

HEADERS += adhesion.h
INTERFACES += adhesionDlg.ui
SOURCES += adhesion.cpp main.cpp
```

et le fichier `main.cpp` :

```
#include <qapplication.h>
#include "adhesion.h"

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

  Adhesion *adhesion = new Adhesion(0);

  a.setMainWidget( adhesion );
  adhesion->show();
  return a.exec();
}
```

Nous constatons que c'est maintenant la classe `Adhesion` qui est utilisée comme dialogue principal dans la fonction main et non plus `adhesionDlg`.

Compilez puis lancez le programme, tout fonctionne comme dans la méthode précédente. Le deux façons de procéder ont leurs avantages et leurs inconvénients. Créer une sous-classe double le nombre de fichiers et de classes, mais offre plus de possibilités. En effet, nous pouvons saisir des opérations à exécuter dans le constructeur de notre classe dérivée. D'autre part, ce constructeur peut recevoir des arguments avec

des valeurs par défaut ce qui n'est pas possible avec les slots saisis dans `Qt Designer`.

Dans les classes saisies dans `Qt Designer`, les constructeur/destructeur n'étant pas accessibles, ils ont été remplacés par deux fonctions `init()`/`destroy()` à appeler pour les remplacer.

Entrer du code dans une classe générée par `uic` ?

Une des erreurs les plus fréquentes est de saisir du code dans les fichiers de la classe produits par `uic` à partir du fichier (.ui). En effet, ces fichiers sont écrasés chaque fois que le fichier (.ui) est modifié. En revanche, saisir du code peut s'effectuer soit dans une classe dérivée, soit dans `Qt Designer` qui sauvegardera ce code dans un fichier (.ui.h).

Tout n'a pas été étudié au sujet des dialogues dans `Qt Designer`. Néanmoins, le lecteur en connaît assez pour commencer leurs utilisations.

8.3. Fenêtres principales

Les fenêtres principales, ont été étudiées dans le chapitre du même nom. La fabrication des menus et des barres d'outils y était basée sur l'utilisation des `QAction`. La construction des fenêtres principales dans `Qt Designer` n'échappe pas à cette règle.

Nous allons étudier dans cette partie la construction d'une application basée sur l'utilisation d'une fenêtre principale, mais comme beaucoup des principes d'utilisation de `Qt Designer` ont été vus dans les sections précédentes, seules les nouveautés seront expliquées dans cette partie. Commencez par créer un nouveau répertoire pour stocker les fichiers. Puis, comme précédemment, choisissez la construction d'un nouveau projet. Je choisis de l'appeler `fenetrePrincipale.pro` mais vous êtes bien sûr libre de choisir un autre nom.

Effectuez maintenant la création d'une fenêtre principale. Dans le dialogue "Main Window Wizard" désactivez toutes les cases à cocher. En effet, nous allons construire ensemble tous les éléments de notre inter-

face. Inutile donc de demander la création des menus et barres d'outils par défaut. En temps normal, sans doute aurions-nous laissé les options cochées. `Qt Designer` affiche une nouvelle fenêtre "Action Editor"

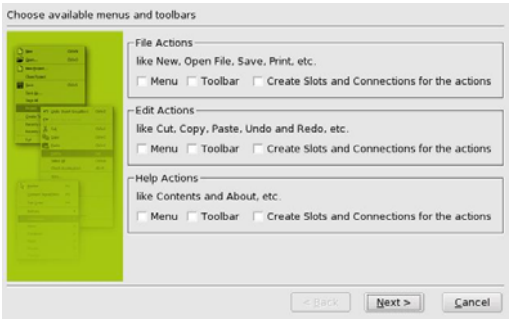


Figure 8.10. Aucune option cochée dans Main Window Wizard.

dans son environnement. C'est dans cette fenêtre que nous allons ajouter les `QAction`.

Nous avons vu dans la section "Création des actions" que les `QAction` permettent de contenir un texte à afficher, un `slot` à déclencher, une image à afficher, un raccourci clavier et bien plus encore. Une fois un `QAction` créé, celui-ci peut être inséré indifféremment dans une barre de menu ou une barre d'icônes.

Dans la fenêtre "Action Editor" cliquez sur l'icône "nouveau". Une nouvelle action est insérée dans la fenêtre. Nous avons plusieurs propriétés à changer la concernant :

- `name` : Comme tout objet Qt, une action à un nom. Comme d'habitude, donnez un nom significatif à l'action. Pour cette première action, je l'appelle "actionNouveau"

- `toogleAction` : Détermine si le bouton présent dans la barre d'outils est un interrupteur. C'est-à-dire s'il va rester enfoncé lors d'un clique. Lorsque sa valeur est à false, le bouton ne reste pas enfoncé.

- `text` : est affiché dans les fenêtres d'informations qui s'affichent lorsque le curseur de la souris reste de manière prolongée sur le widget.

- `menuText` : C'est le texte contenu dans les menus. Entrez par exemple "Nouveau fichier".

- `iconset` : Permet d'indiquer l'icône à afficher à la fois dans la barre de menu et dans la barre d'outils. En générant une fenêtre principale, `Qt Designer` qui fait bien les choses, a créé une collection d'images. Lorsque l'on clique sur le bouton "…" de la propriété `iconSet`, une fenêtre permet de choisir dans la collection d'images celle à afficher par l'action. La collection d'images contient déjà les images des actions les plus courantes (nouveau, ouvrir etc.). Il est quand même possible de rajouter des images à la collection si celles proposées ne suffisent pas.

Figure 8.11. La collection d'images.

Il y a d'autres propriétés qui ne sont pas détaillées, mais le plus important a été vu.

Le bouton "Connect Current Action" donne la possibilité d'indiquer quel `slot` déclencher lorsque l'action sera sélectionnée, soit dans un menu, soit dans une barre d'outils. Vous pouvez comme d'habitude choisir un `slot` appartenant à une classe Qt comme `close()` chargé de fermer la fenêtre. Vous pouvez bien entendu utiliser un `slot` personnel dont le code sera à entrer.

Pour insérer une barre de menu dans la fenêtre principale choisissez l'entrée "Add Menu Item" qui s'affiche en effectuant un clic droit sur la fenêtre. Un titre nommé "Menu" apparaît dans la fenêtre. Renommez-le immédiatement en "&Fichier". Pour ce faire, double-cliquez sur le titre, le curseur de saisie apparaît. Le & représente le raccourci clavier affecté au titre de menu. Ainsi, le menu "Fichier" pourra être déroulé en tapant au clavier Alt+F.

Pour insérer l'action "Nouveau" dans le menu "Fichier" : Faites glisser l'action avec la souris et amenez-la sur le menu "Fichier" qui va se dérouler et permettre d'insérer l'action. L'endroit d'insertion est représenté par une ligne rouge.

Fabriquez à présent une action `quitter` avec "&Quitter" comme `menuText` et connectez y le `slot close`. Notre menu est maintenant composé de deux entrées.

Insérer une barre d'outils n'est pas plus difficile. Commencez par effectuer un clique droit sur la forme puis choisissez "Add Tool Bar". Dans cette barre d'outils, effectuez un glissé-déposé de l'action "Nouveau". Comme par magie, l'icône s'affiche dans la barre d'outils.

Nous avons vu dans ce chapitre comment implémenter les fonctionnalités importantes d'une fenêtre principale, c'est-à-dire un menu et une barre d'outils. L'objet prépondérant est `QAction` qui permet de décrire dans le même objet l'action à déclencher, et ce, de la même manière pour un menu ou une barre d'outils. Une fenêtre principale peut contenir un widget central dans lequel les traitements seront visualisés ou être MDI, c'est à dire avoir la possibilité d'ouvrir plusieurs fenêtres conte-

nant des documents.

Reportez-vous à la section précédente pour connaître la manière d'implémenter le code des slots dans ou hors `Qt Designer`.

Cette partie consacrée aux fenêtres principales dans `Qt Designer` n'est pas exhaustive. Néanmoins, elle est suffisante pour permettre de commencer à travailler dans celui-ci.

8.4. Collection d'images

Dans la section consacrée aux actions, nous avons présenté le concept de collection d'image de projet. Si vous utilisez un projet, vous pouvez ajouter des images à la collection d'images du projet, et ces images peuvent être partagées et employées par n'importe laquelle des formes que vous incluez dans le projet. Les images sont stockées dans un sous-répertoire images, à l'intérieur du répertoire du projet. Toutes les fois que vous modifiez la collection d'image, `uic` crée un fichier source qui contient les données d'image dans leur format binaire et une fonction qui instancie ces images. Elles sont accessibles par toutes les formes dans le projet et les données sont partagées.

8.4.1. Rajouter une collection au projet

Ajouter une collection au fichier projet (.pro) consiste à créer la rubrique `IMAGES:` dans le fichier projet suivi des noms des fichiers images à intégrer dans la collection. Cette fonctionnalité a été vue dans la section traitant des actions. Le lecteur s'y reportera pour plus de détail.

8.4.2. Intégrer une collection dans Qt Designer

En générant avec `Qt Designer` une fenêtre principale, nous avons remarqué qu'est fabriquée également une collection d'images correspondant aux actions les plus courantes. Tout ce que nous allons voir dans cette partie ne concernera donc pas les fenêtres principales.

Nous allons effectuer dans `Qt Designer` la construction d'une collection d'images.

Pour le test, créez un répertoire et un projet tous deux nommés `collection`. Choisissez ensuite dans le menu "Project" l'entrée "Image Collection". Une fenêtre vide s'affiche, elle va nous permettre d'ajouter des images à notre collection. En cliquant sur le bouton "Add…", une fenêtre de sélection de fichiers apparaît. Avec elle, partez à la recherche des images à ajouter. Elles peuvent être stockées dans plusieurs formats (environ 8 différents). Les heureux utilisateurs de Linux pourront chercher les images dans `/usr/share/icons/` où une multitude de sous-répertoires est à leur disposition.

Lorsqu'une collection contient des images, celles-ci peuvent être utilisées dans l'ensemble des formes générées dans le projet. Par exemple, générez un nouveau dialogue puis insérez dans celui-ci un label. Les labels ont la particularité de pouvoir afficher des images en renseignant leur propriété `pixmap`. En cliquant sur cette dernière, vous pouvez sélectionner une des images de la collection. Comme nous l'avons étudié à plusieurs reprises, les objets `QAction` utilisent eux aussi les images stockées dans les collections.

Résumé

Nous avons appris dans ce chapitre l'utilisation de `Qt Designer`, le créateur d'interface de Qt. Ont été étudiés la création des projets, des dialogues ainsi que des fenêtres principales. Enfin, les collections d'images ont été expliquées.

9. Techniques de débogage

Qt dispose de trois fonctions globales pour afficher des textes de débogage ou d'avertissement. Sous Linux, ces messages sont affichés sur le périphérique d'erreur standard (stderr) qui est par défaut la console. Sous Windows ces messages sont transmis au débogueur :

- **qDebug(const char * msg, …)**: Affiche un message de débogage msg. Le fonctionnement est similaire à la fonction C `printf()` et peut recevoir des arguments. Exemple:

```
qDebug("Contenu de i: %d nom: %s", i,
nom.latin1());
```

- **qWarning(const char * msg, …)**: Affiche un message d'avertissement. Cette fonction a un fonctionnement identique à `qDebug`.

- **qFatal(const char * msg, …)**: Affiche un message fatal puis quitte le programme. Cette fonction a un fonctionnement identique à `qDebug`.

Il est également possible d'utiliser des outils standards de débogage comme `gdb` sous Linux/Unix ou sa version graphique `ddd`. Pour permettre le débogage avec ces outils, il est nécessaire de préciser le mode debug dans le fichier projet (.pro) :

```
CONFIG += debug
```

La compilation des fichiers du projet produira des fichiers objets (.o ou .obj) incluant les symboles de débogage. Le programme pourra alors être ouvert et exécuté pas à pas dans le débogueur. Voici un petit exemple de programme construit avec les symboles de débogage puis

tracé avec `gdb` :

Dans un répertoire, créez un fichier nommé par exemple `debogage.cpp` puis mettez à l'intérieur le code suivant :

```
1       class QuiPlante
2       {
3       public:
4               QuiPlante();
5               void calcul();
6       };
7
8       QuiPlante::QuiPlante()
9       {
10      }
11
12      void QuiPlante::calcul()
13      {
14              char tableau[4];
15              for(int x=0; x<5; x++)
16              {
17                      tableau[x] = x;
18              }
19      }
20
21      int main()
22      {
23              QuiPlante quiplante;
24              quiplante.calcul();
25              return 0;
26      }
```

Dans le répertoire, lancez le maintenant bien connu `qmake -project`. Éditez le fichier projet obtenu afin de rajouter la ligne `CONFIG += debug`. Lorsque c'est fait, générez le fichier Makefile en exécutant `qmake`. En lançant la compilation par `make`, nous remarquons que le

paramètre `-g` est transmis au compilateur lui demandant d'inclure les symboles de débogage. Lorsque nous exécutons le programme compilé, on obtient un magnifique "Segmentation fault". Notre programme vient de "planter".

Voyons maintenant comment, grâce à un débogueur, trouver l'origine du plantage. Dans une console lancez le débogueur avec comme paramètre le nom du programme:

```
gdb debogage
```

GDB charge tout ce dont il a besoin… et présente une invite (gdb).

```
r
```

`r` veut dire `run`, d'ailleurs `run` marche aussi. GDB répond :

```
Starting program: /home/brd/debogage/debogage
```

Comme lors de la précédente exécution, le programme plante. Mais cette fois gdb indique l'endroit en cause :

```
0x080484a7 in QuiPlante::calcul()
(this=0xbffff5f7) at debogage.cpp:17
17                      *tableau[x] = x;
```

Il s'agit de la ligne 17. Là il faut quand même réfléchir un peu, car bien que gdb indique la ligne concernée, il ne donne pas la raison du plantage.

Le message de `gdb` nous indique que le problème se situe dans la fonction `QuiPlante::calcul()`. Nous allons insérer un point d'arrêt dans la fonction puis tracer le programme ligne après ligne jusqu'à débusquer la ligne en cause:

```
break QuiPlante::calcul()
run
```

`gdb` a relancé le programme, s'est arrêté au point demandé et nous affiche:

```
Breakpoint 1, QuiPlante::calcul()
(this=0xbffff5f7) at debogage.cpp:15
15              for(int x=0; x<5; x++)
```

Nous pouvons demander l'exécution de la ligne suivante par :

```
Next
```

ou `n`.

Le plantage semble survenir alors que nous sommes dans la boucle `for`. Chaque fois que le début de boucle recommence, on peut afficher le contenu de la variable x:

```
print x
```

Le plante survient alors que x contient 4. Nous avons trouvé l'origine du plantage. En effet, x a la valeur 4 alors que le tableau qui contient 4 éléments avec des indices allant de 0 à 3. Je modifie la boucle et je remplace le

```
for(int x=0; x<5; x++)
```

par

```
for(int x=0; x<4; x++)
```

Tout fonctionne maintenant bien et le plantage a disparu.

Si utiliser `gdb` en ligne de commande vous rebute, vous pouvez à la place choisir `ddd`. En fait, ce n'est qu'une interface graphique car `ddd` exécute lui même `gdb`. Mais bon c'est quand même plus convivial.

Résumé

Ce chapitre a permis de passer en revue les quelques fonctions de débogage fournies par Qt. Les utilisateurs de Linux/Unix ont également pu apprendre le fonctionnement du débogueur gdb.

Index