

# C++, développement d'applications graphiques en QT

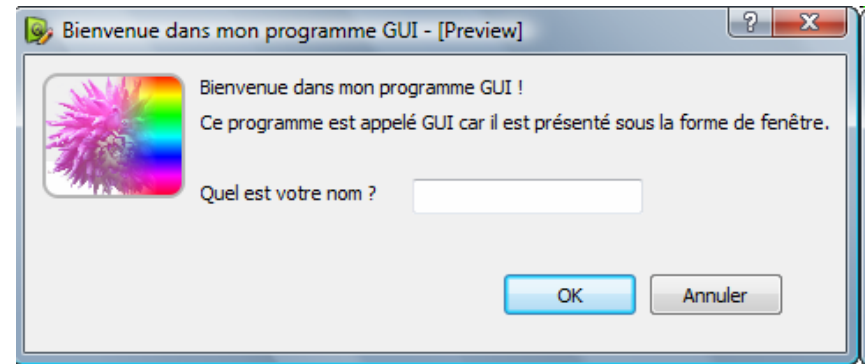
- 1) Présentation du langage QT
- 2) Les différents types de projets
- 3) Les éléments importants d'un projet
- 4) Le modèle MVC avec QT
- 5) Les différents composants graphiques
- 6) La gestion des événements
- 7) Utilisation de XML avec QT
- 8) Le système de plugin de QT
- 9) L'internationalisation
- 10) En plus avec QT

# 1) Présentation du langage QT

- Historique du langage.
- Les différentes possibilités d'utilisation.
- Les autres librairies graphiques existantes.

# 1) Présentation du langage QT

- Qt est la possibilité de construire des GUI (Graphical User Interface) dans différents langages de programmation:
  - C++
  - Python
  - Java
  - ...



# 1.1) Différents créateurs de GUI

- Manque de portabilité:
  - chaque système d'exploitation (Windows, Mac OS X, Linux...) propose un moyen de créer des fenêtres...
  - 2 types de choix :
    - application spécialement écrite pour l'OS que l'on veut,
    - une bibliothèque qui s'adapte à tous les OS telles que **Qt**.

# 1.2) Les bibliothèques propres aux OS

- Sous Windows : on dispose du framework **.NET**.
  - ensemble très complet de bibliothèques utilisables en C++, C#, Visual Basic...
- Sous Mac OS X : la bibliothèque de prédilection s'appelle **Cocoa**,
  - en général en langage "Objective C" , c'est une bibliothèque orientée objet.
- Sous Linux : tous les environnements de bureaux reposent sur X, la base des interfaces graphiques de Linux.
  - X propose une bibliothèque appelée Xlib il y a une bibliothèque plus simple d'utilisation et multi-plateforme comme **GTK+** (sous Gnome) ou **Qt** (sous KDE).

# 1.3) Les bibliothèques multi-plateformes

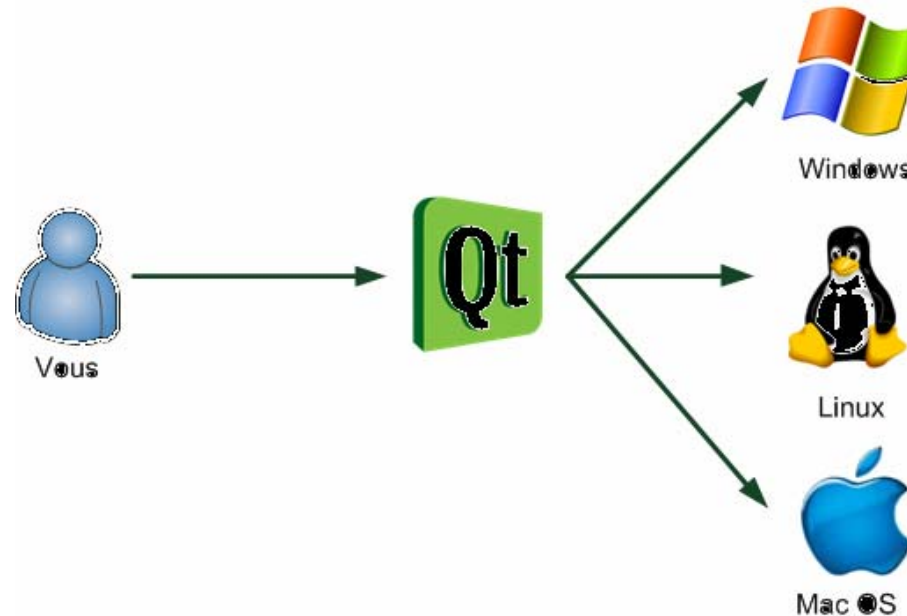
- Elles simplifient la création d'une fenêtre.
  - Il faut moins de lignes de code pour ouvrir une "simple" fenêtre.
  - elles forment un ensemble cohérent avec des conventions de nom.
- Principales bibliothèques multi-plateforme
  - **.NET**: développé par Microsoft pour succéder à l'API Win32.
  - **GTK+** : une des bibliothèques les plus utilisées sous Linux avec Gnome,
  - **Qt** est très utilisée sous Linux sous l'environnement de bureau KDE.
  - **wxWidgets** : une bibliothèque objet complète, comparable en gros à Qt.
  - **FLTK** se veut légère, dédiée uniquement à du prototypage d'interfaces graphiques simples multi-plateforme.

# 1.4) Le framework Qt

- **Qt** est donc constituée d'un ensemble de bibliothèques, appelées "modules":
  - Module **GUI** : la partie création de fenêtres.
  - Module **OpenGL** : Qt peut ouvrir une fenêtre contenant du 3D gérée par OpenGL.
  - Module de **dessin** : pour créer des dessins dans une fenêtre (en 2D), le module de dessin est très complet !
  - Module **réseau** : pour l'accès au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, ...
  - Module **SVG** : possibilité de créer des images et animations vectorielles, à la manière de Flash.
  - Module de **script** : Qt supporte le Javascript que l'on réutilise=r dans les applications pour ajouter des fonctionnalités, sous forme de plugins par exemple.
  - Module **XML** : c'est un moyen pratique d'échanger des données avec des fichiers formés à l'aide de balises, un peu comme le XHTML.
  - Module **SQL** : permet un accès aux bases de données (MySQL, Oracle, ...).

# 1.5) Qt est multiplateforme

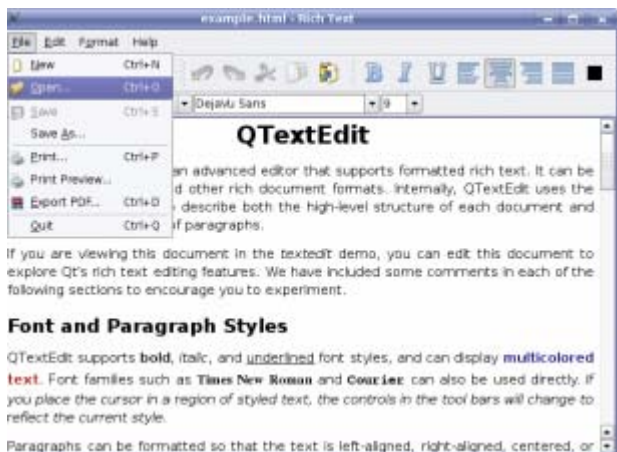
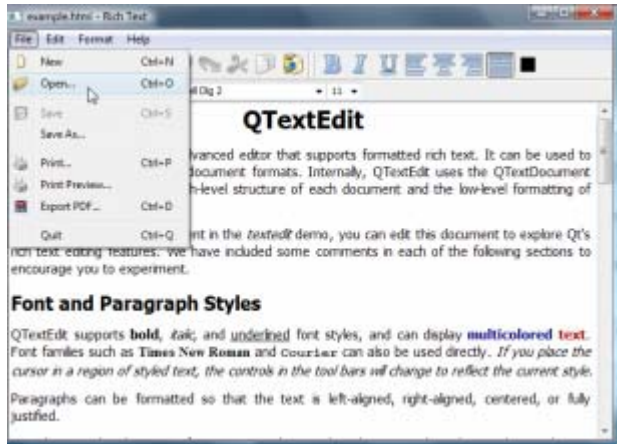
- Grâce à cette technique, les fenêtres ont un "look" adapté à chaque OS.
- Le développeur code pour Qt, et Qt traduit les instructions pour l'OS.





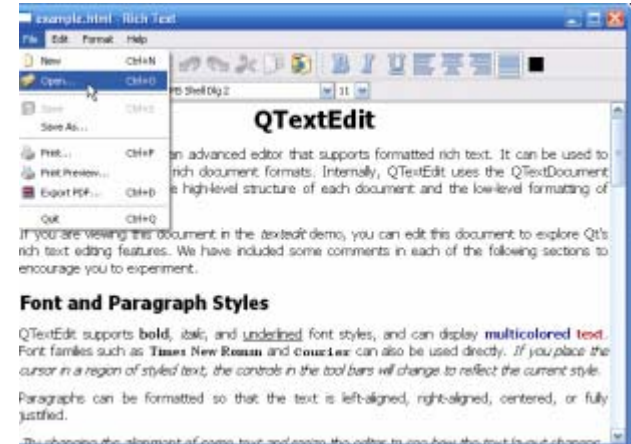
# 1.6) Qt examples

Sous Windows 7



Sous Linux

Sous Windows XP



Sous Mac OS X

# 1.7) Qt genèse

- Qt est un framework développé initialement par la société Trolltech, qui fut racheté par Nokia par la suite.
  - Le développement de Qt a commencé en 1991 et il a été dès le début utilisé par KDE, un des principaux environnements de bureau de Linux.
- Qt s'écrit "Qt" et non "QT", donc avec un "t" minuscule = Cute
- Qt est distribué sous 2 licences:
  - LGPL
  - propriétaire

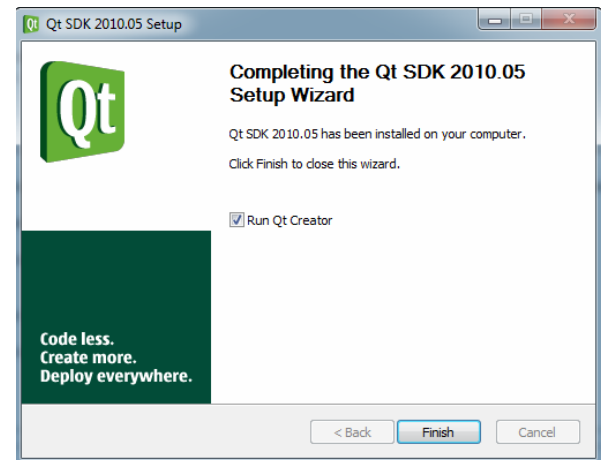
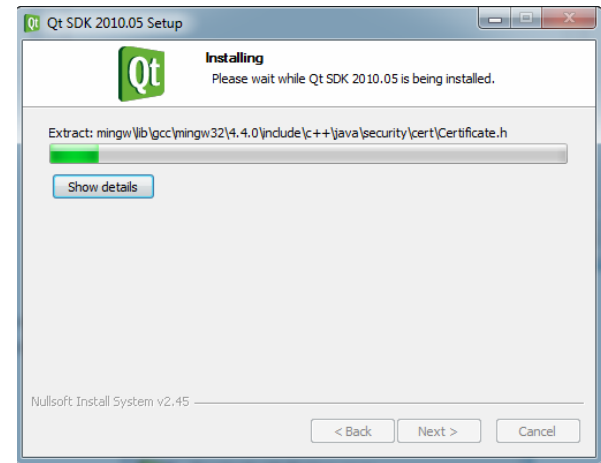


# 1.8) Installation de Qt

- télécharger Qt sur le site  
`http://qt.nokia.com/downloads`
- choisir entre la version LGPL ou la version commerciale.
  - nous allons utiliser la version sous licence LGPL.
- ensuite choisir entre :
  - **Qt SDK** : la bibliothèque Qt + un ensemble d'outils pour développer avec Qt, incluant un IDE spécial appelé Qt Creator.
  - **Qt Framework** : contient uniquement la bibliothèque Qt.

# 1.9) Installation sous Windows

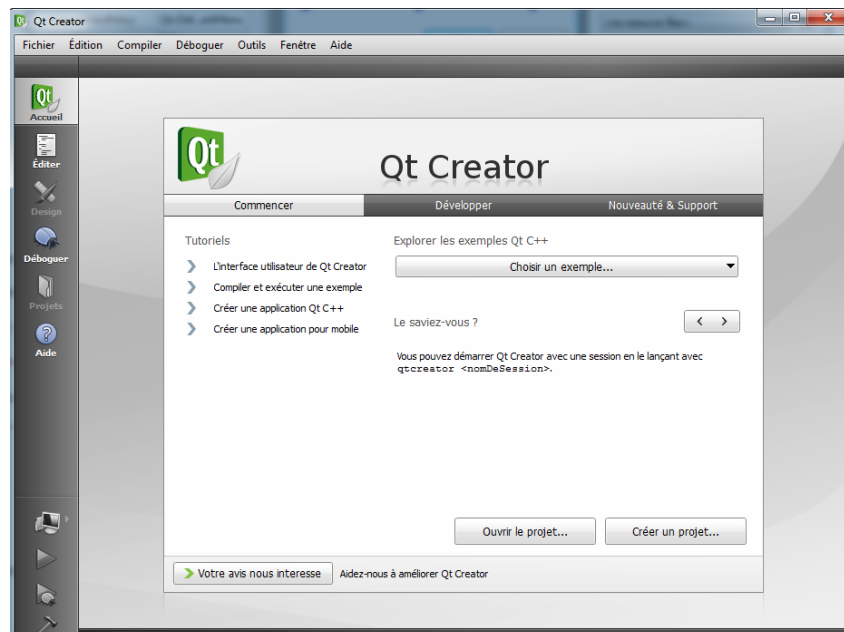
un assistant d'installation classique



Qt s'installe ensuite (il y a beaucoup de fichiers ça peut prendre un peu de temps).  
**Qt Creator** est un IDE spécialement optimisé pour travailler avec Qt.

# 1.10) Qt Creator

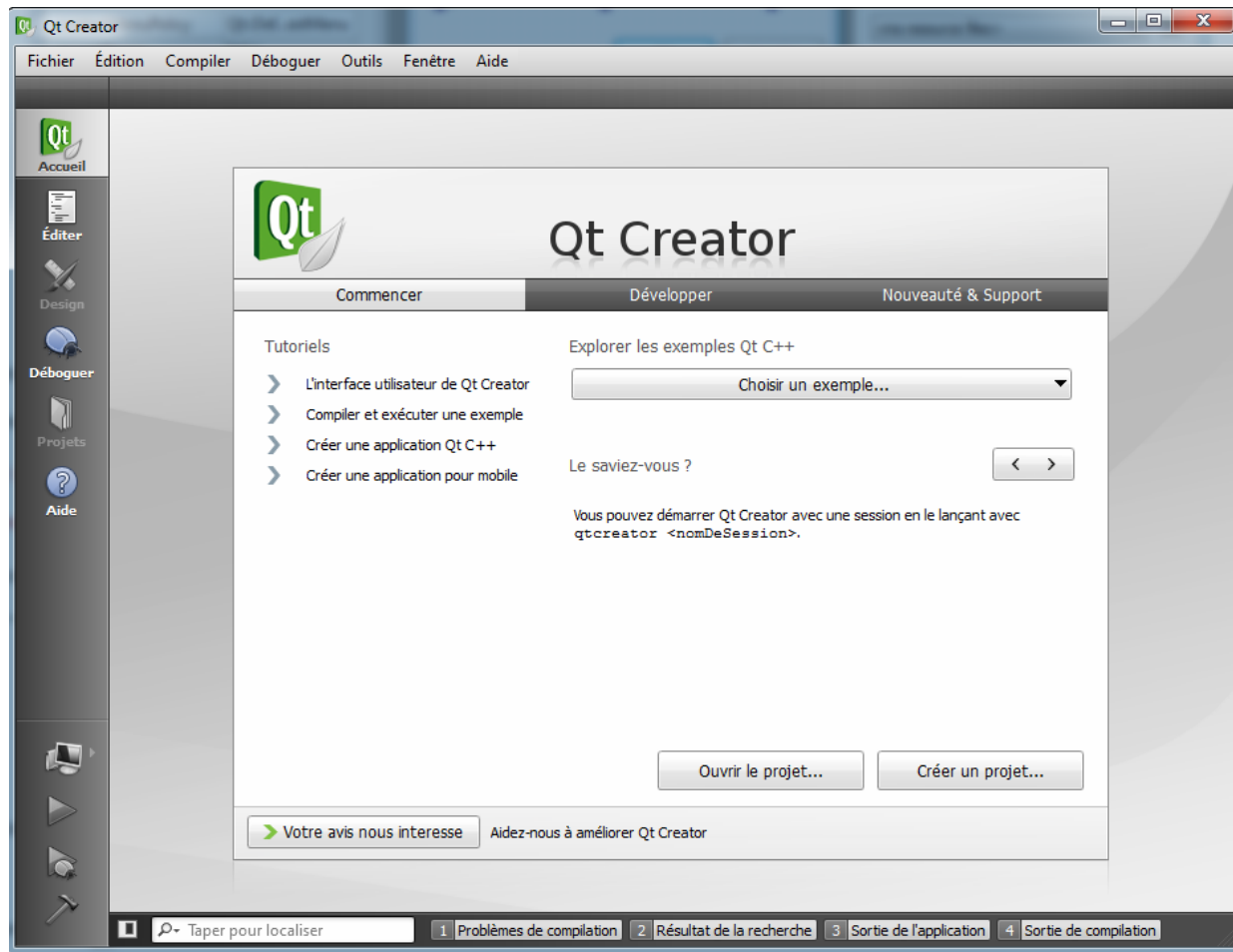
- Il est recommandé fortement d'utiliser l'IDE Qt Creator. Il est particulièrement optimisé pour développer avec Qt. En effet, c'est un programme tout-en-un qui comprend entre autres :
  - Un IDE pour développer en C++, optimisé pour compiler des projets utilisant Qt (pas de configuration fastidieuse)
  - Un éditeur de fenêtres, qui permet de dessiner facilement le contenu de ses interfaces à la souris
  - Une documentation indispensable pour tout savoir sur Qt



## **2) Les différents types de projets**

- Présentation des différents types de projet avec QT.
- Présentation de QT Creator.
- La structure de base d'une application à base d'IHM.

## 2) Les différents types de projets



D'autres possibilités de développement existent

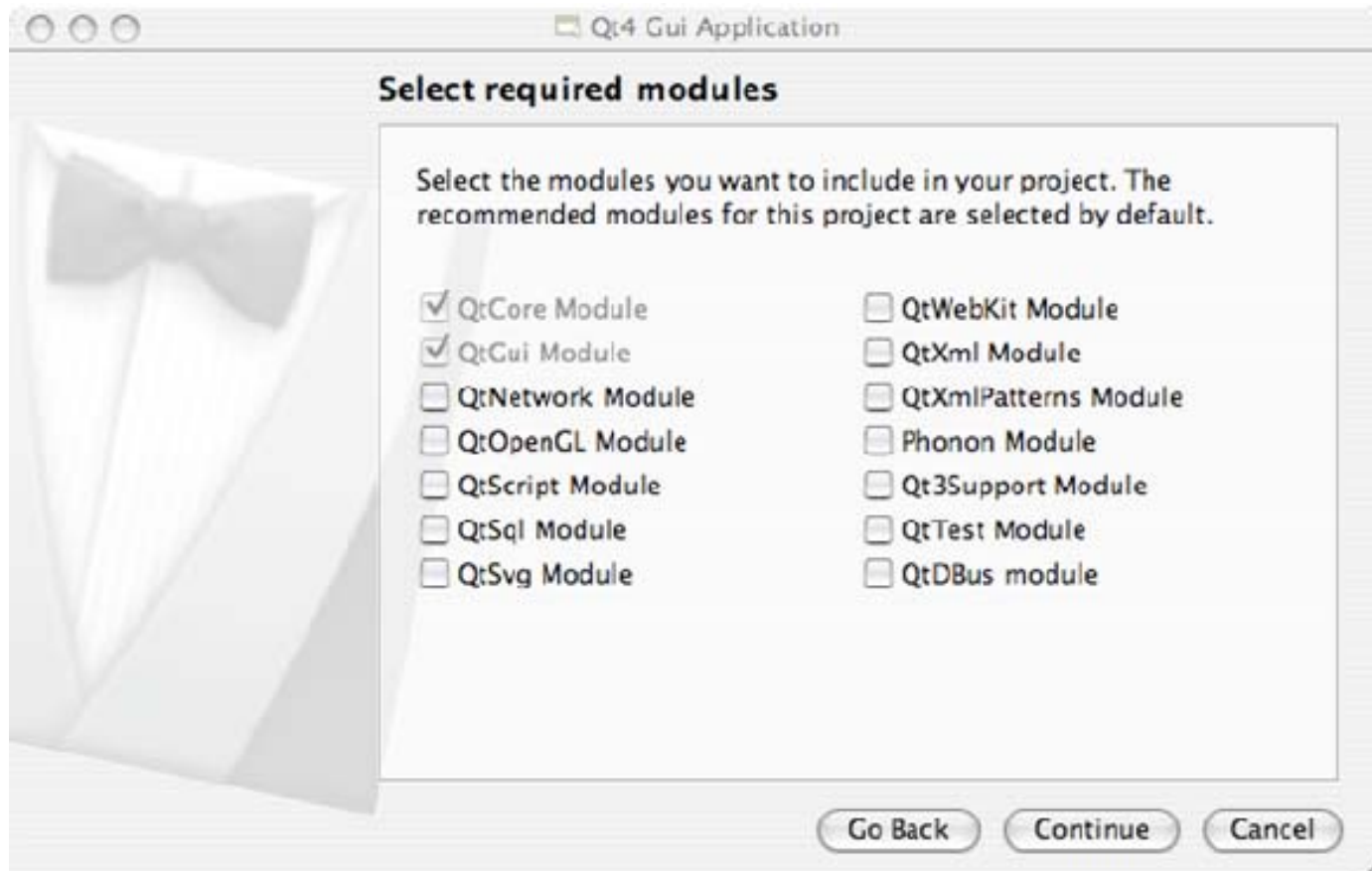
1. Avec Eclipse et MingW
2. Avec Visual C++ ou Visual Studio

# 2.1) Les fonctionnalités de Qt Creator

- Qt Creator est un EDI C++ pour Qt par Qt. interface agréable et intuitive ;
  - autocomplétion très puissante (pour les classes Qt comme pour la STL) ;
  - coloration syntaxique très agréable ;
  - donne des indications basiques sur des erreurs de syntaxe;
  - affichage dans le débogueur adapté pour les classes Qt ;
  - intégration de l'aide Qt ;
  - intégration du designer ;
  - affichage avancé des portées des () et {} par colorisation ;
  - utilisation du `.pro` garantissant le bon déroulement des compilations et rendant la compilation totalement indépendante de l'EDI ;
  - le support de différentes cibles pour chaque projet ;
  - un back-end Python pour GDB ;
  - ...
- Il est pensé pour utiliser des plug-ins et donc s'adapter aux différents compilateurs.



## 2.2) Les types de projets



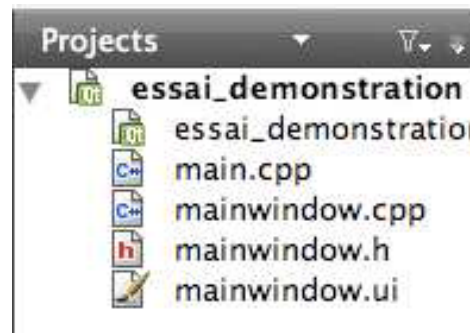
Par défaut :

- `mainwindow.cpp`
- `mainwindow.h`

Le fichier `mainwindow.ui` correspond au fichier de l'interface graphique

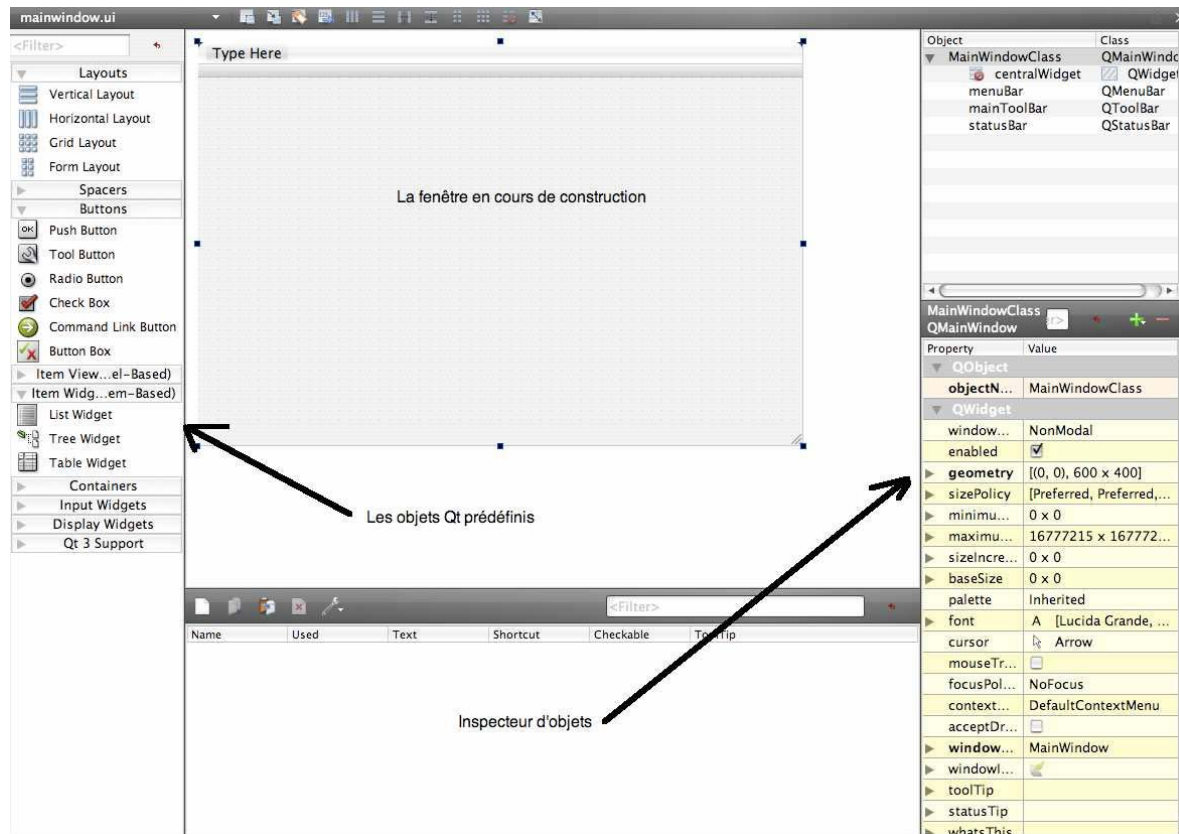
## 2.3) La structure d'un projet

- Le projet se compose de 5 fichiers :
  - les 2 fichiers C++ dont les noms correspondent au projet
  - un fichier `main.cpp` qui correspond au programme principal.
  - Le fichier `mainwindow.ui` qui correspond à l'interface graphique.
  - Le fichier projet Qt nommé ici `essai_demonstration`.



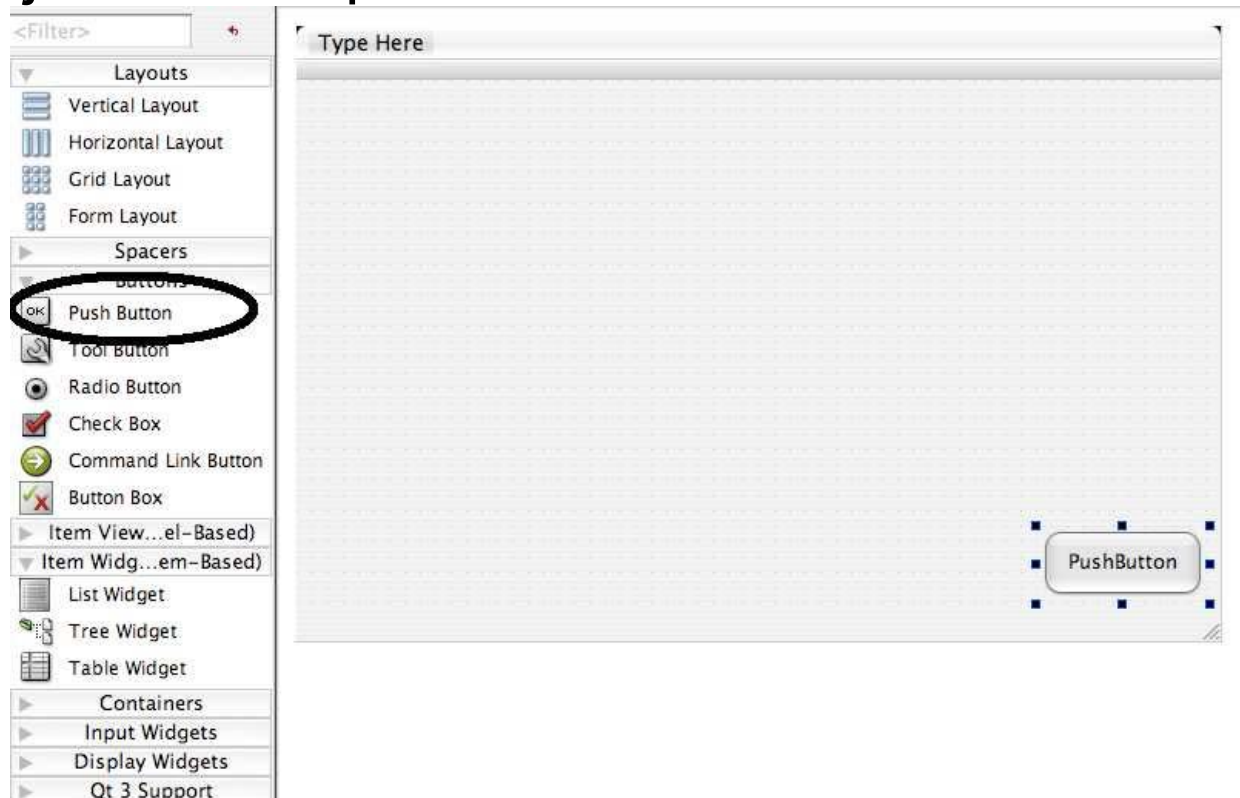
## 2.3) Le générateur d'interface

- Un double clic sur `mainwindow.gui` lance automatique le générateur d'interface.

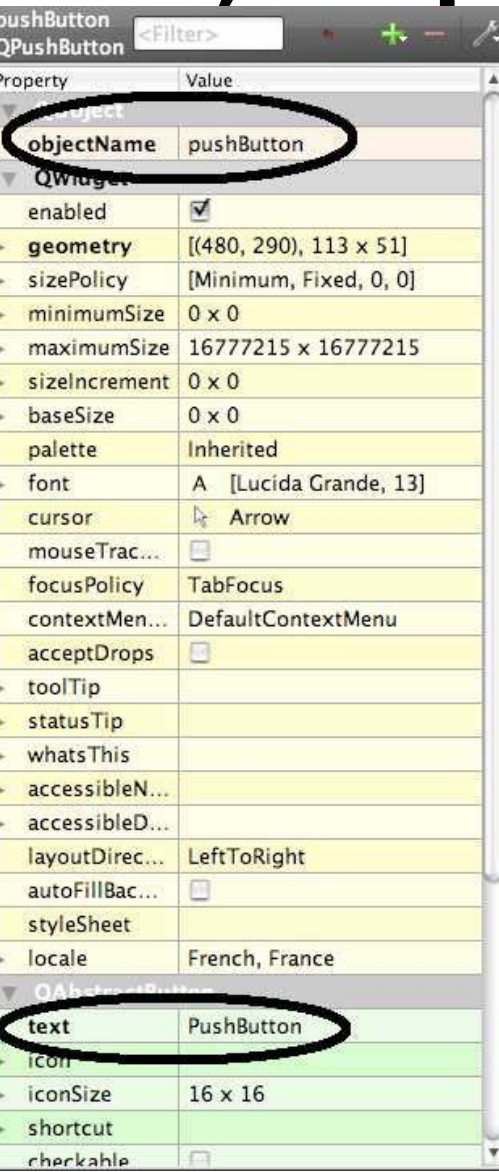


## 2.4) étape 1: création d'un widget

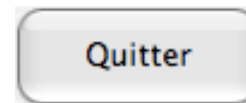
- Création d'un bouton dans la fenêtre.
  - L'inspecteur d'objet affiche alors les caractéristiques de l'objet bouton qui vient d'être créé.



## 2.4) étape 1: création d'un widget



- 2 propriétés sont importantes :
  - La propriété `objectName` correspond au nom C++ de l'instance bouton qui vient d'être créée. Cette propriété est dans la section `QObject`.
  - La propriété `Text` dans la section `QAbstractButton` qui correspond au texte affiché sur le bouton.



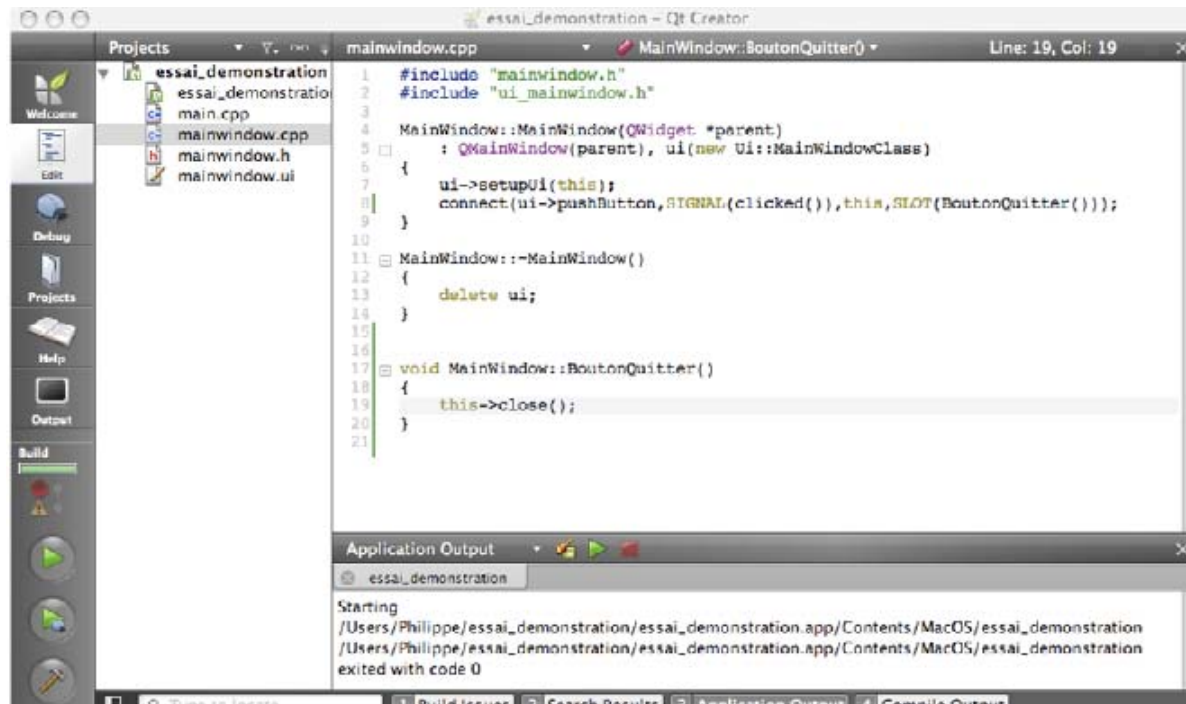
## 2.4) étape 2: gestion d'événement

- Dans le fichier `mainwindow.h`
  - créer une nouvelle section nommée « Private Slots »
  - définir une procédure nommée par exemple `BoutonQuitter`.



## 2.4) étape 2: gestion d'événement

- Il faut inclure dans ce fichier :
  - Le corps de la méthode `BoutonQuitter`
  - Une connexion entre le signal ou événement `clicked` et la méthode `BoutonQuitter` ;



## 2.4) étape 2: gestion d'événement

- La méthode « `BoutonQuitter` » contient un simple appel à la méthode « `close` » de la fenêtre.

```
void MainWindow::BoutonQuitter()  
{  
    this->close();  
}
```

- Le lien entre l'événement « `Click` » et la méthode se fait par la méthode `Connect` :

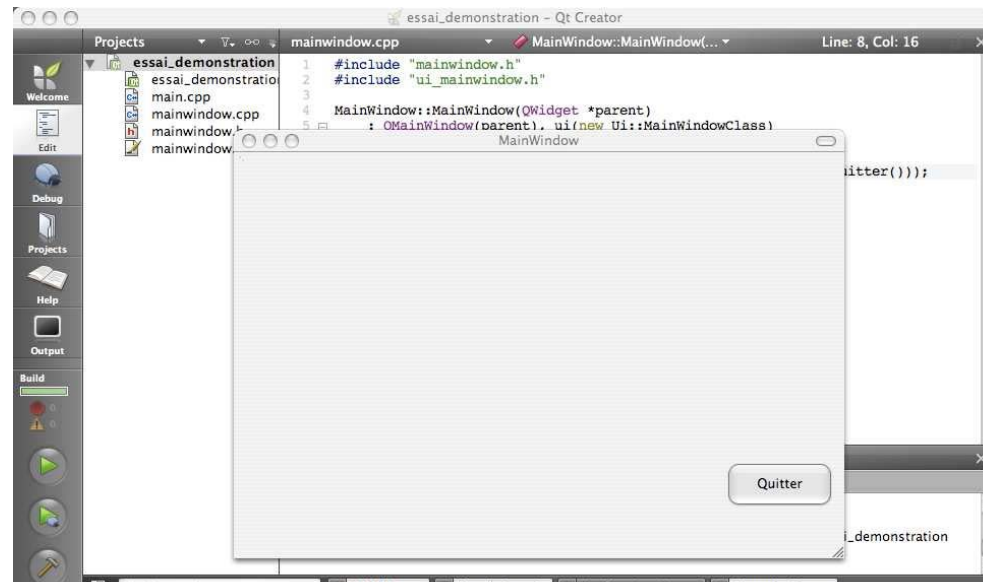
```
connect(  
    ui->pushButton, SIGNAL(clicked()), this, SLOT(BoutonQuitter()));
```

- Celle-ci se compose de 3 parties principales :
  - Le 1<sup>er</sup> paramètre (`ui->pushButton`) fait référence à l'objet graphique ;
  - Le 2<sup>ème</sup> `SIGNAL(clicked())` fait référence à l'événement concerné (ici « clic ») ;
  - Le 4<sup>ème</sup> crée la connexion entre l'objet graphique et la procédure en Qt le lien s'appelle un slot.



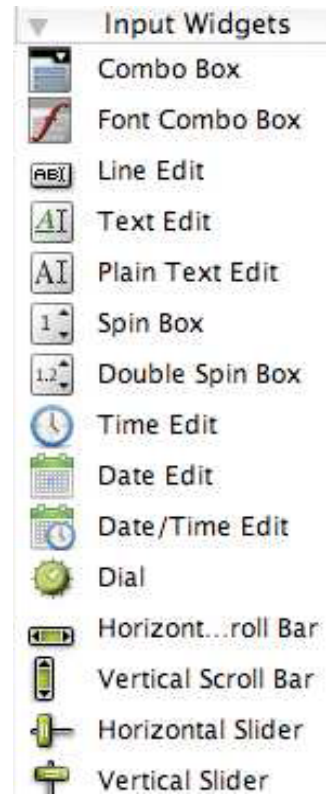
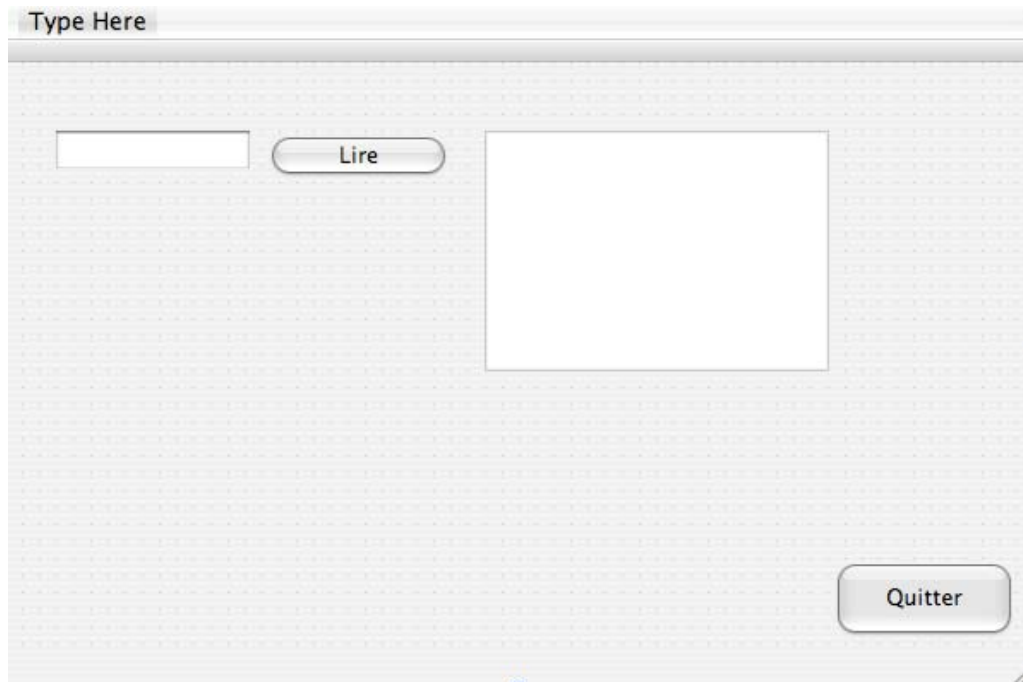
## 2.5) étape 3: validation

- Après compilation et en lançant directement le code à partir de QtCreator on peut valider par un test.
- En examinant le dossier de travail, on retrouve les différents fichiers du projet ainsi qu'un fichier nommé
  - « `essai_demonstration.app` » sous Macintosh
  - « `essai_demonstration.exe` » sous Windows.



## 2.6) Afficher des messages sur la fenêtre

- Créez une fenêtre avec
  - un « `LineEdit` »
  - un « `TextEdit` »
  - séparé par un bouton intitulé « Lire »



## 2.6) Afficher des messages sur la fenêtre

- Déclarer une nouvelle procédure dans le fichier `mainwindow.h` :

```
private slots:
```

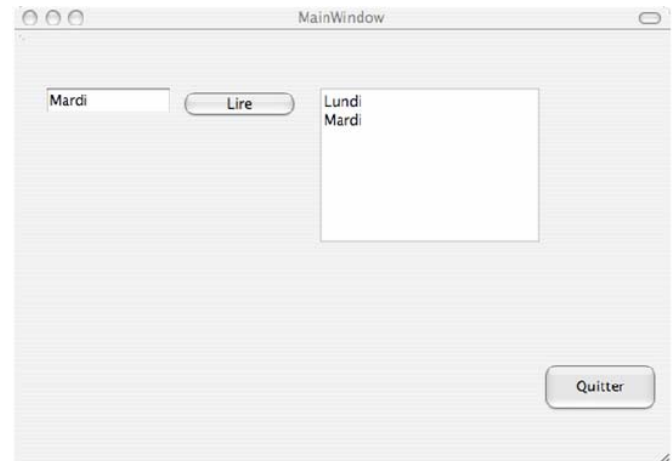
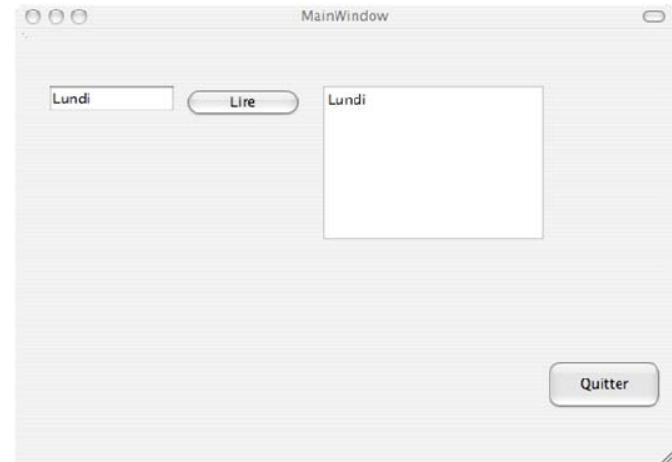
```
void BoutonQuitter();  
void BoutonLire();
```

- Le fichier `mainwindow.cpp` doit être modifié

```
1  #include "mainwindow.h"  
2  #include "ui_mainwindow.h"  
3  #include <QString.h>  
4  
5  MainWindow::MainWindow(QWidget *parent)  
6  : QMainWindow(parent), ui(new Ui::MainWindowClass)  
7  {  
8      ui->setupUi(this);  
9      connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(BoutonQuitter()));  
10     connect(ui->pushButton_2, SIGNAL(clicked()), this, SLOT(BoutonLire()));  
11 }  
12  
13 MainWindow::~MainWindow()  
14 {  
15     delete ui;  
16 }  
17  
18  
19 void MainWindow::BoutonQuitter()  
20 {  
21     this->close();  
22 }  
23  
24 void MainWindow::BoutonLire()  
25 {  
26     QString chaine;  
27     chaine = ui->lineEdit->text();  
28     ui->textEdit->append(chaine);  
29 }  
30
```

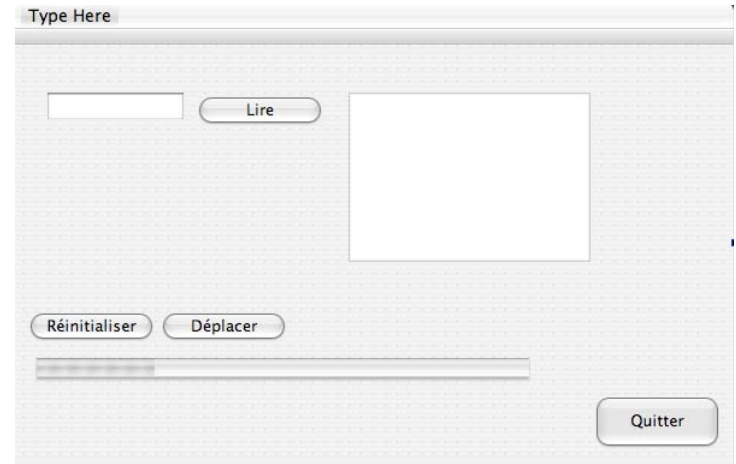
## 2.6) Afficher des messages sur la fenêtre

- On peut constater que le texte du `LineEdit` est bien ajouté au fur et à mesure dans la zone de type `TextEdit`.
- Sur cet exemple on note la différence entre une zone de type `LineEdit` et une zone de type `TextEdit`.



## 2.7) Utilisation des barres de progression

- Ajout sur la fenêtre d'une barre de progression qui se trouve dans la section « Display Widgets ».
- Ajout de 3 boutons
- Comme précédemment, il faut modifier le fichier `mainwindow.h` :



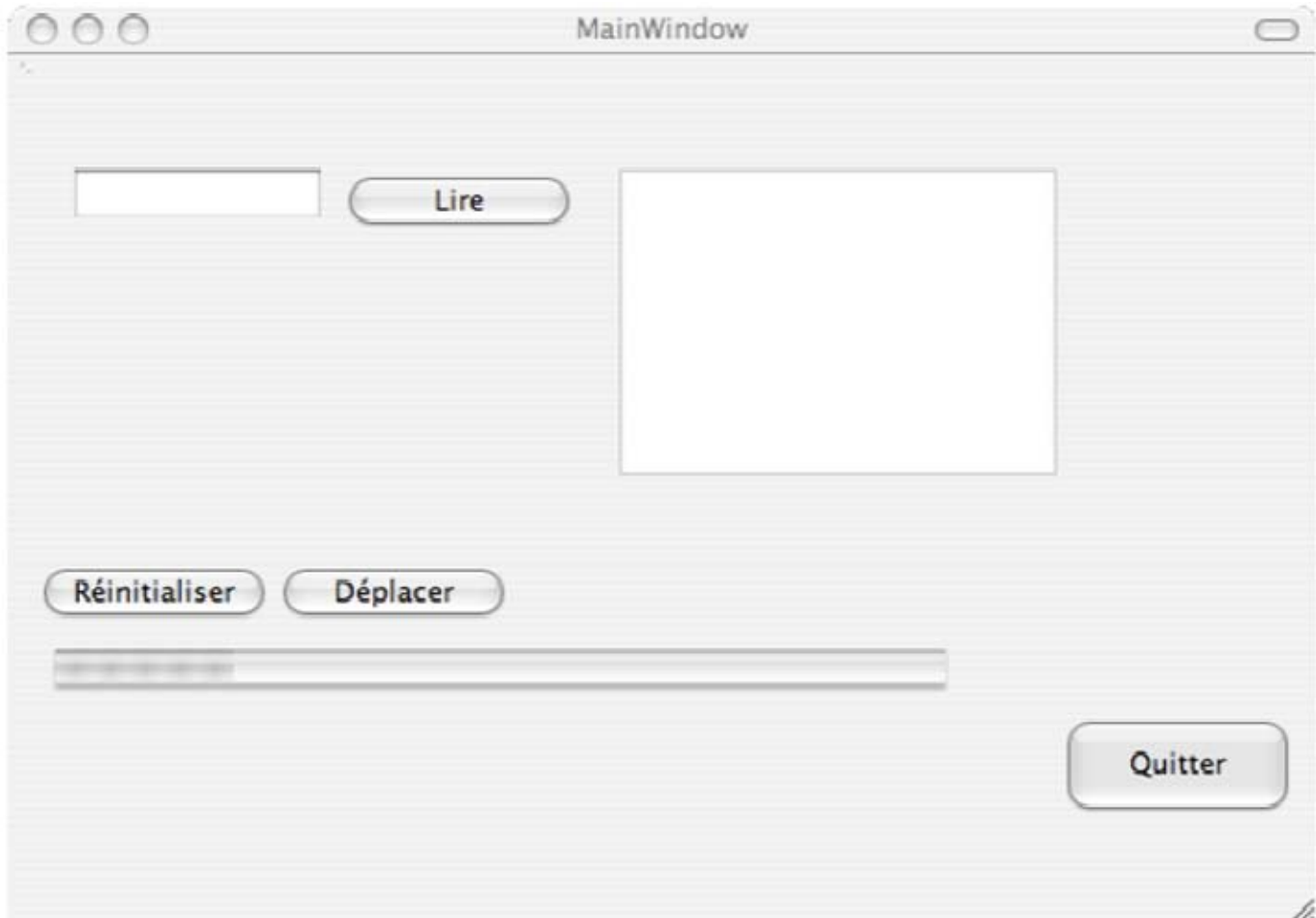
```
void BoutonQuitter();  
void BoutonLire();  
void BoutonReinitialiser();  
void BoutonAugmenter();
```

# 2.7) Utilisation des barres de progression

Le code du fichier `mainwindow.cpp` est modifié

```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3  #include <qstring.h>
4
5  MainWindow::MainWindow(QWidget *parent)
6  {
7      : QMainWindow(parent), ui(new Ui::MainWindowClass)
8      {
9          ui->setupUi(this);
10         connect(ui->pushButton,SIGNAL(clicked()),this,SLOT(BoutonQuitter()));
11         connect(ui->pushButton_2,SIGNAL(clicked()),this,SLOT(BoutonLire()));
12         connect(ui->pushButton_3,SIGNAL(clicked()),this,SLOT(BoutonReinitialiser()));
13         connect(ui->pushButton_4,SIGNAL(clicked()),this,SLOT(BoutonAugmenter()));
14     }
15 }
16
17 MainWindow::~MainWindow()
18 {
19     delete ui;
20 }
21
22
23 void BoutonReinitialiser()
24 {
25     ui->progressBar->setValue(0);
26 }
27
28 void BoutonAugmenter()
29 {
30     int i = ui->progressBar->value();
31     i=i+10;
32     ui->progressBar->setValue(i);
33 }
34
```

## 2.7) Utilisation des barres de progression



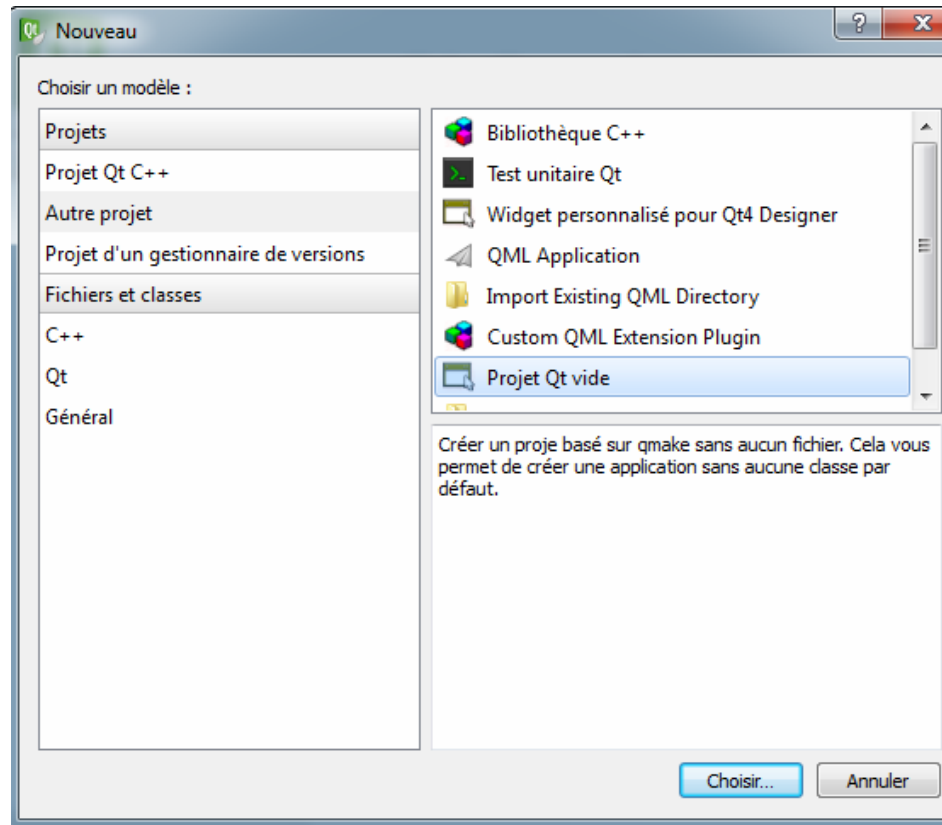
### 3) Les éléments importants d'un projet

- Le fichier `.pro`.
- Les fichiers de conception graphique (`ui`).
- Les fichiers de gestion d'internationalisation (`ts` et `qm`).
- Les types de bases du langage (`qint`, `qfloat` ...).
- La compilation avec `qmake`.
- La classe `QObject`



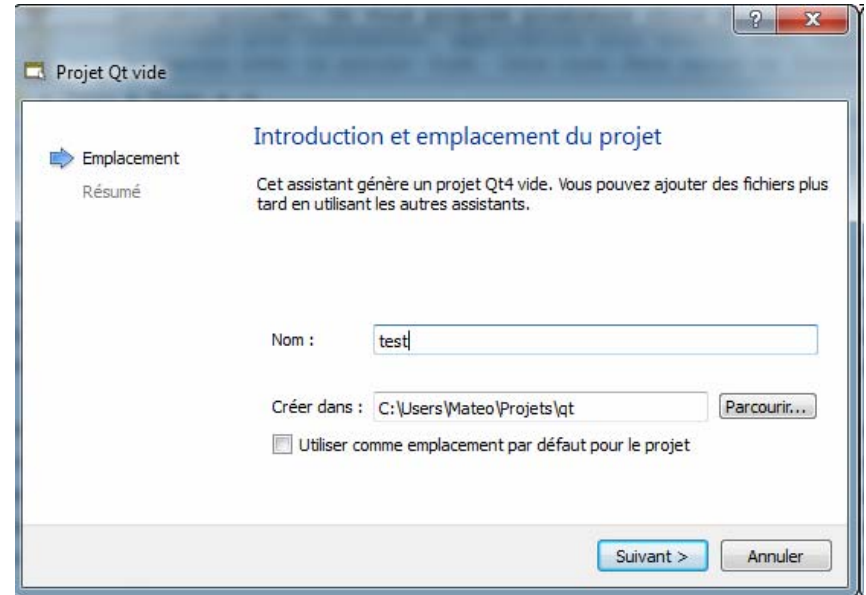
# 3) Les éléments importants d'un projet

- Pour créer un nouveau projet : menu Fichier / Nouveau fichier ou projet.
- Plusieurs choix sont proposé :
  - application graphique pour ordinateur,
  - application pour mobile,
  - etc.
- Choisir donc les options  
Autre projet puis  
Projet Qt vide



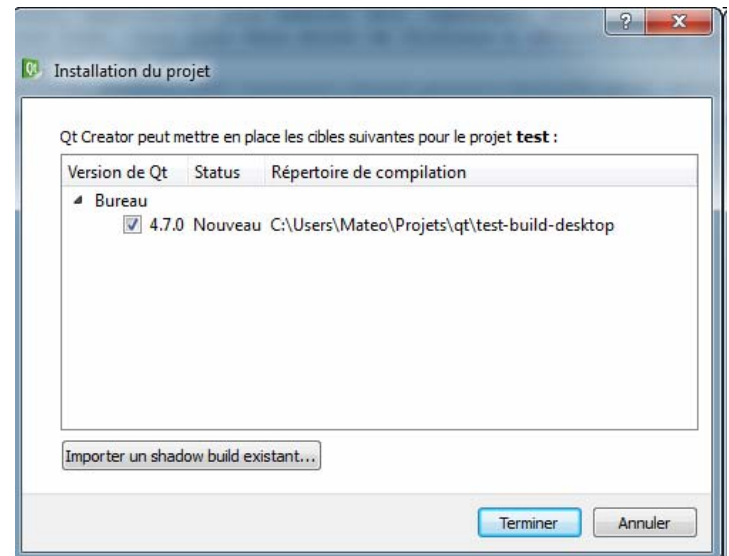
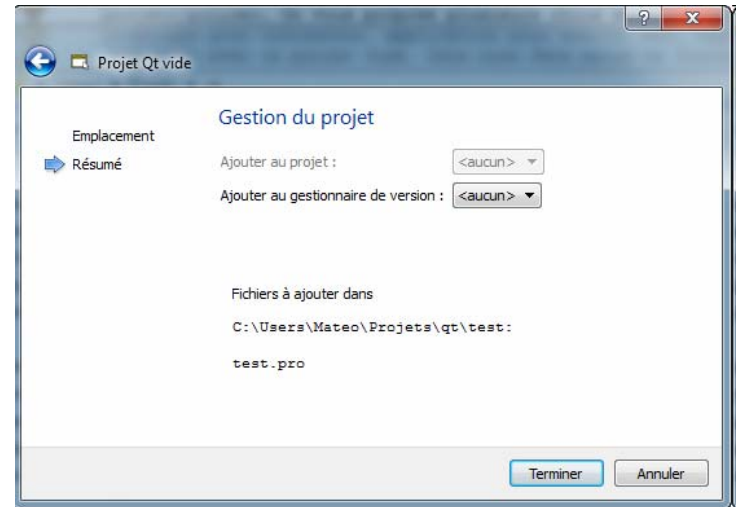
# 3.1) Création d'un projet Qt vide

- Un assistant demande le nom du projet et l'emplacement où l'enregistrer,
- La fenêtre suivante vous demande quelques informations
- Nom du projet: **test**

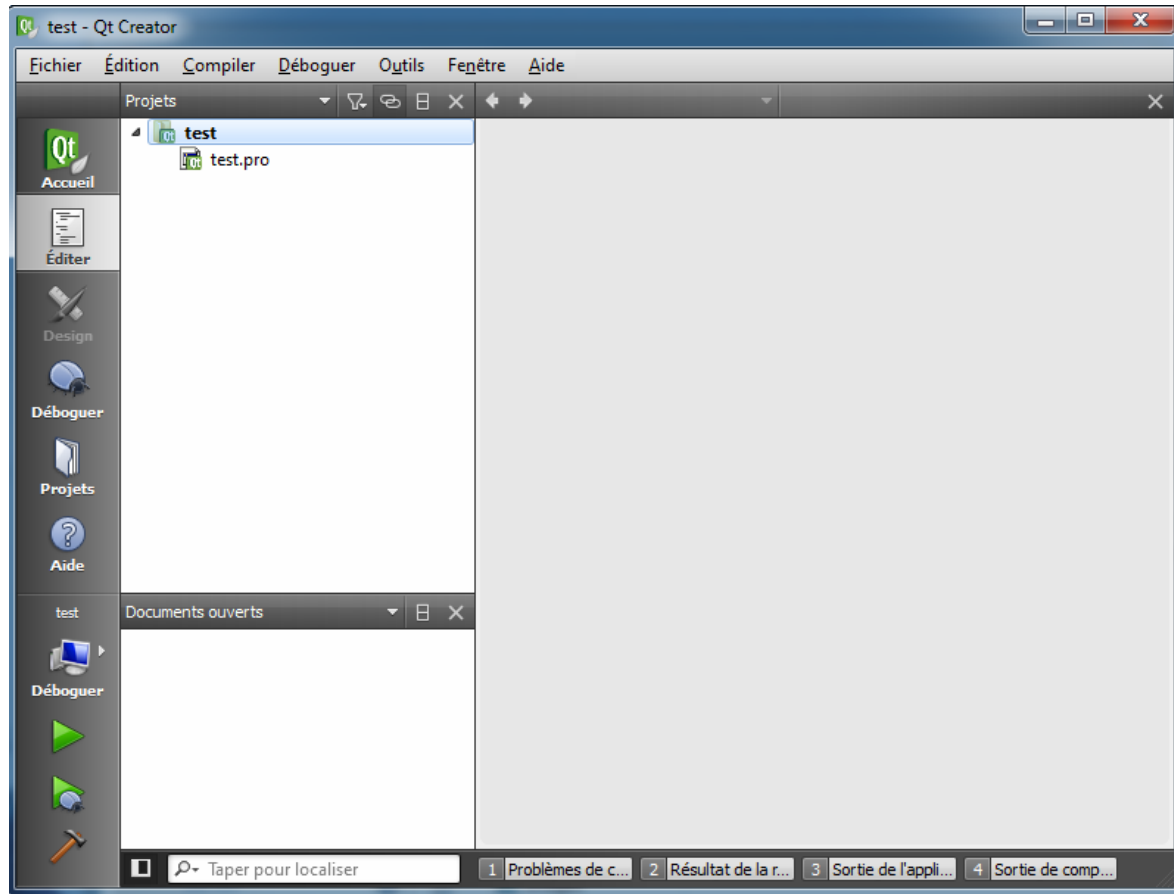


# 3.1) Création d'un projet Qt vide

- Il est possible d'associer le projet à un gestionnaire de version
  - CVS,
  - SVN,
  - Git.
- Qt Creator propose de configurer la compilation.



## 3.2) Développement de projet



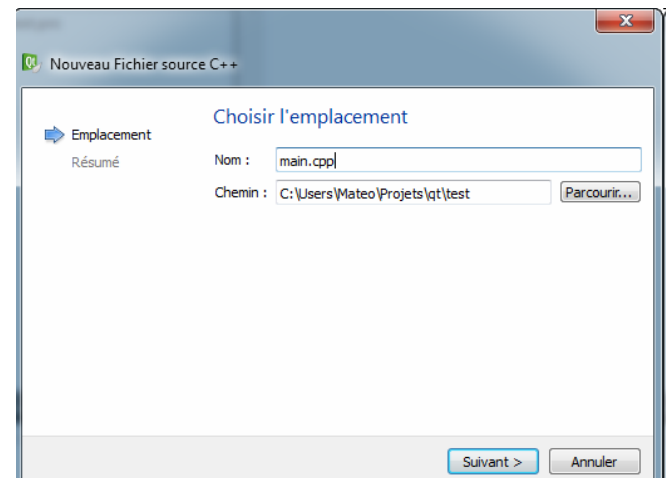
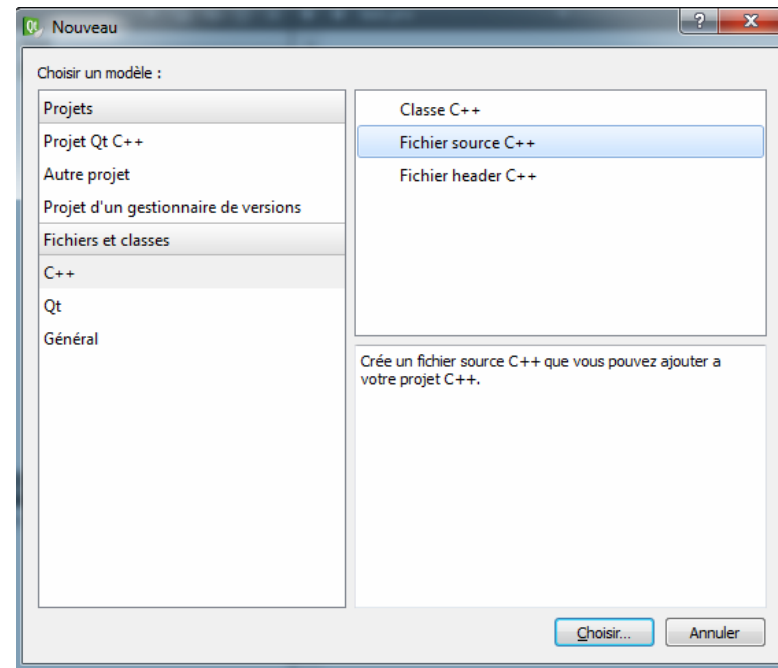
Le projet est constitué seulement d'un fichier `.pro`.

Ce fichier, propre à **Qt**, sert à configurer le projet au moment de la compilation.

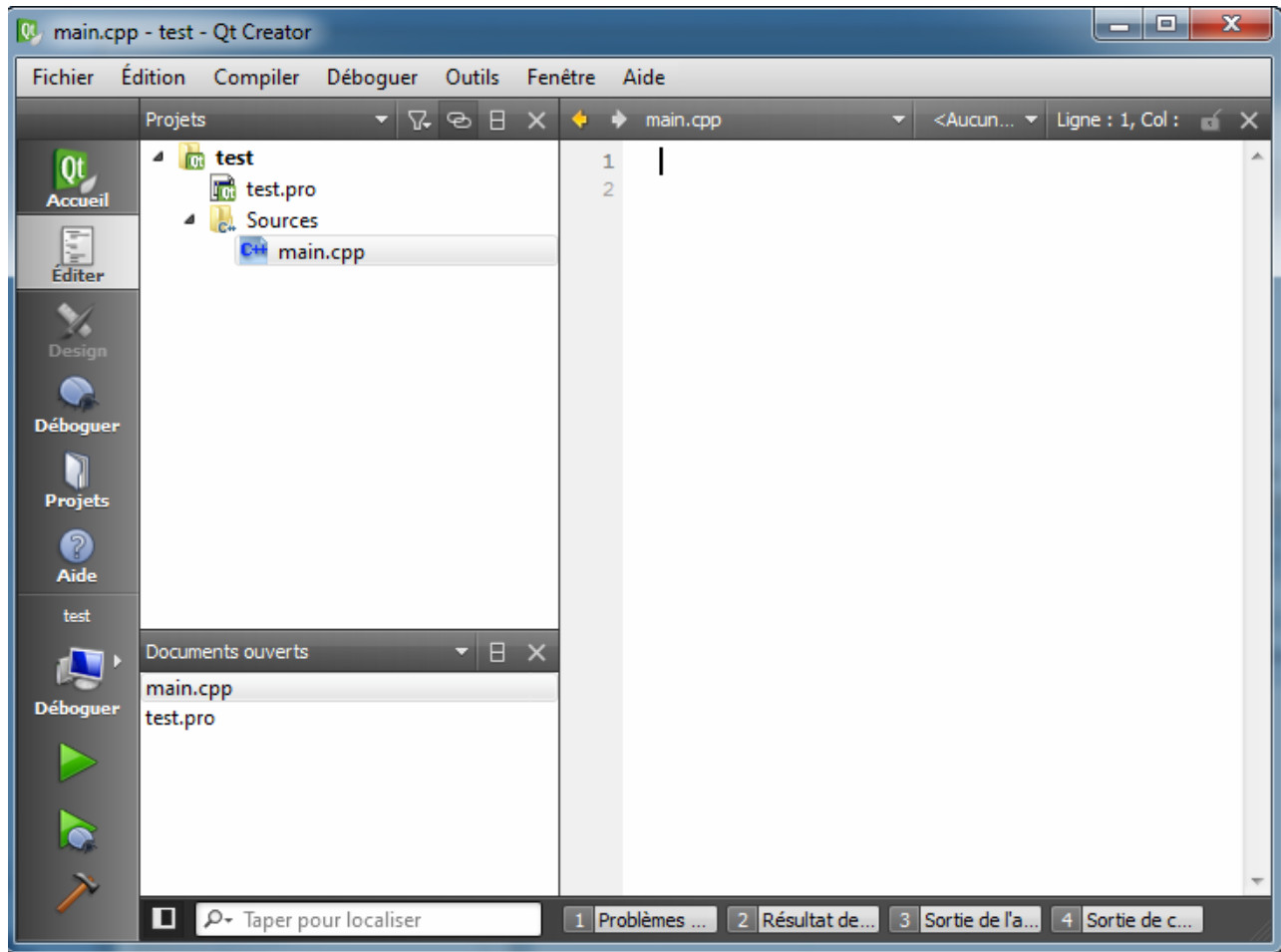
**Qt Creator** modifie automatiquement ce fichier en fonction des besoins

# 3.3) Ajout d'un fichier main.cpp

- dans le menu Fichier / Nouveau fichier ou projet et sélectionnez cette fois C++ / Fichier source C++ pour ajouter un fichier .cpp :
- le nom du fichier à créer, indiquez `main.cpp`



## 3.3) Ajout d'un fichier main.cpp



Le fichier `main.cpp` vide a été ajouté au projet.

C'est dans ce fichier que sera écrit les premières lignes de code C++ utilisant Qt.

Pour compiler le programme, il faut cliquer sur la flèche verte dans le colonne à gauche

ou bien d'utiliser le raccourci clavier `Ctrl + B`

# 3.4) Le code minimal d'un projet

## Qt

main.cpp

- Code minimal d'une application utilisant Qt
- Le seul `include` utile.
  - `iostream` n'est plus utile,
  - permet d'accéder à la classe `QApplication`, c'est la classe de base de tout programme Qt,
- La 1<sup>ère</sup> ligne du `main` crée un nouvel objet de type `QApplication`,
  - Le constructeur de `QApplication` exige de passer les arguments du programme,
- La méthode `exec`
  - démarre le programme et lance l'affichage des fenêtres.
  - retourne le résultat pour dire si le programme s'est bien déroulé ou pas

```
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    return app.exec();
}
```

# 3.5) Affichage d'un widget

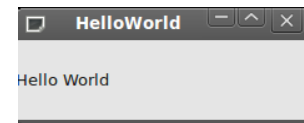
- Tous les éléments d'une fenêtre sont appelés des widgets.
  - les boutons,
  - les cases à cocher,
  - les images...
  - la fenêtre elle-même est considérée comme un widget.
- Les nouvelles lignes
  - permettre de créer des objets de type `QPushButton`
  - créer un nouvel objet de type `QPushButton` appelé `bouton`,
  - le constructeur attend un paramètre : le texte qui sera affiché sur le bouton.
  - l'affichage d'un bouton, Qt l'insère automatiquement dans une fenêtre.

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Hello World")
    bouton.show();

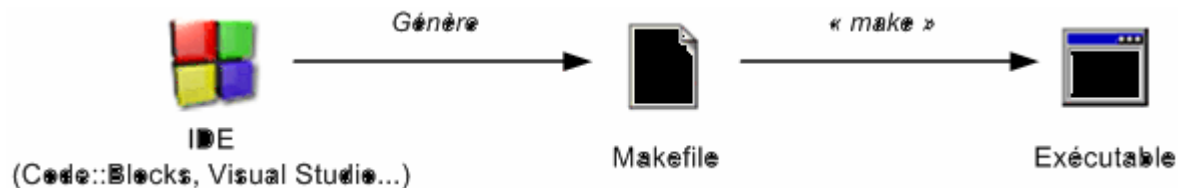
    return app.exec();
}
```





# 3.6) La compilation "normale", sans Qt

- L'IDE regarde la liste des fichiers de votre projet (.cpp et .h) et génère un fichier appelé `Makefile` qui contient la liste des fichiers à compiler pour le compilateur.
- L'IDE appelle le compilateur (via le programme `make`).
  - L'utilitaire `make` recherche un fichier `Makefile` et l'utilise pour savoir quoi compiler et avec quelles options.



```
all: $(PROG)
```

```
$(PROG): $(OBJS)
```

```
$(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)
```

```
.c.o:
```

```
$(CC) $(CFLAGS) -c $*.c
```

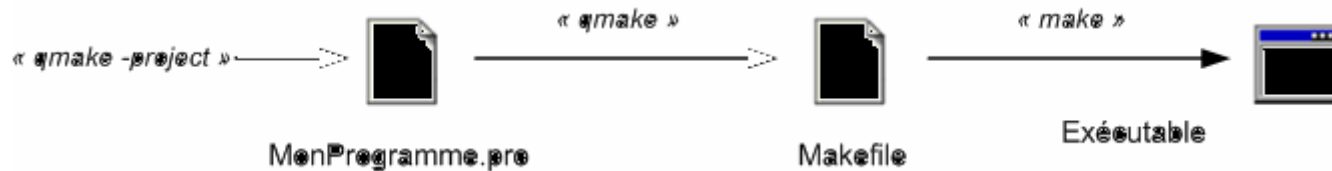
# 3.7) La compilation particulière avec Qt

- **Qt** est livré avec un programme en ligne de commande appelé "qmake".
  - Ce programme est capable de générer un fichier `Makefile` à partir d'un fichier spécifique à Qt : le `.pro`.
- Le `.pro` (nommé `nomDeProjet.pro`) est un fichier texte court à écrire qui donne la liste des fichiers `.cpp` et `.h`, ainsi que les options à envoyer à Qt.



# 3.7) La compilation particulière avec Qt

- Qt peut vous générer un `.pro` automatiquement !
  - Si on utilise d'abord `qmake` avec l'option `-project` dans le dossier du projet,
  - `qmake` va analyser les fichiers du dossier et générer un fichier `.pro` basique.



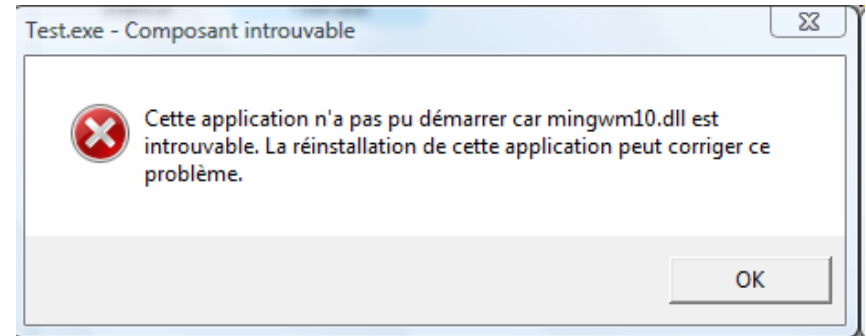
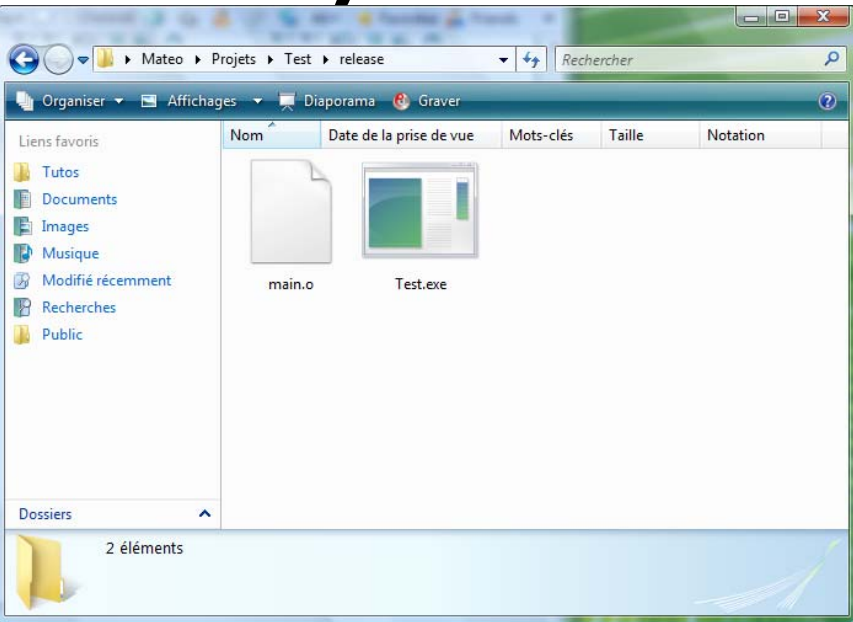
pour compiler avec Qt il y a 3 commandes très simples à taper en console.  
Dans l'ordre :

```
qmake -project
```

```
qmake
```

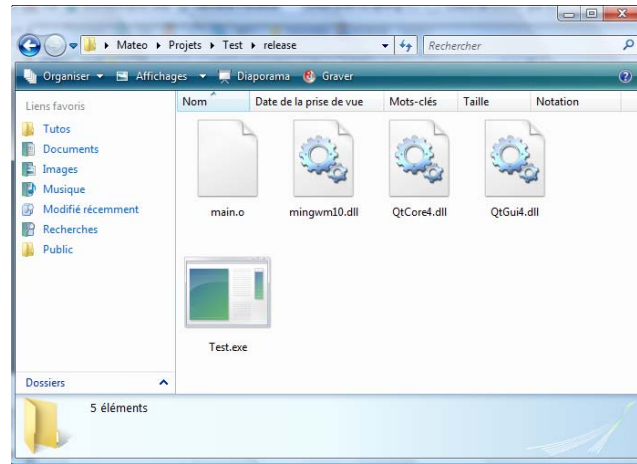
```
make (sous Linux) ou mingw32-make (sous Windows)
```

## 3.8) Diffuser le programme



- Pour tester le programme avec Qt Creator, un clic sur la flèche verte suffit.
  - Les programmes **Qt** ont besoin de ces fichiers DLL avec eux pour fonctionner.
  - Pour lancer un programme depuis Qt Creator, la position des DLL est "connue", donc le programme se lance sans erreur.

## 3.8) Diffuser le programme



- Pour pouvoir lancer l'exécutable depuis l'explorateur, il faut placer les DLL qui manquent dans le même dossier que l'exécutable.
  - Par ex, dans le dossier `C:\Qt\2011.05\mingw\bin` et `C:\Qt\2015.05\bin`).
  - Il y a 3 DLLs :

# 3.9) Structure générale

- L'API Qt est constituée de classes aux noms préfixés par Q et dont chaque mot commence par une majuscule (ex: QLineEdit),
  - Ces classes ont souvent pour attributs des types énumérés déclarés dans l'espace de nommage Qt15.
  - Mis à part une architecture en pur objet, certaines fonctionnalités basiques sont implémentées par des macros,
- Les conventions de nommage des méthodes sont assez semblables à celles de Java : le `lowerCamelCase` est utilisé,
  - ex: `indicatorFollowsStyle()`,
  - les modificateurs sont précédés par `set`,
  - en revanche les accesseurs prennent simplement le nom de l'attribut (ex : `text()`) ou commencent par `is` dans le cas des booléens (ex : `isChecked()`).

## 3.10) Arborescence des objets

- Les objets Qt (hérite de `QObject`) peuvent s'organiser d'eux-mêmes sous forme d'arbre.
  - lorsqu'une classe est instanciée, on peut lui définir un objet parent.
  - cette organisation des objets sous forme d'arbre facilite la gestion de la mémoire car avant qu'un objet parent ne soit détruit, Qt appelle récursivement le destructeur de tous les enfants
- Cette notion d'arbre des objets permet de déboguer plus facilement, via l'appel de méthodes comme
  - `QObject::dumpObjectTree()`
  - `QObject::dumpObjectInfo()`

# 3.11) Compilateur de meta-objets

- Le `moc` (pour Meta Object Compiler) est un préprocesseur qui, appliqué avant compilation du code source d'un programme Qt, génère des meta-informations relatives aux classes utilisées dans le programme.
  - Ces meta-informations sont utilisées par Qt pour fournir des fonctions non disponibles en C++, comme les signaux et slots et l'introspection.
- L'utilisation d'un tel outil démarque les programmes Qt du langage C++ standard.
  - Ce fonctionnement est vu par Qt Development Frameworks comme un compromis nécessaire pour fournir l'introspection et les mécanismes de signaux.
  - À la sortie de Qt 1.x, les implémentations des templates par les compilateurs C++ n'étaient pas suffisamment homogènes



## 3.12) Signaux et slots

- Les signaux et slots sont une implémentation du patron de conception observateur.
  - L'idée est de connecter des objets entre eux via des signaux qui sont émis et reçus par des slots.
  - Du point de vue du développeur, les signaux sont représentés comme de simples méthodes de la classe émettrice, dont il n'y a pas d'implémentation.
  - Ces "méthodes" sont par la suite appelées, en faisant précéder "emit", qui désigne l'émission du signal.
  - Pour sa part, le slot connecté à un signal est une méthode de la classe réceptrice, qui doit avoir la même signature (autrement dit les mêmes paramètres que le signal auquel il est connecté), mais à la différence des signaux, il doit être implémenté par le développeur.
  - Le code de cette implémentation représente les actions à réaliser à la réception du signal.
- C'est le MOC qui se charge de générer le code C++ nécessaire pour connecter les signaux et les slots.

# 3.13) Concepteur d'interface

- Qt Designer est un logiciel qui permet de créer des interfaces graphiques Qt dans un environnement convivial.
  - L'utilisateur, par glisser-déposer, place les composants d'interface graphique et y règle leurs propriétés facilement.
  - Les fichiers d'interface graphique sont formatés en XML et portent l'extension `.ui`.
- Lors de la compilation, un fichier d'interface graphique est converti en classe C++ par l'utilitaire `uic`. Il y a plusieurs manières pour le développeur d'employer cette classe :
  - l'instancier directement et connecter les signaux et slots
  - l'agréger au sein d'une autre classe
  - l'hériter pour en faire une classe mère et ayant accès ainsi à tous les éléments constitutifs de l'interface créée
  - la générer à la volée avec la classe `QUiLoader` qui se charge d'interpréter le fichier XML `.ui` et retourner une instance de classe `QWidget`

## 3.14) qmake

- Qt est un cadre de développement portable et ayant le MOC comme étape intermédiaire avant la phase de compilation/édition de liens,
  - un moteur de production spécifique est le programme `qmake`.
  - il prend en entrée un fichier (avec l'extension `.pro`) décrivant le projet
    - liste des fichiers sources,
    - dépendances,
    - paramètres passés au compilateur,
    - etc.
  - il génère un fichier de projet spécifique à la plateforme.
- Le fichier de projet est fait pour être très facilement éditable par un développeur.
  - Il consiste en une série d'affectations de variables. En voici un exemple pour un petit projet:

```
TARGET = monAppli
SOURCES = main.cpp mainwindow.cpp
HEADERS = mainwindow.h
FORMS = mainwindow.ui
QT += sql
```

# 3.15) Internationalisation

- Qt intègre son propre système de traduction,
  - Selon le manuel de Qt Linguist, l'internationalisation est assurée par la collaboration de trois types de personnes : les développeurs, le chef de projet et les traducteurs
- Dans leur code source, les développeurs entrent des chaînes de caractères dans leur propre langue. Ils doivent permettre la traduction de ces chaînes grâce à la méthode `tr()`.
  - En cas d'ambiguïté sur le sens d'une expression, ils peuvent également indiquer des commentaires destinés à aider les traducteurs.
- Le chef de projet déclare les fichiers de traduction (un pour chaque langue) dans le fichier de projet.
  - L'utilitaire `lupdate` parcourt les sources à la recherche de chaînes à traduire et synchronise les fichiers de traduction avec les sources.
  - Les fichiers de traductions sont des fichiers XML portant l'extension `.ts`.
- Les traducteurs utilisent `Qt Linguist` pour renseigner les fichiers de traduction.
  - Quand les traductions sont finies, le chef de projet peut compiler les fichiers `.ts` à l'aide de l'utilitaire `lrelease` qui génère des fichiers binaires portant l'extension `.qm`, exploitables par le programme.
  - Ces fichiers sont lus à l'exécution et les chaînes de caractères qui y sont trouvées remplacent celles qui ont été écrites par les développeurs.

## 3.16) Style

- La bibliothèque embarque divers thèmes de widgets qui lui donnent une bonne intégration visuelle sur toutes les plateformes.
  - Sur les environnements de bureau GNOME, Mac OS X et Windows les applications Qt ont ainsi l'apparence d'applications natives.
- Qt permet de personnaliser l'apparence des différents composants d'interface graphique en utilisant le principe des feuilles de style en cascade (CSS)

# 3.17) Le modèle Objet de Qt

- Qt est basé autour du modèle d'objet de Qt.
  - Elle est entièrement basée autour de la classe `QObject` et de l'outil `moc`.
- En dérivant des classes de `QObject` un certain nombre d'avantages sont hérités:
  - Gestion facile de la mémoire.
  - Signaux et Slots.
  - Propriétés.
  - Introspection.
- Rappel:
  - Qt est du C++ standard avec quelques macros, juste comme n'importe quelle autre application de C/C++.
  - Il n'y a rien de non standard avec Qt, ce qui est le mieux illustré par le fait que le code est très portable.

## 3.18) Gestion Facile de la Mémoire

- En créant une instance d'une classe dérivée de `QObject` il est possible de passer un pointeur vers un objet parent au constructeur.
  - C'est la base de la gestion simplifiée de la mémoire.
  - Quand un parent est supprimé, ses enfants sont supprimés aussi.

```
class VerboseObject : public QObject {
public:
    VerboseObject( QObject *parent=0, char *name=0 )
        : QObject( parent, name ) {
        std::cout << "Created: " << QObject::name() << std::endl;
    }
    ~VerboseObject() {
        std::cout << "Deleted: " << name() << std::endl;
    }
    void doStuff() {
        std::cout << "Do stuff: " << name() << std::endl;
    }
};
```

# 3.18) Gestion Facile de la Mémoire

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

```
{
```

```
    // Create an application
```

```
    QApplication a( argc, argv );
```

```
    // Create instances
```

```
    VerboseObject top( 0, "top" );
```

```
    VerboseObject *x = new VerboseObject( &top, "x" );
```

```
    VerboseObject *y = new VerboseObject( &top, "y" );
```

```
    VerboseObject *z = new VerboseObject( x, "z" );
```

```
    // Do stuff to stop the optimizer
```

```
    top.doStuff();
```

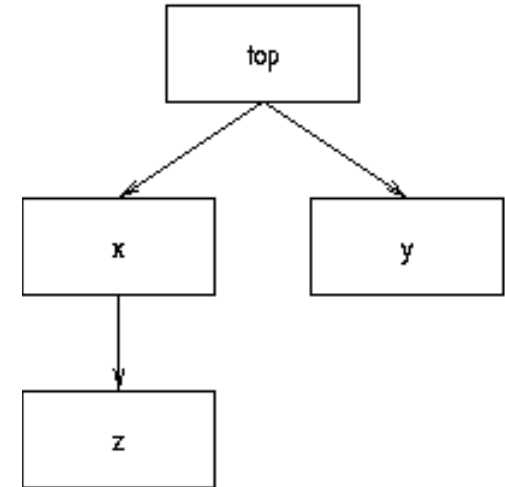
```
    x->doStuff();
```

```
    y->doStuff();
```

```
    z->doStuff();
```

```
    return 0;
```

```
}
```



```
$ ./mem
```

```
Created: top
```

```
Created: x
```

```
Created: y
```

```
Created: z
```

```
Do stuff: top
```

```
Do stuff: x
```

```
Do stuff: y
```

```
Do stuff: z
```

```
Deleted: top
```

```
Deleted: x
```

```
Deleted: z
```

```
Deleted: y
```

```
$
```

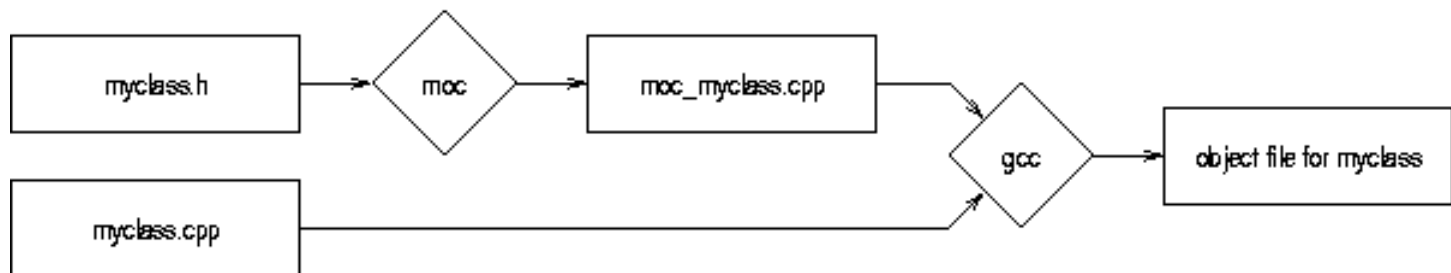


## 3.19) Signaux et Slots

- Les signaux et les slots sont ce qui rend les différents composants de Qt aussi réutilisables qu'ils le sont. Ils fournissent un mécanisme par lequel il est possible de librement relier ensemble les interfaces.
  - par exemple, une entrée de menu, un bouton poussoir, un bouton de barre d'outils et n'importe quel autre élément peuvent fournir le signal correspondant à l'événement approprié "activé", "cliqué" ou n'importe quel autre.
  - en reliant un **signal** aux slots de tous autres éléments et l'événement appelle automatiquement les **slots**.
- Un signal peut également inclure des valeurs, de ce fait permettant de relier une réglette, un `spinbox`, un bouton ou n'importe quelle autre valeur produite par l'élément à n'importe quel élément acceptant des valeurs,
  - par exemple réglette, bouton ou un `spinbox`, ou une quelque chose de complètement différent comme un affichage à cristaux liquides.

## 3.19) Signaux et Slots

- Pour employer les signaux et les slots chaque classe doit être déclarée dans un fichier d'en-tête.
  - L'implémentation est mieux placée dans un fichier cpp séparé.
  - Le fichier d'en-tête est alors passé par un outil de Qt connu sous le nom de moc.
  - Le moc produit un cpp contenant le code qui permet la prise en compte des signaux et des slots (et plus)



## 3.19) Signaux et Slots

```
class Reciever : public QObject {
    Q_OBJECT

public:
    Reciever( QObject *parent=0, char *name=0 )
        : QObject( parent, name ) {

public slots:
    void get( int x ) {
        std::cout << "Recieved: " << x << std::endl;
    }
};
```

Cette classe commence par la macro `Q_OBJECT` qui est nécessaire si d'autres dispositifs que la gestion simplifiée de mémoire doivent être employés. Cette macro inclut quelques déclarations principales et sert de marqueur au `moc` qui indique que la classe doit être analysée.

## 3.19) Signaux et Slots

```
class SenderA : public QObject {
    Q_OBJECT

public:
    SenderA( QObject *parent=0, char *name=0 )
        : QObject( parent, name ) {

    void doSend() {
        emit( send( 7 ) );
    }

signals:
    void send( int );
};
```

La classe contient les réalisations des classes émettrices. La classe, `SenderA`. met en application le signal `send` qui est émis de la fonction membre `doEmit`.

## 3.19) Signaux et Slots

```
class SenderB : public QObject {
    Q_OBJECT

public:
    SenderB( QObject *parent=0, char *name=0 )
        : QObject( parent, name ) {

        void doStuff() {
            emit( transmit( 5 ) );
        }

signals:
    void transmit( int );
};
```

La classe émettrice, `SenderB` émet le signal `transmit` depuis le fonction membre `doStuff`.

Notez que ces classes ont en commun d'hériter de `QObject`.

## 3.19) Signaux et Slots

```
#include "moc_sisl.h"

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

    Reciever r;
    SenderA sa;
    SenderB sb;

    QObject::connect( &sa, SIGNAL(send(int)), &r, SLOT(get(int)) );
    QObject::connect( &sb, SIGNAL(transmit(int)), &r, SLOT(get(int)) );

    sa.doSend();
    sb.doStuff();

    return 0;
}
```

Pour démontrer comment le code de `moc` est fusionné avec le code écrit par les programmeurs humains, cet exemple n'emploie pas la méthode `qmake` classique, mais inclut à la place le code explicitement.

Pour invoquer `moc`, la ligne de commande suivante est employée :

```
QTDIR/bin/moc sisl.cpp -o moc_sisl.h.
```

## 3.19) Signaux et Slots

- Les signaux sont émis et les receveurs affichent les valeurs.

```
$ ./sisl
```

```
Recieved: 7
```

```
Recieved: 5
```

```
$
```

## 3.20) Propriétés

- Un objet Qt peut avoir des propriétés.
  - Ce sont simplement des valeurs qui ont un type et, au moins une fonction de lecture, mais aussi une fonction d'écriture.
  - Celles-ci, sont employées par `Qt Designer` pour montrer les propriétés de tous les widgets.
- Les propriétés sont non seulement une bonne manière d'organiser le code et de définir quelle fonction affecte quelle propriété.
  - Elles peuvent également être employées comme forme primitive de réflexion.
  - N'importe quel pointeur de `QObject` peut accéder aux propriétés de l'objet qu'il pointe.

```
#include "propobject.h"
```

```
PropObject::PropObject( QObject *parent, char *name )  
    : QObject( parent, name ) {  
    m_testProperty = InitialValue;  
}
```

```
void PropObject::setTestProperty( TestProperty p ) {  
    m_testProperty = p;  
}
```

```
PropObject::TestProperty PropObject::testProperty() const {  
    return m_testProperty;  
}
```



## 3.20) Propriétés

```
#ifndef PROPOBJECT_H
#define PROPOBJECT_H

#include <qobject.h>
#include <qvariant.h>

class PropObject : public QObject
{
    Q_OBJECT

    Q_PROPERTY( TestProperty testProperty READ testProperty WRITE setTestProperty )
    Q_ENUMS( TestProperty )
    Q_PROPERTY( QString anotherProperty READ anotherProperty )

public:
    PropObject( QObject *parent=0, char *name=0 );
    enum TestProperty { InitialValue, AnotherValue };
    void setTestProperty( TestProperty p );
    TestProperty testProperty() const;
    QString anotherProperty() const { return QString( "I'm read-only!" ); }

private:
    TestProperty m_testProperty;
};

#endif
```

## 3.20) Propriétés

```
#include <qapplication.h>
#include <iostream>

#include "propobject.h"

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

    QObject *o = new PropObject();

    std::cout << o->property( "testProperty" ).toString() << std::endl;

    o->setProperty( "testProperty", "AnotherValue" );
    std::cout << o->property("testProperty" ).toString() << std::endl;

    std::cout << o->property("anotherProperty" ).toString() << std::endl;

    return 0;
}
```

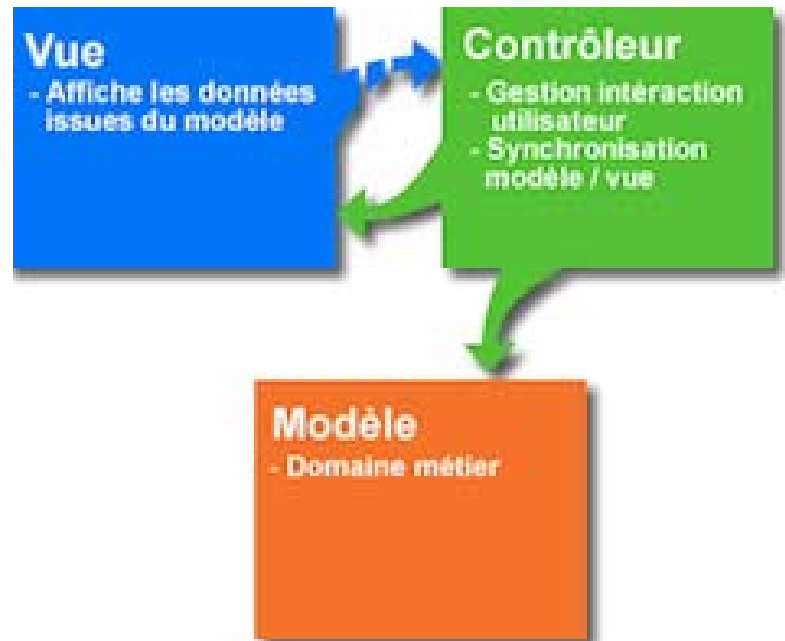
```
$ ./prop
0
1
I'm read-only!
$
```

# 3.21) Introspection

- Chaque objet de Qt a un méta-objet. Cet objet est représenté par une instance de la classe `QMetaObject`. Il est employé pour fournir des informations au sujet de la classe courante.
  - Le méta-objet peut être consulté par la fonction de membre `QObject::metaObject()`.
  - Le méta-objet fournit quelques fonctions utiles énumérées ci-dessous.
- `QMetaObject::className()`  
Donne le nom de la classe, par exemple `PropObject` dans l'exemple précédent.
- `QMetaObject::superClass()`  
Donne le méta-objet de la classe supérieure, ou 0 (nul) s'il n'y en a aucun.
- `QMetaObject::propertyNames()` and `QMetaObject::property()`  
Donne les noms des noms des propriétés et les méta-données pour chaque propriété comme `QMetaProperty`.
- `QMetaObject::slotNames()`  
Donne les noms des slots de la classe. Si le paramètre facultatif, `super`, est positionné à `true` les slots de la classe supérieure sont inclus aussi.
- `QMetaObject::signalNames()`  
Donne les noms des signaux de la classe. Un paramètre facultatif, `super`, est disponible quant à la fonction membre `signalNames`.

# 4) Le modèle MVC avec QT

- Présentation du modèle MVC.
- Le modèle MVC dans QT.

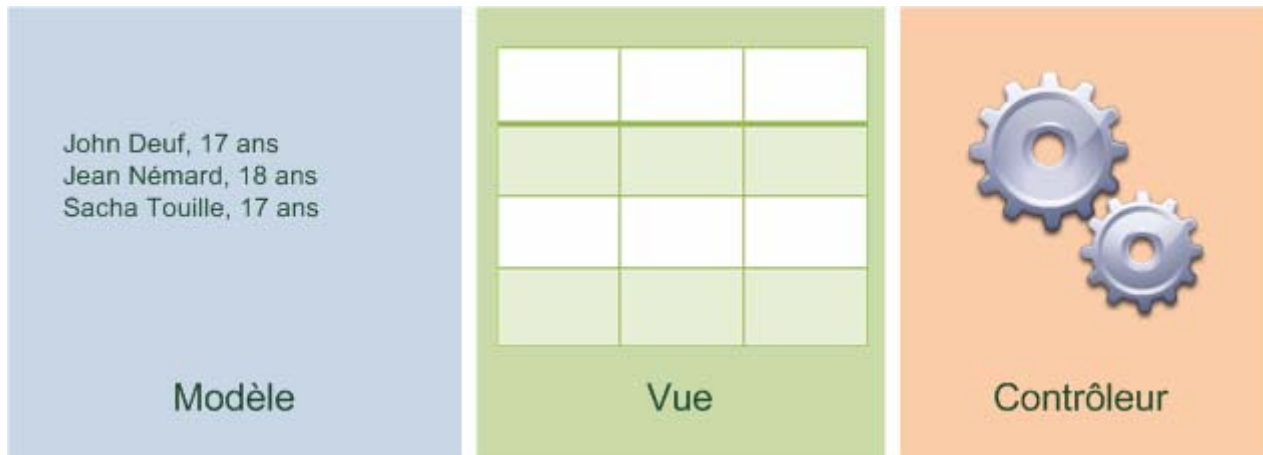


# 4) Le modèle MVC avec QT

- MVC est l'abréviation de Model-View-Controller,
- Il s'agit d'un design pattern, une technique de programmation.
- L'architecture MVC propose de séparer les éléments d'un programme en 3 parties :
  - **Le modèle** : c'est la partie qui contient les données.
    - Le modèle peut par exemple contenir la liste des élèves d'une classe, avec leurs noms, prénoms, âges...
  - **La vue** : c'est la partie qui s'occupe de l'affichage. Elle affiche ce que contient le modèle.
    - Par exemple, la vue pourrait être un tableau. Ce tableau affichera la liste des élèves si c'est ce que contient le modèle.
  - **Le contrôleur** : c'est la partie "réflexion" du programme.
    - Lorsque l'utilisateur sélectionne 3 élèves dans le tableau et appuie sur la touche "Supprimer", le contrôleur est appelé et se charge de supprimer les 3 élèves du modèle.

# 4.1) Présentation de l'architecture MVC

- l'intérêt de séparer le code en 3 parties et le rôle de chacune de ces parties
  - Le modèle est la partie qui contient les données. Les données sont généralement récupérées en lisant un fichier ou une base de données.
  - La vue est juste la partie qui affiche le modèle, ce sera donc un widget. Si un élément est ajouté au modèle (par exemple un nouvel élève apparaît) la vue se met à jour automatiquement pour afficher le nouveau modèle.
  - Le contrôleur est la partie la plus algorithmique, c'est-à-dire le cerveau du programme. S'il y a des calculs à faire, c'est là qu'ils sont faits.

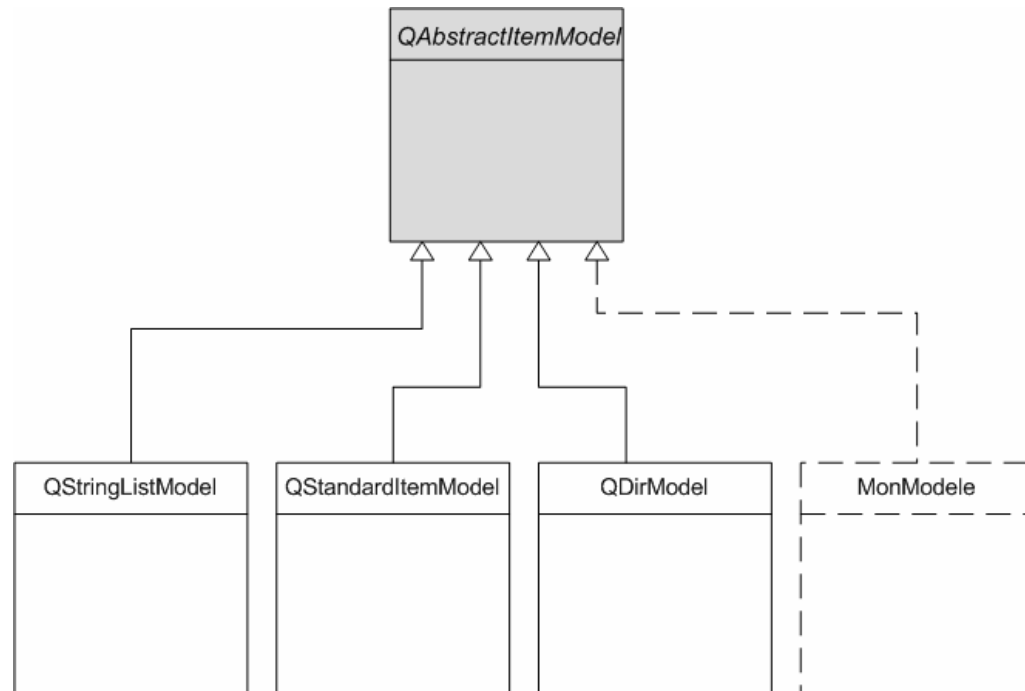


## 4.2) Les classes gérant le modèle

- Il y a plusieurs types de modèles différents. En effet : on ne stocke pas de la même manière une liste d'élèves qu'une liste de villes.
- 2 possibilités :
  - Soit. Il faut créer une classe héritant de `QAbstractItemModel`. C'est la solution la plus flexible mais aussi la plus complexe.
  - Soit une des classes génériques toutes prêtes offertes par Qt :
- `QStringListModel` : une liste de chaînes de caractères, de type `QString`. Très simple, très basique.
- `QStandardItemModel` : une liste d'éléments organisés sous forme d'arbre (chaque élément peut contenir des sous-éléments). Ce type de modèle est plus complexe que le précédent, car il gère plusieurs niveaux d'éléments. Avec `QStringListModel`, c'est un des modèles les plus utilisés.
- `QDirModel` : la liste des fichiers et dossiers stockés sur votre ordinateur. Ce modèle va analyser en arrière-plan votre disque dur, et restitue la liste de vos fichiers sous la forme d'un modèle prêt à l'emploi.
- `QSqlQueryModel`, `QSqlTableModel` et `QSqlRelationalTableModel` : données issues d'une base de données. On peut s'en servir pour accéder à une base de données.

## 4.2) Les classes gérant le modèle

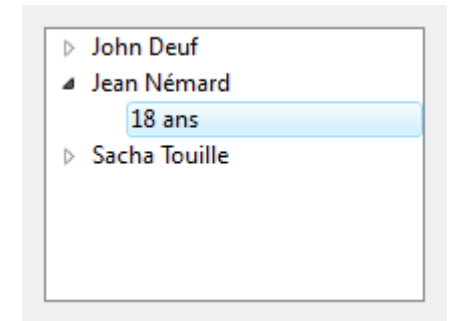
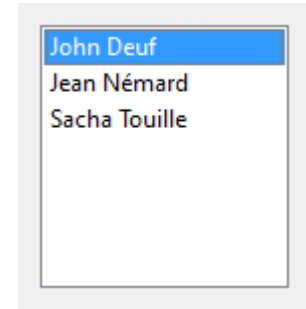
- Toutes ces classes proposent donc des modèles prêts à l'emploi, qui héritent de `QAbstractItemModel`.
- Si aucune de ces classes ne convient, il faut créer sa propre classe en héritant de `QAbstractItemModel`.





## 4.3) Les classes gérant la vue

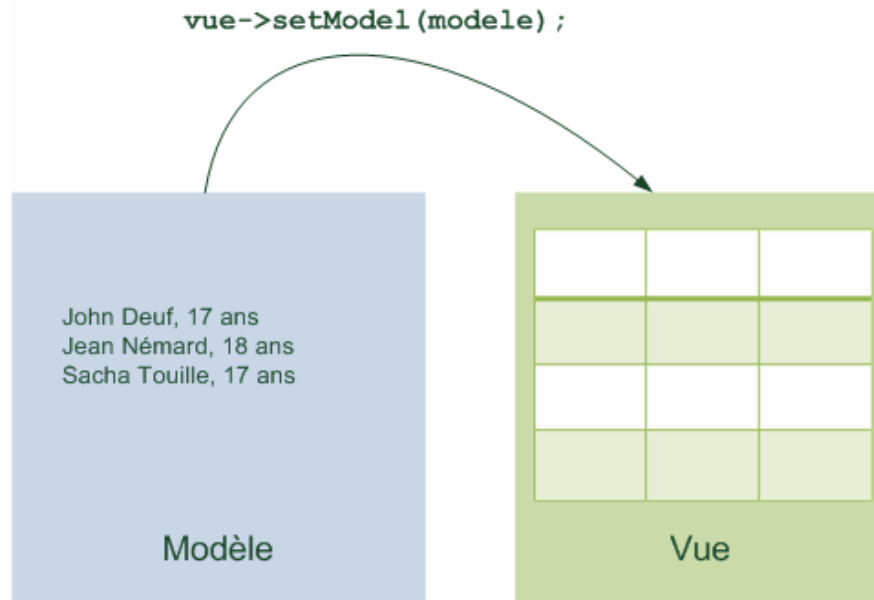
- Pour afficher les données issues du modèle, il faut une vue.  
Avec Qt, la vue est un widget, puisqu'il s'agit d'un affichage dans une fenêtre.
- Tous les widgets de Qt ne sont pas bâtis autour de l'architecture modèle/vue,
- On compte 3 widgets adaptés pour la vue avec Qt :
  - `QListView` : une liste d'éléments.
  - `QTreeView` : un arbre d'éléments, où chaque élément peut avoir des éléments enfants.
  - `QTableView` : un tableau.



John	Deuf	17 ans
Jean	Némard	18 ans
Sacha	Touille	17 ans

# 4.4) Appliquer un modèle à la vue

- Lorsqu'on utilise l'architecture modèle/vue avec Qt, cela se passe toujours en 3 temps. Il faut :
  - Créer le modèle
  - Créer la vue
  - Associer la vue et le modèle
- La dernière étape est essentielle. Cela revient en quelque sorte à "connecter" notre modèle à notre vue. Si on ne donne pas de modèle à la vue, elle ne saura pas quoi afficher,
- La connexion se fait toujours avec la méthode `setModel()` de la vue :



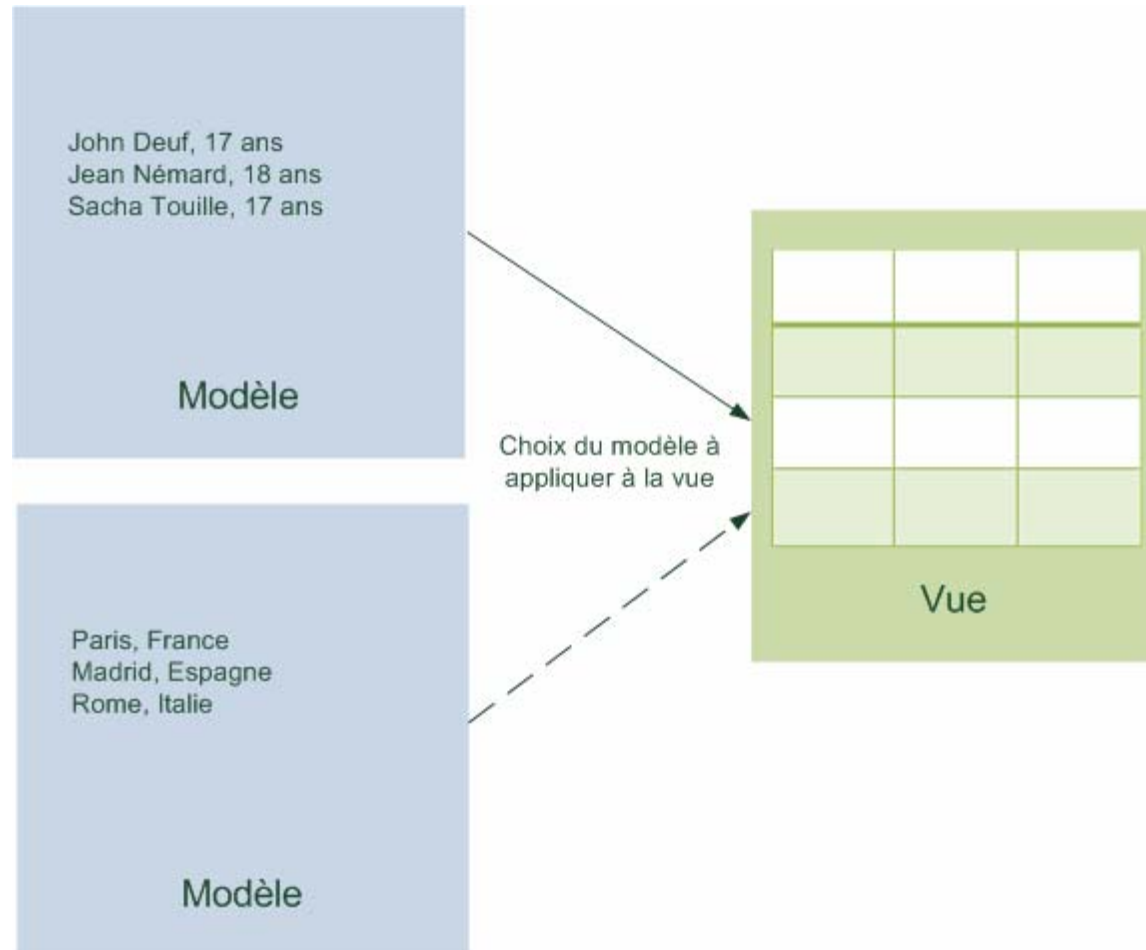
## 4.5) Plusieurs modèles et une vue

Imaginons que l'on ait 2 modèles :

- un qui contient une liste d'élèves,
- un autre qui contient une liste de capitales avec leur pays.

Notre vue peut afficher soit l'un, soit l'autre

Une vue ne peut afficher qu'un seul modèle à la fois. L'avantage, c'est qu'au besoin on peut changer le modèle affiché par la vue en cours d'exécution, en appelant juste la méthode `setModel()`

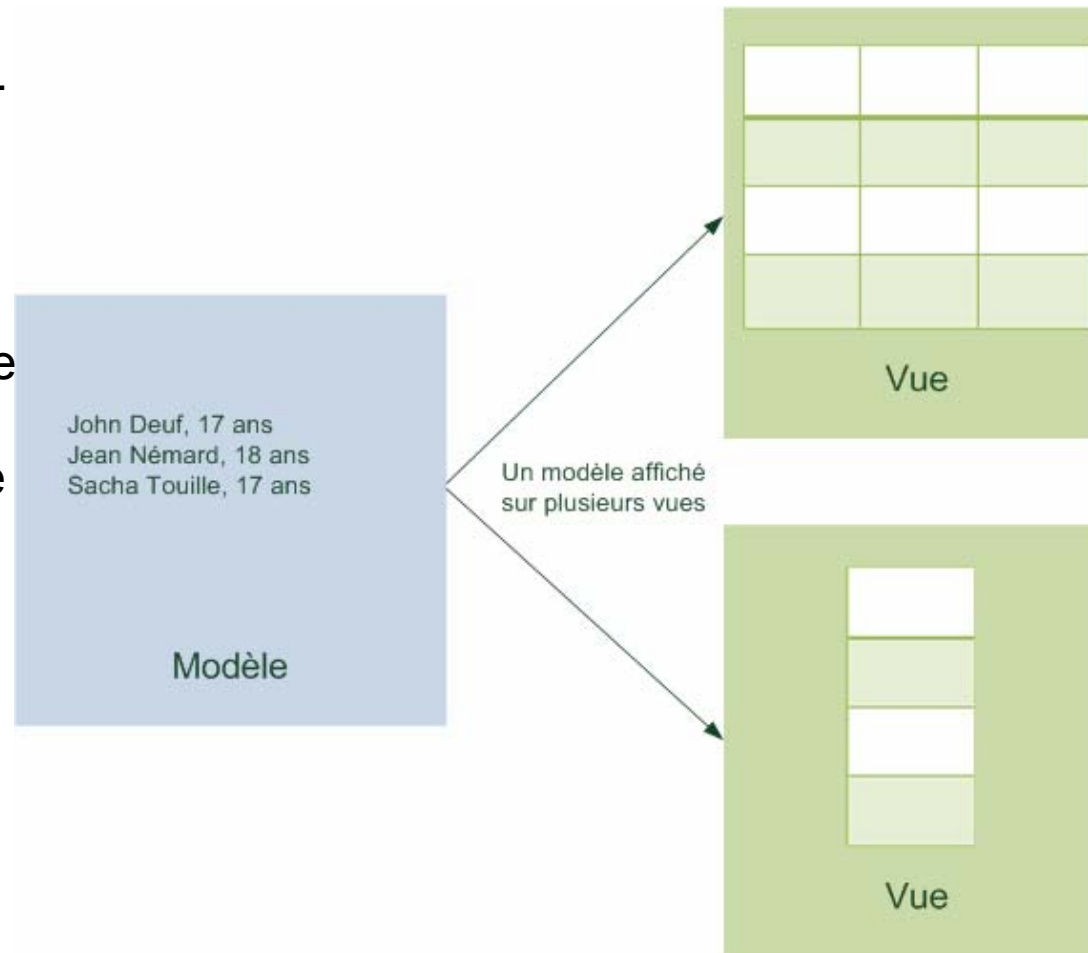


## 4.6) Un modèle pour plusieurs vues

Imaginons le cas inverse. On a un modèle, mais plusieurs vues. Cette fois, rien ne nous empêche d'appliquer ce modèle à 2 vues en même temps !

On peut ainsi visualiser le même modèle de 2 façons différentes (ici sous forme de tableau ou de liste dans mon schéma).

Comme le même modèle est associé à 2 vues différentes, si le modèle change alors les 2 vues changent en même temps



## 4.7) Utilisation d'un modèle simple

- Soit un modèle qui représente le disque. On va l'appliquer à une vue. quelle vue utiliser ? Une liste, un arbre, un tableau
- Toutes les vues peuvent afficher n'importe quel modèle. C'est toujours compatible.
- Pour un `QDirModel`, la vue la plus adaptée est sans aucun doute l'arbre (`QTreeView`).

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale() {
    QVBoxLayout *layout = new QVBoxLayout;

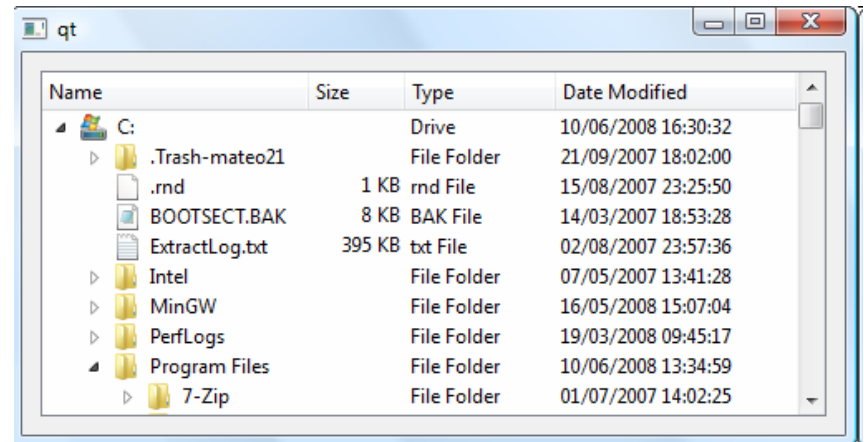
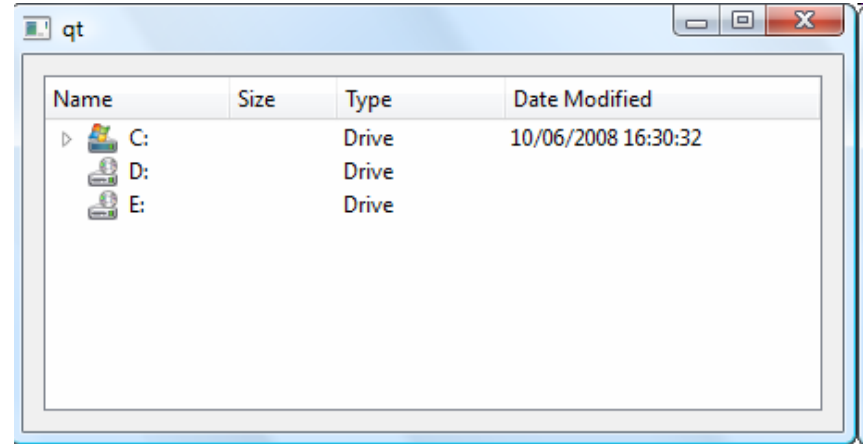
    QDirModel *modele = new QDirModel;
    QTreeView *vue = new QTreeView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

# 4.8) Le modèle appliqué à un QTreeView

- Le résultat
- Une vue en forme d'arbre affiche le modèle du disque.
- Chaque élément peut avoir des sous-éléments dans un QTreeView:



# 4.9) Le modèle appliqué à un QListView

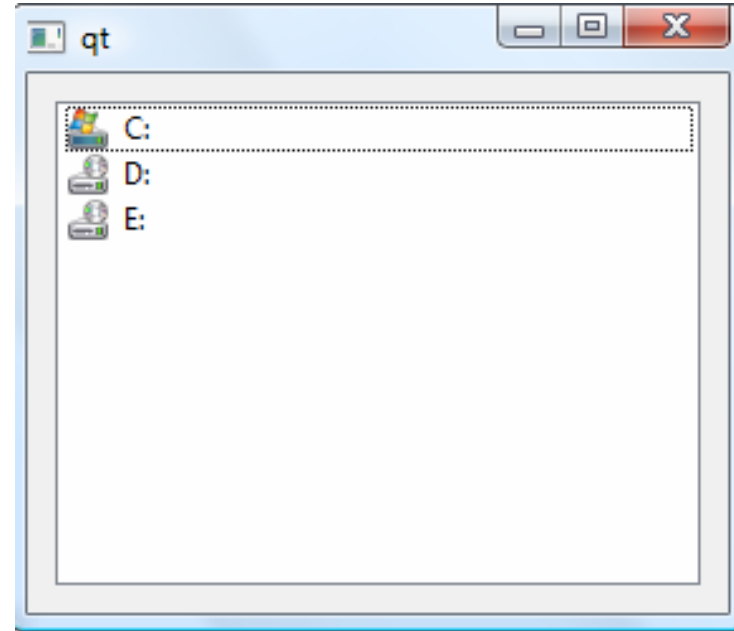
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QDirModel *modele = new QDirModel;
    QListView *vue = new QListView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```



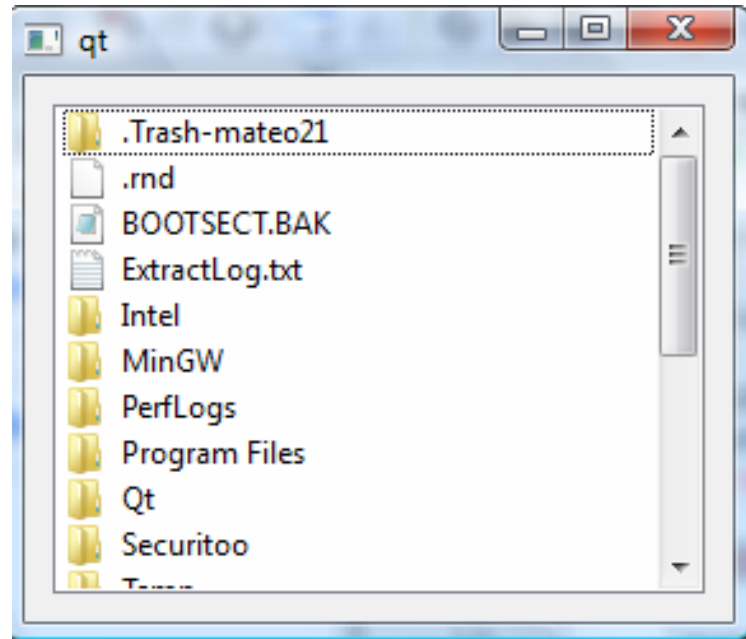
Ce type de vue ne peut afficher qu'un seul niveau d'éléments à la fois. on ne peut pas afficher leur contenu

Un même modèle marche sur plusieurs vues différentes, mais que certaines vues sont plus adaptées que d'autres

# 4.9) Le modèle appliqué à un QListView

modifier la racine utilisée

```
vue->setRootIndex (modele->index ("C:")) ;
```





# 4.10) Le modèle appliqué à un QTableView

Un tableau ne peut pas afficher plusieurs niveaux d'éléments (seul l'arbre QTreeView peut le faire). Par contre, il peut afficher plusieurs colonnes :

```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale()  
{
```

```
    QVBoxLayout *layout = new QVBoxLayout;
```

```
    QDirModel *modele = new QDirModel;
```

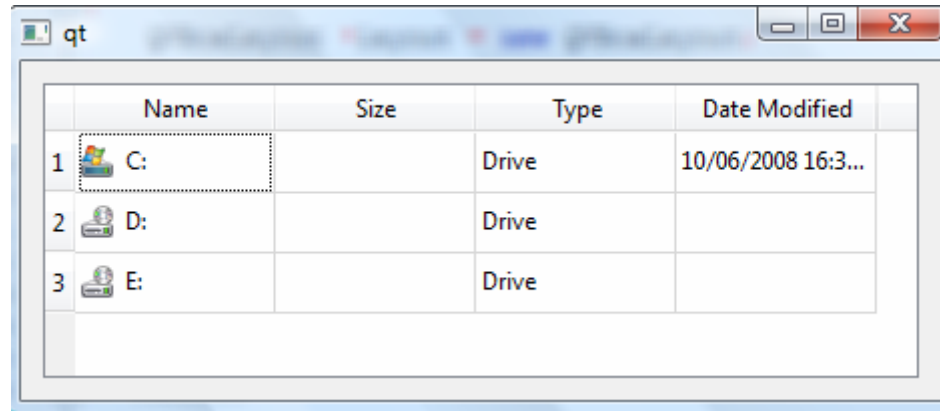
```
    QTableView *vue = new QTableView;
```

```
    vue->setModel(modele);
```

```
    layout->addWidget(vue);
```

```
    setLayout(layout);
```

```
}
```



`setRootIndex()` pour modifier la racine des éléments affichés par la vue.

# 4.11) Utilisation de modèles personnalisables

- Le modèle `QDirModel` est très simple à utiliser.
  - Rien à paramétrer, rien à configurer, il analyse automatiquement le disque dur pour construire son contenu.
- `QStringListModel` : une liste simple d'éléments de type texte, à un seul niveau.
- `QStandardItemModel` : une liste plus complexe à plusieurs niveaux et plusieurs colonnes, qui peut convenir dans la plupart des cas.

## 4.12) QStringListModel : une liste de chaînes de caractères QStringList

- Ce modèle, très simple permet de gérer une liste de chaînes de caractères. Par exemple, si l'utilisateur doit choisir son pays parmi une liste :
  - France
  - Espagne
  - Italie
  - Portugal
  - Suisse
- Pour construire ce modèle, il faut procéder en 2 temps :
  - Construire un objet de type QStringList, qui contiendra la liste des chaînes.
  - Créer un objet de type QStringListModel et envoyer à son constructeur le QStringList.
- QStringList surcharge l'opérateur "<<" pour vous permettre d'ajouter des éléments à l'intérieur simplement.
- Si, au cours de l'exécution du programme, un pays est ajouté, supprimé ou modifié, la vue (la liste) affichera automatiquement les modifications.

```
listePays.append("Belgique");
```

## 4.12) QStringListModel : une liste de chaînes de caractères QString

```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale()
```

```
{  
    QVBoxLayout *layout = new QVBoxLayout;
```

```
    QStringList listePays;
```

```
    listePays <<"France" <<"Espagne" <<"Italie" <<"Portugal" <<"Suisse";
```

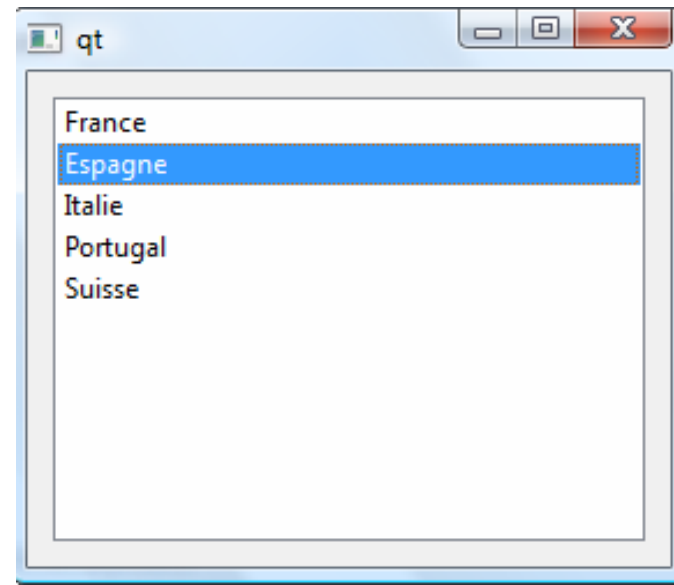
```
    QStringListModel *modele = new QStringListModel(listePays);
```

```
    QListView *vue = new QListView;
```

```
    vue->setModel(modele);
```

```
    layout->addWidget(vue);
```

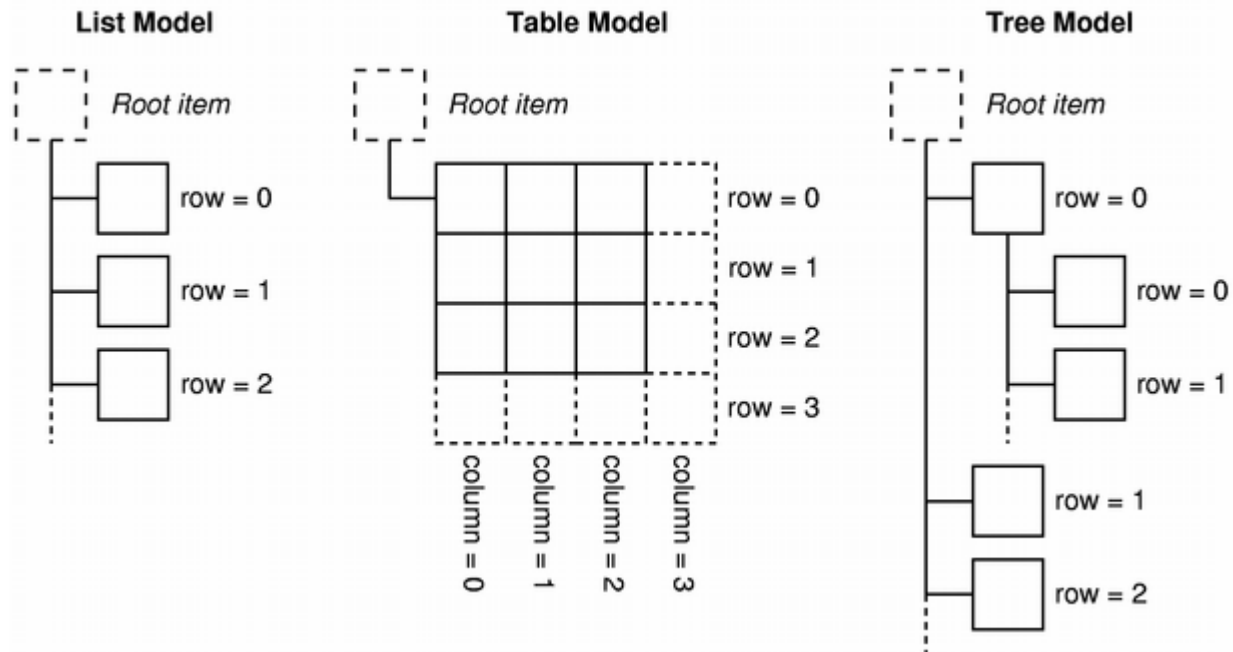
```
    setLayout(layout);  
}
```



# 4.13) QStandardItemModel : une liste à plusieurs niveaux et plusieurs colonnes

Il permet de créer tous les types de modèles possibles et imaginables.

Pour visualiser les différents types de modèles que l'on peut concevoir avec un `QStandardItemModel`,



## 4.13) QStandardItemModel : une liste à plusieurs niveaux et plusieurs colonnes

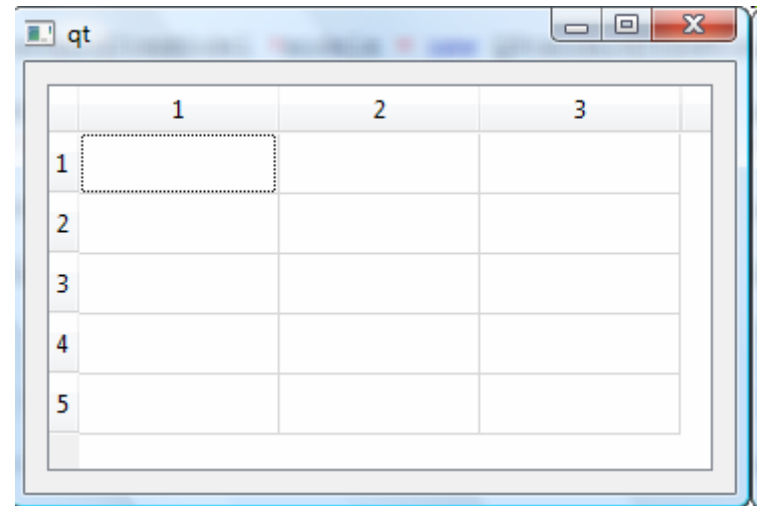
- **List Model** : c'est un modèle avec une seule colonne et pas de sous-éléments. C'est le modèle utilisé par `QStringList`, mais `QStandardItemModel` peut le faire.
  - Ce type de modèle est en général adapté à un `QListView`.
- **Table Model** : les éléments sont organisés avec plusieurs lignes et colonnes.
  - Ce type de modèle est en général adapté à un `QTableView`.
- **Tree Model** : les éléments ont des sous-éléments, ils sont organisés en plusieurs niveaux, rien n'interdit de mettre plusieurs colonnes dans un modèle en arbre!
  - Ce type de modèle est en général adapté à un `QTreeView`.

```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale() {  
    QVBoxLayout *layout = new QVBoxLayout;  
  
    QStandardItemModel *modele = new QStandardItemModel(5, 3);  
    QTableView *vue = new QTableView;  
    vue->setModel(modele);  
  
    layout->addWidget(vue);  
    setLayout(layout);  
}
```

# 4.14) Gérer plusieurs lignes et colonnes

- Pour construire un `QStandardItemModel`, on doit indiquer en paramètres le nombre de lignes et de colonnes qu'il doit gérer.
  - Des lignes et des colonnes supplémentaires peuvent toujours être ajoutées par la suite au besoin.
  - Si on ne demande qu'une seule colonne, cela reviendra à créer un modèle de type "List Model".

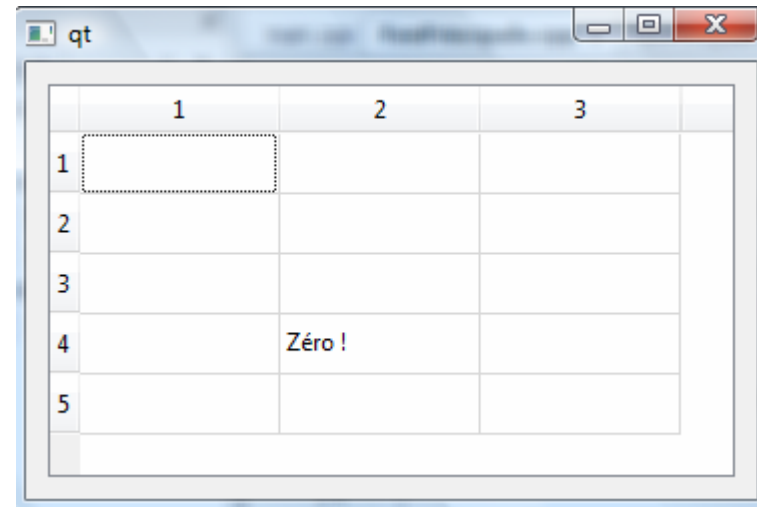


# 4.14) Gérer plusieurs lignes et colonnes

Chaque élément est représenté par un objet de type `QStandardItem`.  
Pour définir un élément, on utilise la méthode `setItem()` du modèle.  
On fournit respectivement le numéro de ligne, de colonne, et un `QStandardItem` à afficher.

```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale() {  
    QVBoxLayout *layout = new QVBoxLayout;  
  
    QStandardItemModel *modele = new QStandardItemModel(5, 3);  
    modele->setItem(3, 1, new QStandardItem("Zéro !"));  
    QTableView *vue = new QTableView;  
    vue->setModel(modele);  
    layout->addWidget(vue);  
  
    setLayout(layout);  
}
```

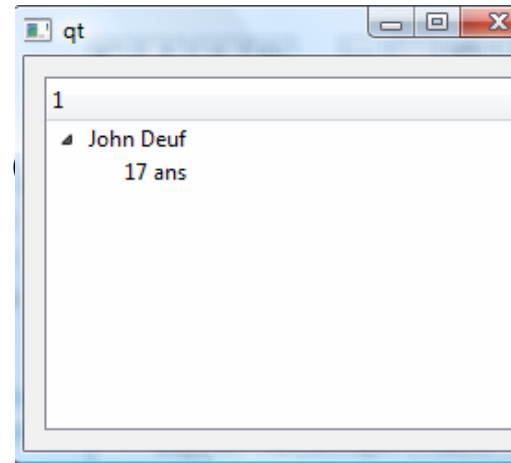


Attention : la numérotation commence à 0.



# 4.15) Ajouter des éléments enfants

- Pour pouvoir voir les éléments enfants, il faut changer de vue et passer par un `QTreeView`.
  - Créer un élément (par exemple "item"), de type `QStandardItem`.
  - Ajouter cet élément au modèle avec `appendRow()`
  - Ajouter un sous-élément à "item" avec `appendRow()`



```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale()
```

```
{
```

```
    QVBoxLayout *layout = new QVBoxLayout;
```

```
    QStandardItemModel *modele = new QStandardItemModel;
```

```
    QStandardItem *item = new QStandardItem("John Deuf");
```

```
    modele->appendRow(item);
```

```
    item->appendRow(new QStandardItem("17 ans"));
```

```
    QTreeView *vue = new QTreeView;
```

```
    vue->setModel(modele);
```

```
    layout->addWidget(vue);
```

```
    setLayout(layout);
```

```
}
```

# 4.16) Gestion des sélections

- Comment récupérer le ou les éléments sélectionnés dans la vue, pour savoir quel est le choix de l'utilisateur.
  - L'architecture modèle/vue de Qt est extrêmement flexible, il y a plusieurs étapes à suivre dans un ordre précis.
- sélections d'un `QListView`.
  - rajouter un bouton "Afficher la sélection" à la fenêtre.
  - Lorsqu'on cliquera sur le bouton, il ouvrira une boîte de dialogue (`QMessageBox`) qui affichera le nom de l'élément sélectionné.



# 4.17) Une sélection unique

- Il faut créer une connexion entre un signal et un slot pour que le clic sur le bouton fonctionne.

- la macro `Q_OBJECT`, pour pouvoir y accéder dans le slot,
- ajouter le slot `clicSelection()` qui sera appelé après un clic sur le bouton.

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QWidget
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    QListView *vue;
    QStringListModel *modele;
    QPushButton *bouton;

private slots:
    void clicSelection();
};

#endif
```

## 4.17) Une sélection unique

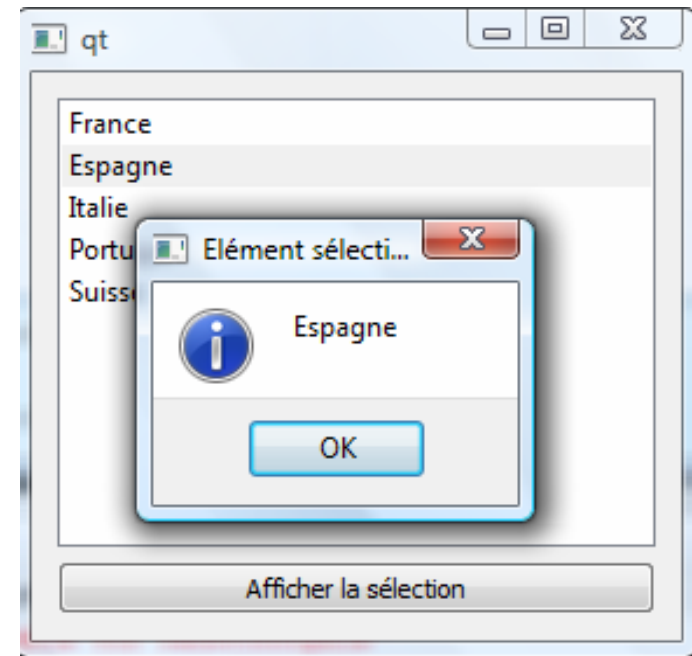
```
#include "FenPrincipale.h"
```

```
FenPrincipale::FenPrincipale() {  
    QVBoxLayout *layout = new QVBoxLayout;  
    QStringList listePays;  
    listePays <<"France" <<"Espagne" <<"Italie" <<"Portugal" <<"Suisse"  
    modele = new QStringListModel(listePays);  
    vue = new QListView ;  
    vue->setModel(modele);  
    bouton = new QPushButton("Afficher la sélection");  
    layout->addWidget(vue);  
    layout->addWidget(bouton);  
    setLayout(layout);  
    connect(bouton, SIGNAL(clicked()), this, SLOT(clicSelection()));  
}
```

```
void FenPrincipale::clicSelection() {  
    QItemSelectionModel *selection = vue->selectionModel();  
    QModelIndex indexElementSelectionne = selection->currentIndex();  
    QVariant elementSelectionne = modele->data(indexElementSelectionne,  
        Qt::DisplayRole);  
    QMessageBox::information(this, "Elément sélectionné",  
        elementSelectionne.toString());  
}
```

# 4.17) Une sélection unique

- Récupérer un objet `QItemSelectionModel` qui contient des informations sur ce qui est sélectionné sur la vue.
  - C'est la vue qui nous donne un pointeur vers cet objet grâce à `vue->selectionModel()`.
- Appeler la méthode `currentIndex()` de l'objet qui contient des informations sur la sélection.
  - Cela renvoie un index, c'est-à-dire en gros le numéro de l'élément sélectionné sur la vue.
- A partir du numéro de l'élément sélectionné, retrouver son texte.
  - Appeler la méthode `data()` du modèle, et lui donner l'index récupéré (c'est-à-dire le numéro de l'élément sélectionné).
  - Le résultat est dans un `QVariant`, qui est une classe qui peut aussi bien stocker des `int` que des chaînes de caractères.
- Afficher l'élément récupéré.  
Pour extraire la chaîne du `QVariant`, appeler `toString()`.



## 4.18) Une sélection multiple

- Par défaut, on ne peut sélectionner qu'un seul élément à la fois sur une liste.  
Pour changer ce comportement et autoriser la sélection multiple:

```
vue->setSelectionMode(QAbstractItemView::ExtendedSelection);
```

- D'autres modes de sélection sont disponibles:  
`QAbstractItemView`
  - Avec ce mode, il est possible sélectionner n'importe quels éléments.
    - utiliser la touche Shift du clavier pour faire une sélection continue, ou Ctrl pour une sélection discontinue (avec des trous).

# 4.18) Une sélection multiple

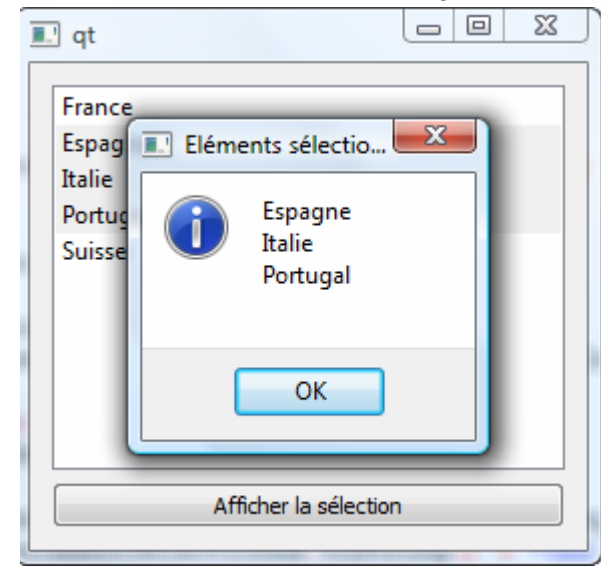
```
void FenPrincipale::clicSelection() {  
    QItemSelectionModel *selection = vue->selectionModel();  
    QModelIndexList listeSelections = selection->selectedIndexes();  
    QString elementsSelectionnes;  
  
    for (int i = 0 ; i < listeSelections.size() ; i++)  
    {  
        QVariant elementSelectionne = modele->data(listeSelections[i],  
                                                    Qt::DisplayRole);  
        elementsSelectionnes += elementSelectionne.toString() + "<br />"  
    }  
    QMessageBox::information(this, "Eléments sélectionnés",  
                             elementsSelectionnes);  
}
```



code du slot

# 4.18) Une sélection multiple

- 1- Récupérer l'objet qui contient des informations sur les éléments sélectionnés.
- 2- Ensuite, au lieu d'appeler `currentIndex()`, demander `selectedIndexes()` parce qu'il peut y en avoir plusieurs.
- 3- Créer un `QString` vide dans lequel sera stocker la liste des pays pour l'afficher dans la boîte de dialogue.
- 4- Boucle, l'objet `listeSelections` récupéré est un tableau (en fait c'est un objet de type `QList`. On parcourt ce tableau ligne par ligne, pour récupérer à chaque fois le texte correspondant.
- 4.1- Stocker ce texte à la suite du `QString`, qui se remplit au fur et à mesure.
- 4- Une fois la boucle terminée, afficher le `QString` qui contient la liste des pays sélectionnés.





## 5) Les différents composants graphiques

- Les composants de base de l'IHM (`QMainWindow`, `QFrame`, `QLabel` ...).
- La gestion du positionnement des composants.
- Les boîtes de dialogue (`QDialog`).
- Les menus (`QMenu`).
- Modèles prédéfinis et personnalisés.
- Les outils de conception visuelle de QT (`QT Designer` ...).

# 5.1) Modifier les propriétés d'un widget

- Comme tous les éléments d'une fenêtre, le bouton est un widget.
  - Avec Qt, on crée un bouton à l'aide de la classe `QPushButton`.
- Une classe est constituée de 2 éléments :
  - Des attributs : ce sont les "variables" internes de la classe.
  - Des méthodes : ce sont les "fonctions" internes de la classe.
- La règle d'encapsulation dit que les utilisateurs de la classe ne doivent pas pouvoir modifier les attributs : ceux-ci doivent donc tous être privés.
- Or, les développeurs sont des utilisateurs des classes de Qt.
  - ils n'ont pas accès aux attributs puisque ceux-ci sont privés
- Le créateur d'une classe doit rendre ses attributs privés, mais du coup proposer des méthodes accesseurs,
  - des méthodes permettant de lire et de modifier les attributs de manière sécurisée (get et set).

## 5.2) Les accesseurs avec Qt

- Les développeurs de Trolltech ont codé proprement en respectant ces règles. Pour chaque propriété d'un widget, on a :
  - Un attribut : il est privé on ne peut pas le lire ni le modifier directement.
  - Exemple : `text`
  - Un accesseur pour le lire :  
cet accesseur est une méthode constante qui porte le même nom que l'attribut Exemple : `text()`
  - Un accesseur pour le modifier : c'est une méthode qui se présente sous la forme `setAttribut()`. Elle modifie la valeur de l'attribut.
  - Exemple : `setText()`
- Qt peut vérifier que la valeur donnée est valide.
- Cela permet d'éviter par exemple de donner à une barre de progression la valeur "150%", alors que la valeur d'une barre de progression doit être comprise entre 0 et 100%.



# 5.3) Quelques exemples de propriétés des boutons

- `text` : le texte
- Cette propriété est probablement la plus importante : elle permet de modifier le texte présent sur le bouton.
  - En général, le texte du bouton est défini au moment de sa création.
- Pour chaque attribut, la documentation de Qt donne à quoi il sert et quels sont ses accesseurs. Elle
  - indique de quel type est l'attribut. Ici, `text` est de type `QString`, comme tous les attributs qui stockent du texte avec Qt.
  - explique en quelques mots à quoi sert cet attribut.
  - indique les accesseurs qui permettent de lire et de modifier l'attribut. Dans le cas présent, il s'agit de :
- `QString text () const` : c'est l'accesseur qui permet de lire l'attribut. Il retourne un `QString`, ce qui est logique puisque l'attribut est de type `QString`. Le mot-clé "`const`" qui indique que c'est une méthode constante qui ne modifie aucun attribut.
- `void setText ( const QString & text )` : c'est l'accesseur qui permet de modifier l'attribut. Il prend un paramètre : le texte à mettre sur le bouton.

Cela suit toujours le même schéma :

`attribut ()` : permet de lire l'attribut.

`setAttribut ()` : permet de modifier l'attribut.

# 5.3) Quelques exemples de propriétés des boutons

```
#include <QApplication>
#include <QPushButton>
```

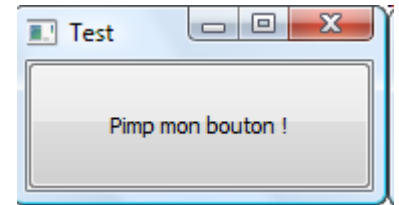
text : le texte

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Hello world !");
    bouton.setText("Pimp mon bouton !");

    bouton.show();

    return app.exec();
}
```



## 5.3) Quelques exemples de propriétés des boutons

toolTip : l'infobulle

Il est courant d'afficher une petite aide qui apparaît lorsqu'on pointe sur un élément avec la souris.

L'infobulle peut afficher un court texte d'aide.

Elle est défini à l'aide de la propriété `toolTip`.

Pour modifier l'infobulle, la méthode à appeler est donc `setToolTip`.

```
#include <QApplication>
#include <QPushButton>
```

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

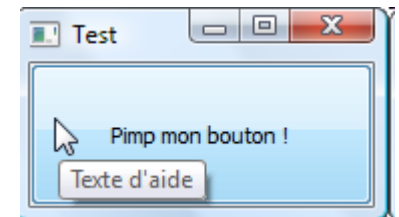
```
    QApplication app(argc, argv);
```

```
    QPushButton bouton("Pimp mon bouton !");
    bouton.setToolTip("Texte d'aide");
```

```
    bouton.show();
```

```
    return app.exec();
```

```
}
```



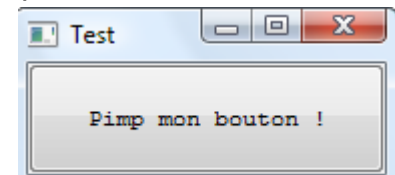
# 5.3) Quelques exemples de propriétés des boutons

- `font` : la police
- La propriété `font` contient 3 informations :
  - Le nom de la police de caractères utilisée (Times New Roman, Arial, Comic Sans MS...)
  - La taille du texte en pixels (12, 16, 18...)
  - Le style du texte (gras, italique...)
- La signature de la méthode `setFont` est :

```
void setFont ( const QFont & )
```
- Cela veut dire que `setFont` attend un objet de type `QFont`. Le constructeur de `QFont` attend 4 paramètres.

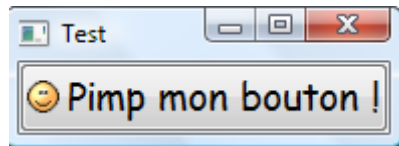
```
QFont(const QString& family, int pointSize = -1, int weight = -1,  
      bool italic = false)
```
- Seul le 1<sup>er</sup> argument est obligatoire : il s'agit du nom de la police à utiliser. Dans l'ordre, les paramètres signifient :
  - `family` : le nom de la police de caractères à utiliser.
  - `pointSize` : la taille des caractères en pixels.
  - `weight` : le niveau d'épaisseur du trait (gras). Cette valeur peut être comprise entre 0 et 99 (du plus fin au plus gras). Il est possible d'utiliser la constante `QFont::Bold` qui correspond à une épaisseur de 75.
  - `italic` : un booléen pour dire si le texte doit être affiché en italique ou non.

```
QFont maPolice("Courier");  
bouton.setFont(maPolice);
```



## 5.3) Quelques exemples de propriétés des boutons

- `icon` : l'icône du bouton
- la méthode `setIcon` attend juste un objet de type `QIcon`.
  - Un `QIcon` peut se construire très facilement en donnant le nom du fichier image à charger.
  - `bouton.setIcon(QIcon("smile.png")) ;`
- Pour utiliser le chemin de votre application  
`QIcon(QCoreApplication::applicationDirPath() + "/smile.png") ;`

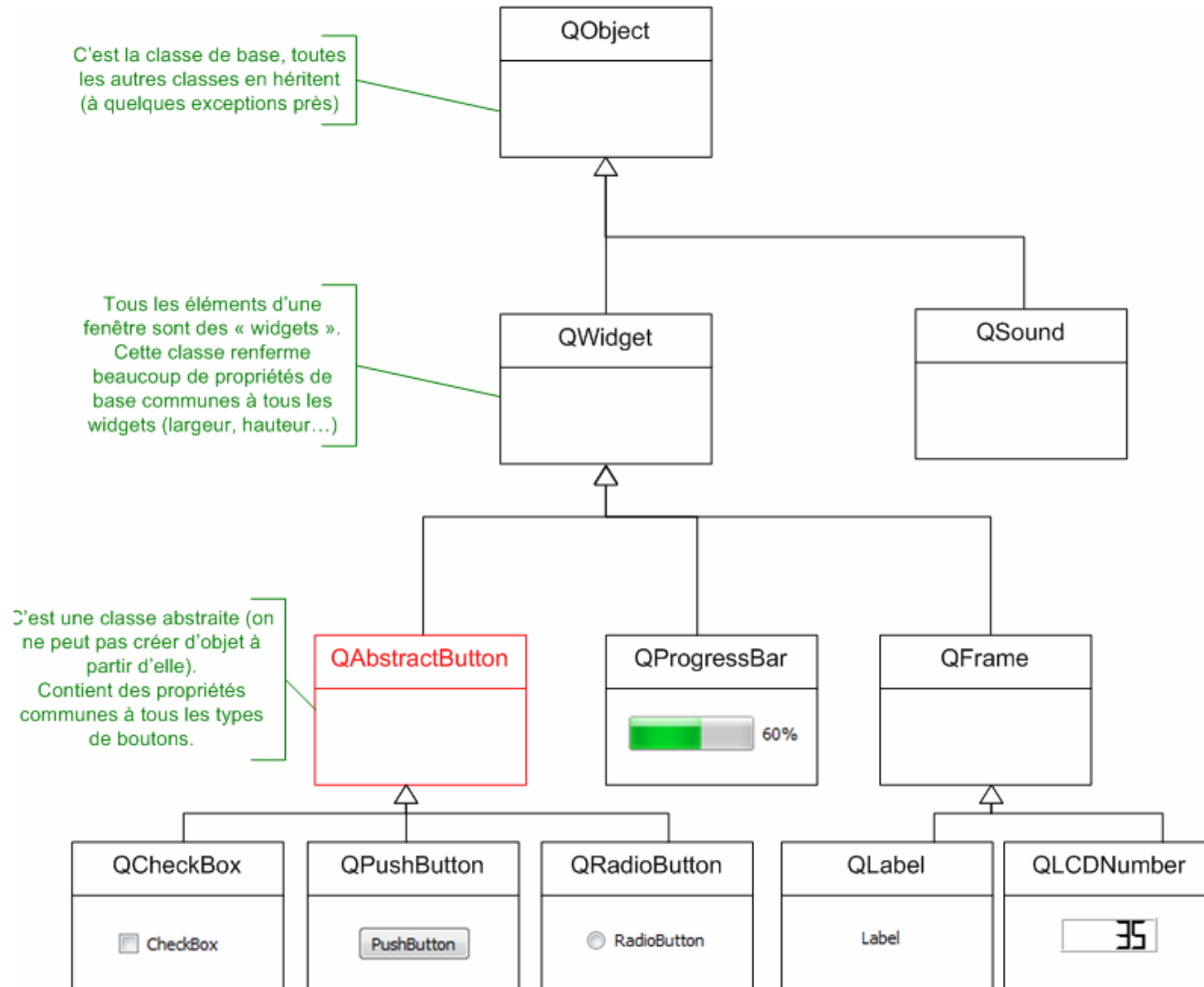




## 5.4) Qt et l'héritage

- Quasiment toutes les classes de Qt font appel à l'héritage.  
    `QAbstractExtensionFactory`  
        `QExtensionFactory`  
    `QAbstractExtensionManager`  
        `QExtensionManager`
- `QAbstractExtensionFactory` et `QAbstractExtensionManager` sont des classes dites "de base". Elles n'ont pas de classes parentes.
- En revanche, `QExtensionFactory` et `QExtensionManager` sont des classes-filles, qui héritent respectivement de `QAbstractExtensionFactory` et `QAbstractExtensionManager`.
- `QObject` est la classe de base de tous les objets sous Qt.

# 5.5) QObject : une classe de base incontournable



# 5.5) QObject : une classe de base incontournable

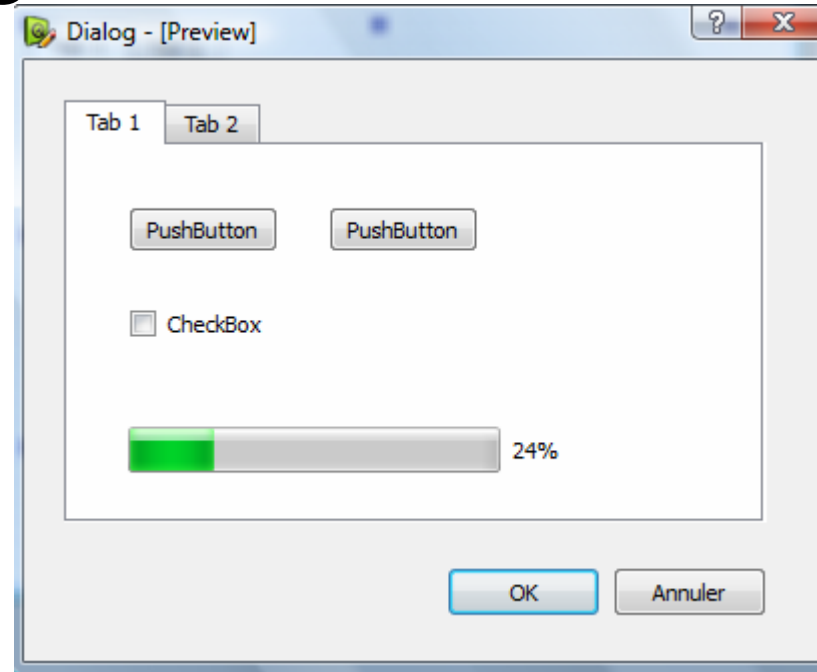
- `QObject` est indispensable pour réaliser le mécanisme des signaux et des slots. Ce mécanisme permet si un bouton est cliqué, alors une autre fenêtre s'ouvre
  - on dit qu'il envoie un signal à un autre objet.
- Certaines classes comme `QSound` (gestion du son) héritent directement de `QObject`.
- Pour la création de GUI, tout est considéré comme un widget (même la fenêtre est un widget).
  - il existe une classe de base `QWidget` pour tous les widgets. Elle contient énormément de propriétés communes à tous les widgets, comme :
    - La largeur
    - La hauteur
    - La position en abscisse (x)
    - La position en ordonnée (y)
    - La police de caractères utilisée (eh oui, la méthode `setFont` est définie dans `QWidget`, et comme `QPushButton` en hérite, il possède lui aussi cette méthode)
    - Le curseur de la souris (`setCursor` est en fait défini dans `QWidget` et non dans `QPushButton`, car il est aussi susceptible de servir sur tous les autres widgets)
    - L'infobulle (`toolTip`)
    - etc.
- Par exemple que vous pouvez retrouver la méthode `setCursor` dans la classe `QProgressBar`.

# 5.6) Les classes abstraites

- Une classe abstraite sert de classe de base pour d'autres sous-classes.
  - `QAbstractButton` définit un certain nombre de propriétés communes à tous les types de boutons (boutons classiques, cases à cocher, cases radio...).
  - Parmi les propriétés communes on trouve :
    - `text` : le texte affiché
    - `icon` : l'icône affichée à côté du texte du bouton
    - `shortcut` : le raccourci clavier pour activer le bouton
    - `down` : indique si le bouton est enfoncé ou non
    - etc.
- Il existe un grand nombre de classes abstraites sous Qt, qui contiennent toutes le mot "Abstract" dans leur nom.
- Défini qu'une fois dans `QAbstractButton`, et on le retrouve ensuite automatiquement dans `QPushButton`, `QCheckBox`, etc.

# 5.7) Un widget peut en contenir un autre

- Un widget peut en contenir un autre.
- Par exemple, une fenêtre (QWidget) peut contenir
  - 3 boutons (QPushButton),
  - une case à cocher (QCheckBox),
  - une barre de progression (QProgressBar),
  - etc.



Les widgets sont donc imbriqués les uns dans les autres de cette manière :

QWidget (la fenêtre)

  QPushButton

  QPushButton

  QTabWidget (le conteneur à onglets)

    QPushButton

    QPushButton

    QCheckBox

    QProgressBar

# 5.8) Créer une fenêtre contenant un bouton

```
#include <QApplication>
#include <QPushButton>
```

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

```
{
    QApplication app(argc, argv);
```

```
    // Création d'un widget qui servira de fenêtre
```

```
    QWidget fenetre;
```

```
    fenetre.setFixedSize(300, 150);
```

```
    // Création du bouton, ayant pour parent la "fenetre"
```

```
    QPushButton bouton("Pimp mon bouton !", &fenetre);
```

```
    // Customisation du bouton
```

```
    bouton.setFont(QFont("Comic Sans MS", 14));
```

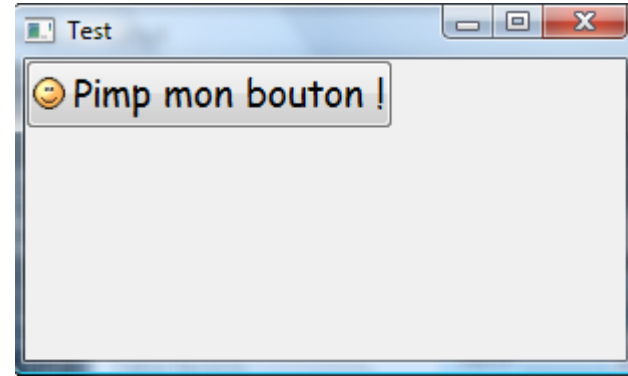
```
    bouton.setCursor(Qt::PointingHandCursor);
```

```
    bouton.setIcon(QIcon("smile.png"));
```

```
    // Affichage de la fenêtre
```

```
    fenetre.show();
```

```
    return app.exec();
}
```



# 5.9) Tout widget peut en contenir d'autres

```
#include <QApplication>
#include <QPushButton>
```

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

```
{
```

```
    QApplication app(argc, argv);
```

```
    QWidget fenetre;
```

```
    fenetre.setFixedSize(300, 150);
```

```
    QPushButton bouton("Pimp mon bouton !", &fenetre);
```

```
    bouton.setFont(QFont("Comic Sans MS", 14));
```

```
    bouton.setCursor(Qt::PointingHandCursor);
```

```
    bouton.setIcon(QIcon("smile.png"));
```

```
    bouton.setGeometry(60, 50, 180, 70);
```

```
    // Création d'un autre bouton ayant pour parent le premier bouton
```

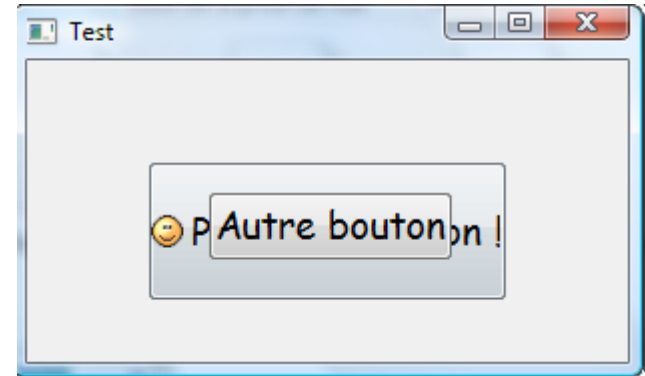
```
    QPushButton autreBouton("Autre bouton", &bouton);
```

```
    autreBouton.move(30, 15);
```

```
    fenetre.show();
```

```
    return app.exec();
```

```
}
```



## 5.10) Des includes "oubliés"

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QFont>
#include <QIcon>
```

Pour être sûr d'inclure une bonne fois pour toutes les classes du module "Qt GUI", il vous suffit de faire :

```
#include <QtGui>
```

Le header "QtGui" inclut à son tour toutes les classes du module GUI, donc QWidget, QPushButton, QFont, etc.

Attention toutefois, la compilation sera un peu ralentie du coup.



# 5.11) Hériter un widget

La personnalisation des widgets se fait en "inventant" un nouveau type de widget  
Qui dit nouvelle classe dit 2 nouveaux fichiers :

`MaFenetre.h` : contiendra la définition de la classe

`MaFenetre.cpp` : contiendra l'implémentation des méthodes

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

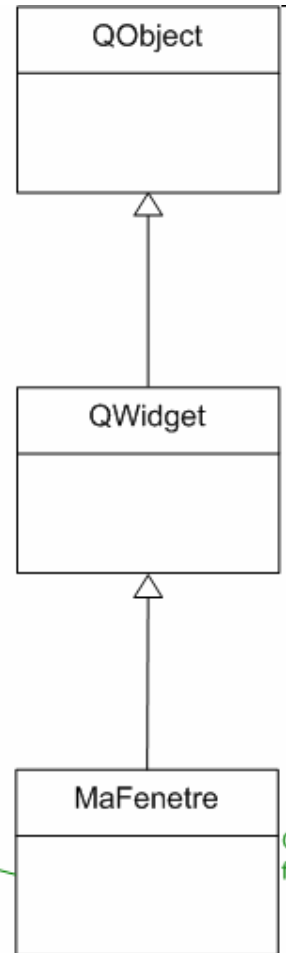
#include <QApplication>
#include <QWidget>
#include <QPushButton>

class MaFenetre : public QWidget
{
    public:
        MaFenetre();

    private:
        QPushButton *m_bouton;
};

#endif
```

MaFenetre hérite de  
QWidget. Ce sera une  
fenêtre personnalisée.



# 5.11) Hériter un widget

```
#include "MaFenetre.h"
```

```
MaFenetre::MaFenetre() : QWidget()
```

```
{
```

```
    setFixedSize(300, 150);
```

```
    // Construction du bouton
```

```
    m_bouton = new QPushButton("Pimp mon bouton !", this);
```

```
    m_bouton->setFont(QFont("Comic Sans MS", 14));
```

```
    m_bouton->setCursor(Qt::PointingHandCursor);
```

```
    m_bouton->setIcon(QIcon("smile.png"));
```

```
    m_bouton->move(60, 50);
```

```
}
```

```
#include <QApplication>
```

```
#include "MaFenetre.h"
```

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

```
{
```

```
    QApplication app(argc, argv);
```

```
    MaFenetre fenetre;
```

```
    fenetre.show();
```

```
    return app.exec();
```

```
}
```

# 5.12) La destruction automatique des widgets enfants

- Tout objet créé dynamiquement avec un `new` implique forcément un `delete` quelque part.
- Normalement, on devrait écrire le destructeur de `MaFenetre`, qui contiendrait ceci :

```
MaFenetre::~~MaFenetre()  
{  
    delete m_bouton;  
}
```

- Toutefois, Qt supprimera automatiquement le bouton lors de la destruction de la fenêtre (à la fin du `main`).
- En effet, quand on supprime un widget parent (ici la fenêtre), Qt supprime automatiquement tous les widgets qui se trouvent à l'intérieur

## 5.13) Afficher un message

- Il faut créer un bouton sur la fenêtre de type `MaFenetre` qui appellera un slot personnalisé.
  - Ce slot ouvrira une boîte de dialogue.
  - Les boîtes de dialogue "afficher un message" sont contrôlées par la classe `QMessageBox`.

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QMessageBox>

class MaFenetre : public QWidget {
    Q_OBJECT

public:
    MaFenetre();

public slots:
    void ouvrirDialogue();

private:
    QPushButton *m_boutonDialogue;
};

#endif
```

## 5.13) Afficher un message

```
#include "MaFenetre.h"
```

```
MaFenetre::MaFenetre() : QWidget() {  
    setFixedSize(230, 120);  
    m_boutonDialogue = new QPushButton("Ouvrir la boîte de dialogue",  
                                        this);  
  
    m_boutonDialogue->move(40, 50);  
    QObject::connect(m_boutonDialogue, SIGNAL(clicked()), this,  
                    SLOT(ouvrirDialogue()));  
}  
  
void MaFenetre::ouvrirDialogue() {  
    // insérer le code d'ouverture des boîtes de dialogue  
}
```

créer un bouton dans la boîte de dialogue  
qui appelle le slot personnalisé  
ouvrirDialogue().  
dans ce slot que une boîte de dialogue  
sera chargée.

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

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

# 5.14) Ouvrir une boîte de dialogue avec une méthode statique

- La classe `QMessageBox` permet de créer des objets de type `QMessageBox`
  - ses méthodes statiques  
`StandardButton information(QWidget* parent, const QString & title, const QString& text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton );`
    - Ces 3 premiers paramètres sont
      - `parent` : un pointeur vers la fenêtre parente (qui doit être de type `QWidget` ou hériter de `QWidget`).
      - `title` : le titre de la boîte de dialogue (affiché en haut de la fenêtre).
      - `text` : le texte affiché au sein de la boîte de dialogue.
  - La méthode statique `information()` permet d'ouvrir une boîte de dialogue constituée d'une icône "information".

```
void MaFenetre::ouvrirDialogue() {  
    QMessageBox::information(this, "Titre de la fenêtre",  
        "Bonjour et bienvenue à tous les Nouveaux !");  
}
```

# 5.14) Ouvrir une boîte de dialogue avec une méthode statique

- `QMessageBox::warning`
  - la boîte de dialogue warning le met en garde contre quelque chose.
  - l'icône change

```
QMessageBox::warning(this, "Titre de la fenêtre",  
"Attention, vous êtes peut-être un Nouveaux !"),
```

- `QMessageBox::critical`
  - Quand une erreur s'est produite, il ne vous reste plus qu'à utiliser la méthode statique `critical()` :

```
QMessageBox::critical(this, "Titre de la fenêtre",  
"Vous n'êtes pas un Nouveaux , sortez d'ici ou j'appelle la police !")
```

## 5.14) Ouvrir une boîte de dialogue avec une méthode statique

- `QMessageBox::question`
- une question à poser à l'utilisateur,

```
QMessageBox::question(this, "Titre de la fenêtre", "Dites voir, je  
me posais la question comme ça, êtes-vous vraiment un Nouveaux?");
```



# 5.15) Personnaliser les boutons de la boîte de dialogue

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "MaFenetre.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);
    QTranslator translator;
    translator.load(QString("qt_") + locale,
        QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    MaFenetre fenetre;
    fenetre.show();
    return app.exec();
}
```

# 5.16) Récupérer la valeur de retour de la boîte de dialogue

```
void MaFenetre::ouvrirDialogue()  
{  
    int reponse = QMessageBox::question(this, "Interrogatoire",  
        "êtes-vous d'accord ou pas d'accord ?",  
        QMessageBox::Yes | QMessageBox::No);  
  
    if (reponse == QMessageBox::Yes) {  
        QMessageBox::information(this, "Interrogatoire",  
            "Alors bienvenue!");  
    }  
    else if (reponse == QMessageBox::No) {  
        QMessageBox::critical(this, "Interrogatoire",  
            "Alors pas bienvenue!");  
    }  
}
```

## 5.17) Saisir une information

- Les boîtes de dialogue "saisir une information" peuvent être de 4 types:
- Elles sont gérées par la classe `QInputDialog`
  - Saisir un texte
  - Saisir un entier
  - Saisir un nombre décimal (double)
  - Choisir un élément parmi une liste
- Chacune de ces fonctionnalités est assurée par une méthode statique différente.

# 5.17) Saisir une information

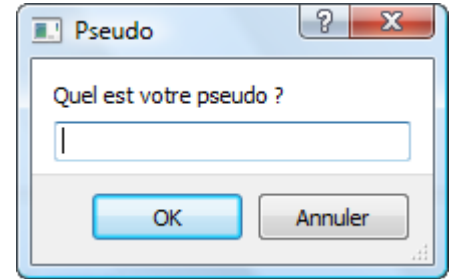
- La méthode statique `getText()` ouvre une boîte de dialogue qui permet à l'utilisateur de saisir un texte.

```
QString QInputDialog::getText ( QWidget * parent,  
    const QString & title, const QString & label,  
    QLineEdit::EchoMode mode = QLineEdit::Normal,  
    const QString & text = QString(), bool * ok = 0,  
    Qt::WindowFlags f = 0 );
```

- `parent` : pointeur vers la fenêtre parente. Peut être mis à NULL pour ne pas indiquer de fenêtre parente.
- `title` : titre de la fenêtre affiché en haut.
- `label` : texte affiché dans la fenêtre.
- `mode` : mode d'édition du texte. Par défaut, les lettres s'affichent normalement (`QLineEdit::Normal`).
- `text` : le texte par défaut dans la zone de saisie.
- `ok` : un pointeur vers un booléen pour que Qt puisse vous dire si l'utilisateur a cliqué sur OK ou sur Annuler.
- `f` = quelques flags (options) permettant d'indiquer si la fenêtre est modale (bloquante) ou pas.

# 5.17) Saisir une information

```
void MaFenetre::ouvrirDialogue() {  
    QString pseudo = QInputDialog::getText(this, "Pseudo",  
        "Quel est votre pseudo ?");  
}
```



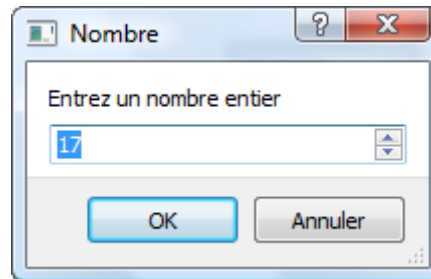
- vérifier si le bouton OK a été actionné, et si c'est le cas on peut alors afficher le pseudo de l'utilisateur dans une QMessageBox.

```
void MaFenetre::ouvrirDialogue() {  
    bool ok = false;  
    QString pseudo = QInputDialog::getText(this, "Pseudo",  
        "Quel est votre pseudo ?", QLineEdit::Normal, QString(), &ok);  
    if (ok && !pseudo.isEmpty()) {  
        QMessageBox::information(this, "Pseudo", "Bonjour " + pseudo +  
            ", ça va ?");  
    } else {  
        QMessageBox::critical(this, "Pseudo",  
            "Vous n'avez pas voulu donner votre nom... snif.");  
    }  
}
```

# 5.18) Saisir un entier

## (QInputDialog::getInteger)

```
int QInputDialog::getInteger(QWidget* parent, const QString & title,  
    const QString& label, int value = 0,  
    int minValue = -2147483647, int maxValue = 2147483647,  
    int step = 1, bool * ok = 0, Qt::WindowFlags f = 0 );  
  
int entier = QInputDialog::getInteger(this, "Nombre", "Entrez un  
    nombre entier");
```



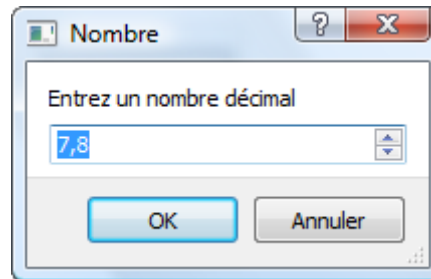
Le nombre saisi est retourné par la méthode dans un entier,

# 5.19) Saisir un nombre décimal (QInputDialog::getDouble)

- un paramètre qui permet d'indiquer le nombre maximal de chiffres après la virgule

```
double QInputDialog::getDouble(QWidget* parent, const QString& title,  
    const QString& label, double value = 0,  
    double minValue = -2147483647, double maxValue = 2147483647,  
    int decimals = 1, bool* ok = 0, Qt::WindowFlags f = 0 );
```

```
double nombreDecimal = QInputDialog::getDouble(this, "Nombre",  
    "Entrez un nombre décimal");
```



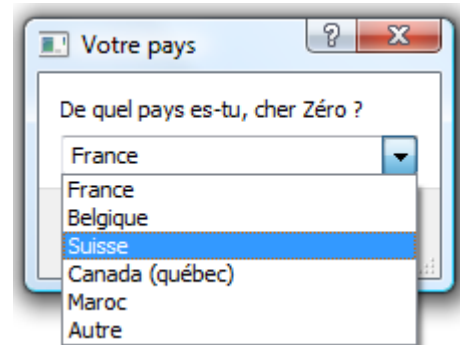
# 5.20) Choix d'un élément parmi une liste (QInputDialog::getItem)

- Si l'utilisateur doit faire son choix dans une liste

```
QString QInputDialog::getItem(QWidget* parent, const QString& title,  
    const QString & label, const QStringList & list, int current = 0,  
    bool editable = true, bool * ok = 0, Qt::WindowFlags f = 0 );
```

- `list` : la liste des choix possibles, envoyée via un objet de type `QStringList` (liste de chaînes) à construire au préalable.
- `current` : le numéro du choix qui doit être sélectionné par défaut.
- `editable` : un booléen qui indique si l'utilisateur a le droit d'entrer sa propre réponse (comme avec `getText`) ou s'il est obligé de faire un choix parmi la liste.

```
void MaFenetre::ouvrirDialogue() {  
    QStringList pays;  
    pays << "France" << "Belgique" << "Suisse" << "Canada (québec)"  
        << "Maroc" << "Autre";  
    QInputDialog::getItem(this, "Votre pays",  
        "De quel pays es-tu, cher Zéro ?", pays);  
}
```

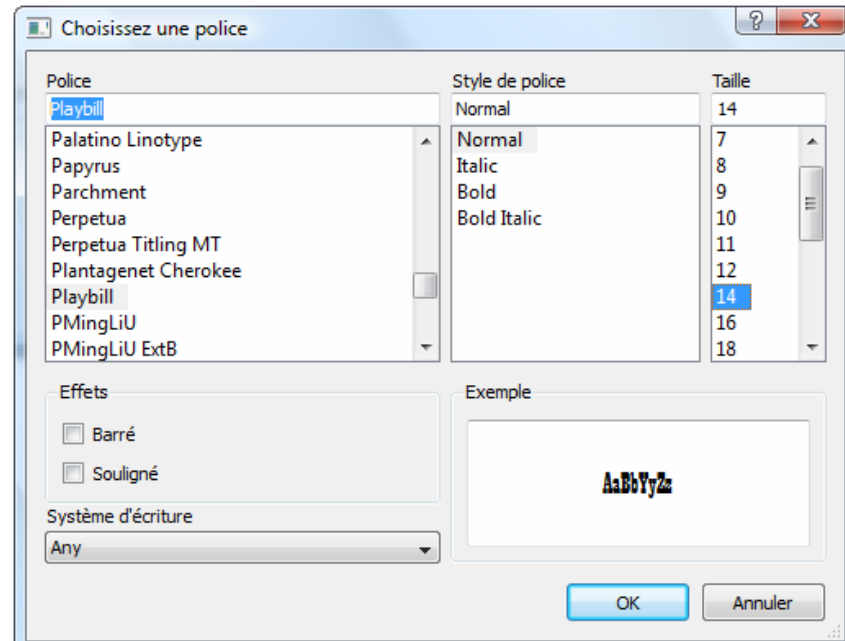




# 5.21) Sélectionner une police

La boîte de dialogue de sélection de police est gérée par la classe QFontDialog.

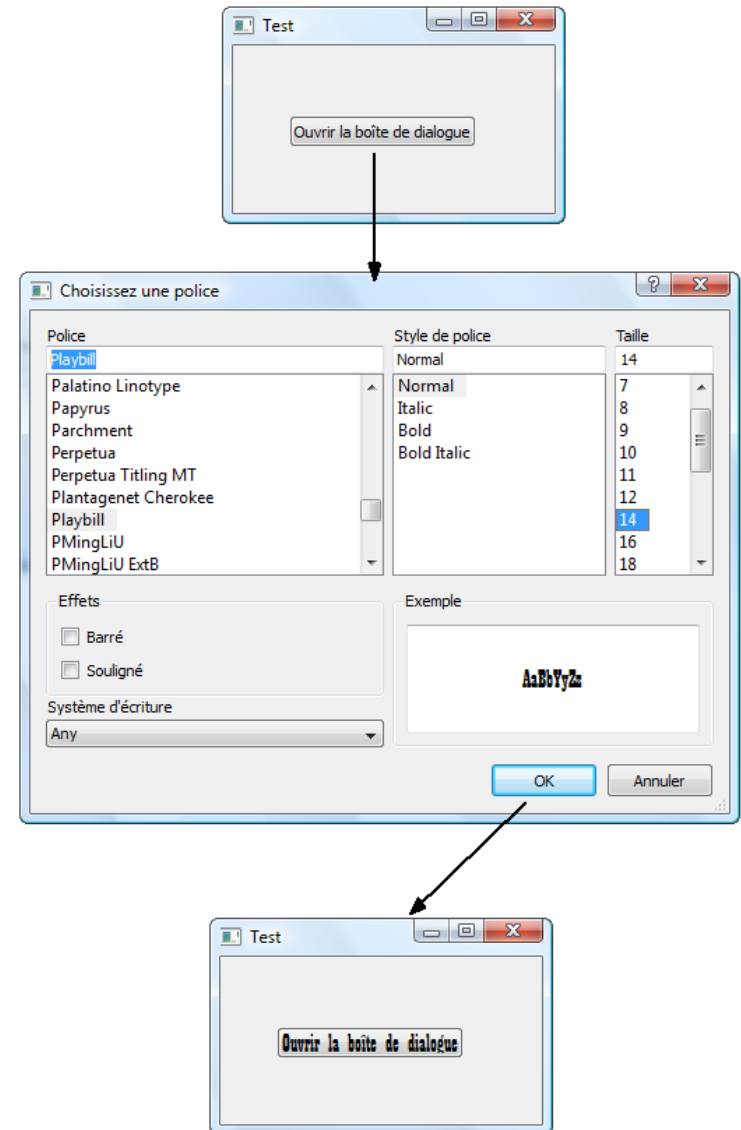
```
QFont getFont(bool* ok, const QFont& initial, QWidget* parent,  
              const QString& caption )  
  
void MaFenetre::ouvrirDialogue() {  
    bool ok = false;  
  
    QFont police = QFontDialog::getFont(&ok, m_boutonDialogue->font(),  
                                         this, "Choisissez une police");  
  
    if (ok) {  
        m_boutonDialogue->setFont(police);  
    }  
}
```



# 5.21) Sélectionner une police

La méthode `getFont` prend comme police par défaut celle qui est utilisée par le bouton `m_boutonDialogue`

`font()` est une méthode accesseur qui renvoie un `QFont`



## 5.22) Sélectionner une couleur

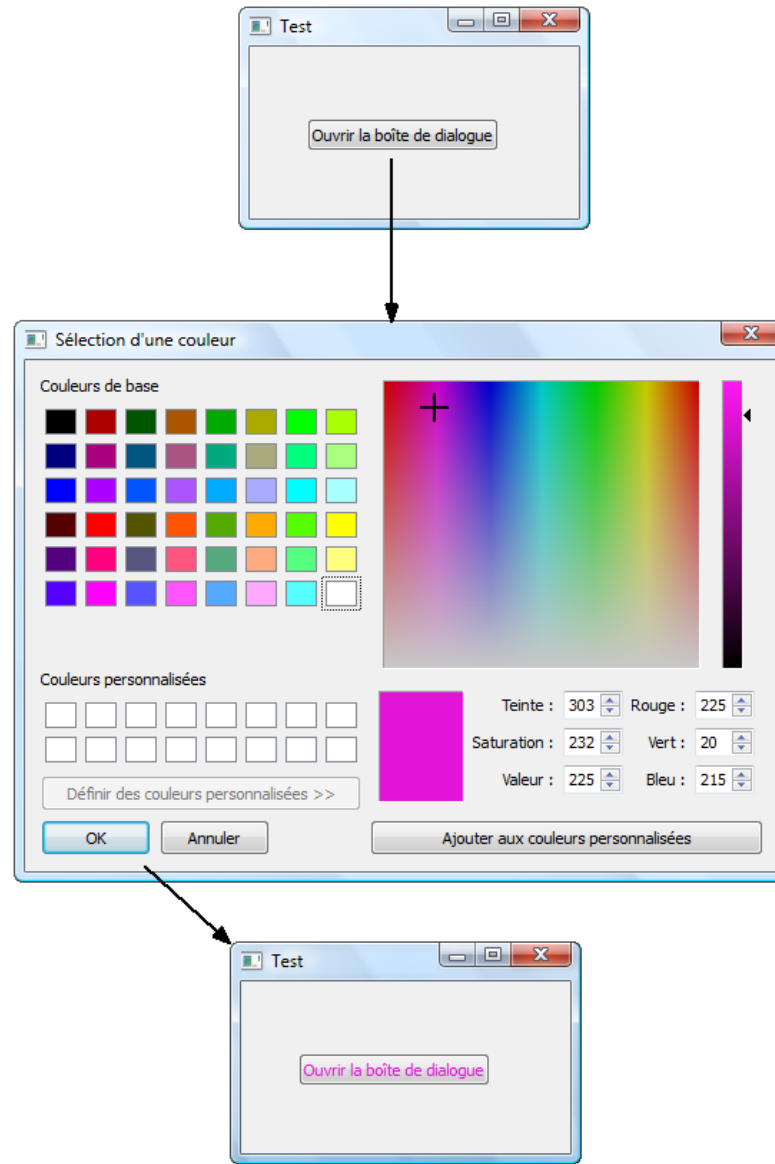
```
QColor QColorDialog::getColor ( const QColor & initial = Qt::white,  
    QWidget * parent = 0 );
```

En l'absence de paramètre, c'est la couleur blanche qui sera sélectionnée

```
void MaFenetre::ouvrirDialogue()  
{  
    QColor couleur = QColorDialog::getColor(Qt::white, this);  
  
    QPalette palette;  
    palette.setColor(QPalette::ButtonText, couleur);  
    m_boutonDialogue->setPalette(palette);  
}
```

il n'existe pas de méthode `setColor` pour les widgets, mais une méthode `setPalette`

## 5.22) Sélectionner une couleur



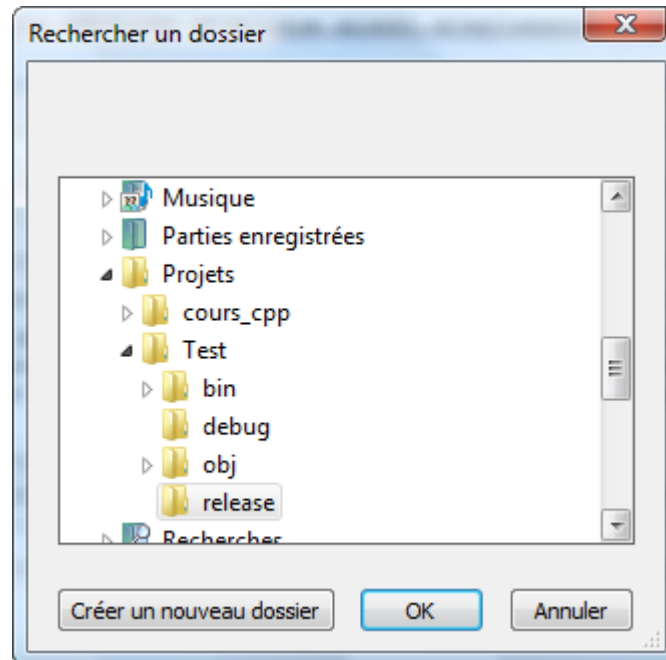
# 5.23) Sélection d'un fichier ou d'un dossier

La sélection de fichiers et de dossiers est gérée par la classe `QFileDialog`

```
QString dossier = QFileDialog::getExistingDirectory(this);
```

- retourne un `QString` contenant le chemin complet vers le dossier demandé.

La fenêtre qui s'ouvre devrait ressembler à cela :



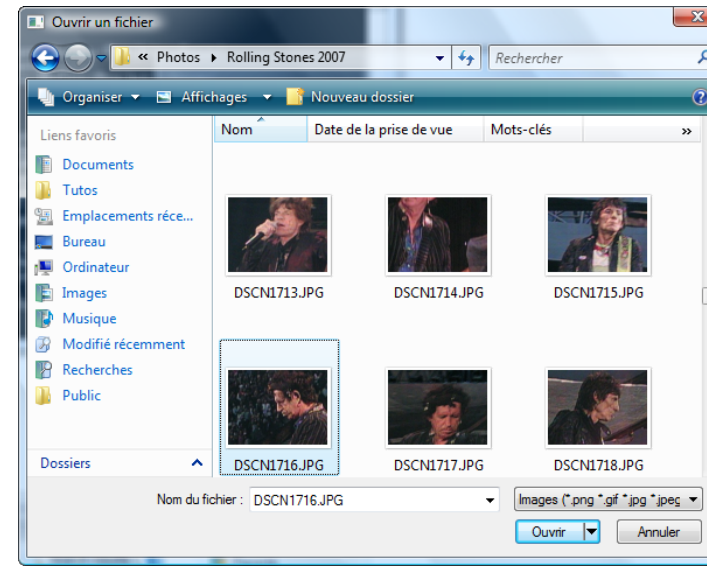
# 5.24) Ouverture d'un fichier

## (QFileDialog::getOpenFileName)

Le code demande d'ouvrir un fichier image. Le chemin vers le fichier est stocké dans un `QString`, que l'on affiche ensuite via une `QMessageBox`

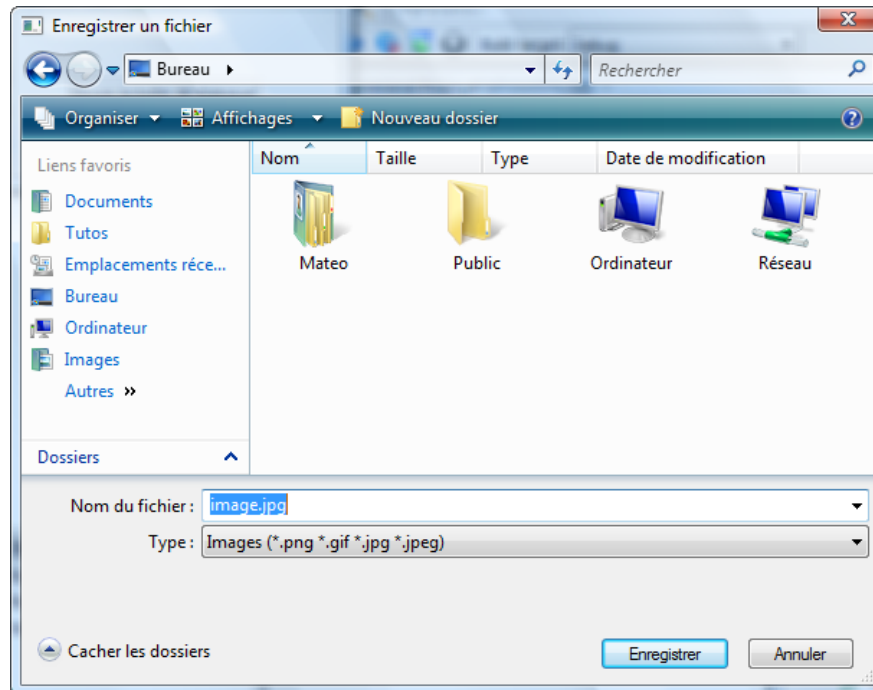
```
void MaFenetre::ouvrirDialogue() {  
    QString fichier = QFileDialog::getOpenFileName(this,  
        "Ouvrir un fichier", QString(),  
        "Images (*.png *.gif *.jpg *.jpeg)");  
    QMessageBox::information(this, "Fichier",  
        "Vous avez sélectionné :\n" + fichier);  
}
```

Le 3<sup>ème</sup> paramètre de `getOpenFileName` est le nom du répertoire par défaut dans lequel l'utilisateur est placé.



# 5.25) Enregistrement d'un fichier (QFileDialog::getSaveFileName)

```
QString fichier = QFileDialog::getSaveFileName(this,  
    "Enregistrer un fichier", QString(),  
    "Images (*.png *.gif *.jpg *.jpeg)");
```



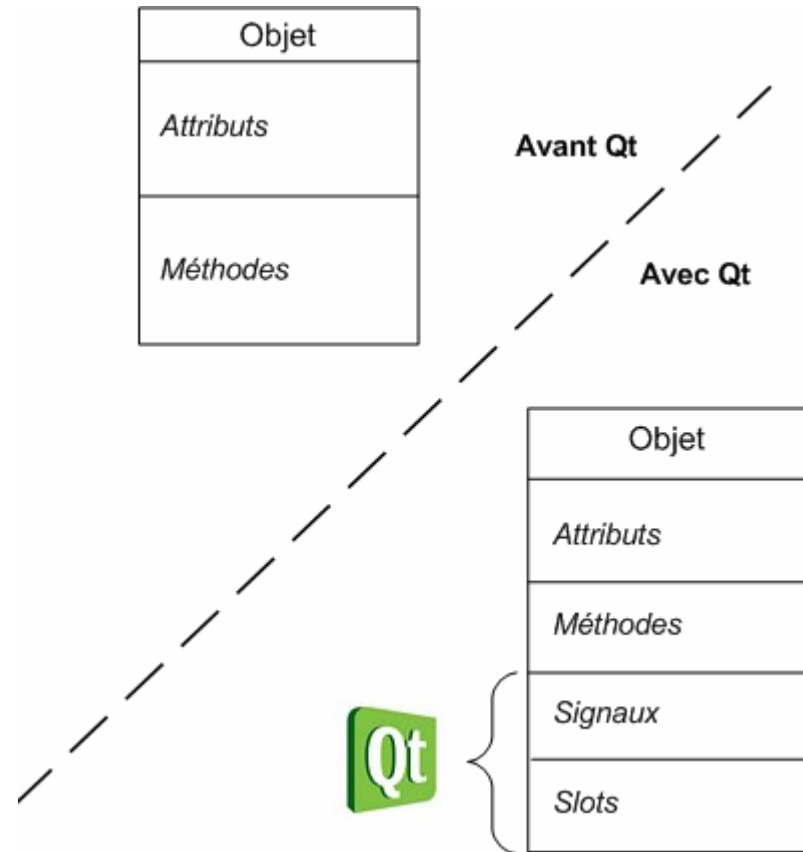
## 6) La gestion des événements

- Notions de signal et slot.
- Déclaration de signaux et de slots.
- Installer des filtres d'événement.
- Accéder à l'application pendant un traitement lourd (`timer` et `hasPendingEvents()`).



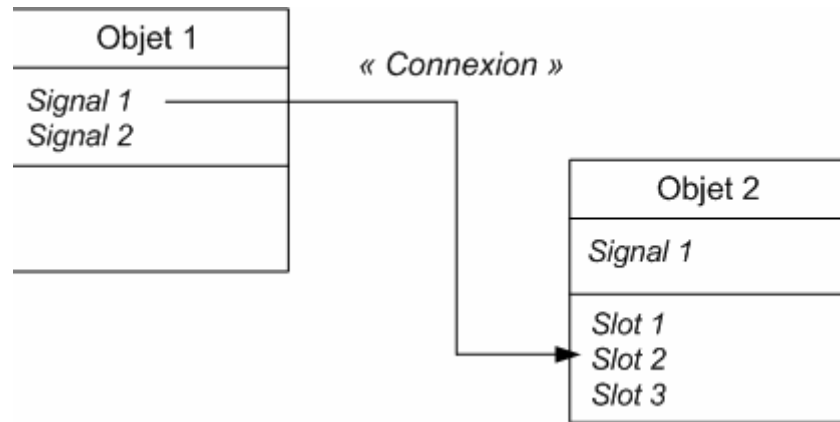
# 6.1) Le principe des signaux et slots

- Une application de type GUI réagit à partir d'évènements.
- Un signal : c'est un message envoyé par un widget lorsqu'un évènement se produit.
  - Exemple : on a cliqué sur un bouton.
- Un slot : c'est la fonction qui est appelée lorsqu'un évènement s'est produit. On dit que le signal appelle le slot. Concrètement, un slot est une méthode d'une classe.
  - Exemple : le slot `quit()` de la classe `QApplication`, qui provoque l'arrêt du programme.
- Les signaux et les slots sont considérés par Qt comme des éléments d'une classe à part entière, en plus des attributs et des méthodes.



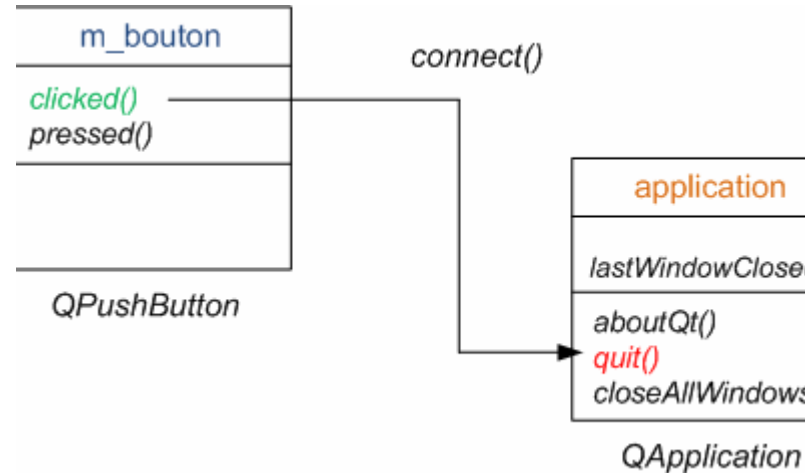
# 6.1) Le principe des signaux et slots

- Un signal est un message envoyé par l'objet (par exemple "on a cliqué sur le bouton").
- Un slot est une méthode. En fait, c'est une méthode classique comme toutes les autres, à la différence près qu'elle a le droit d'être connectée à un signal.
- Avec Qt, on dit que l'on connecte des signaux et des slots entre eux.  
Supposons 2 objets, chacun ayant ses propres attributs, méthodes, signaux et slots



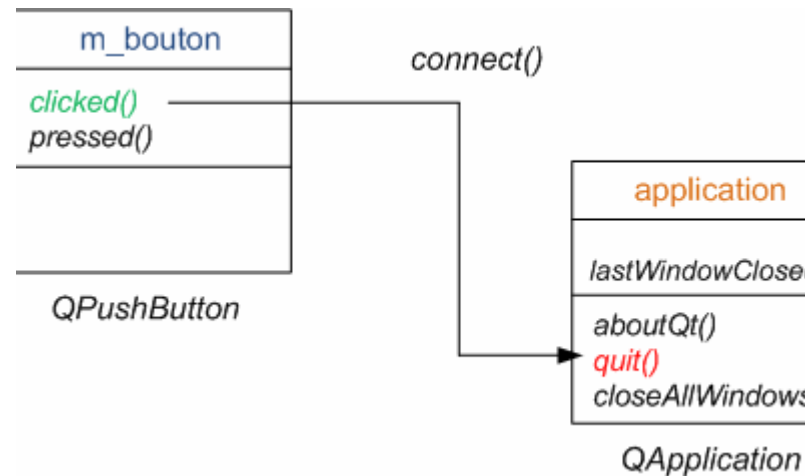
# 6.2) Connexion d'un signal à un slot simple

- 2 objets, l'un de type `QPushButton`, et l'autre de type `QApplication`.
- Il y a d'un côté notre bouton appelé "m\_bouton" (de type `QPushButton`), et de l'autre l'application (de type `QApplication`, utilisée dans le `main`).
- Par exemple on connecte le signal "bouton cliqué" au slot "quitter l'application". Ainsi, un clic sur le bouton provoquerait l'arrêt de l'application.
- Pour ce faire, il faut utiliser une méthode statique de la classe `QObject` : `connect()`.



# 6.3) Le principe de la méthode `connect()`

- `connect()` est une méthode statique.
- Une méthode statique est une méthode d'une classe que l'on peut appeler sans créer d'objet. C'est en fait exactement comme une fonction classique du langage C.
- Comme `connect()` appartient à la classe `QObject`, il faut donc écrire :  
`QObject::connect();`
- La méthode `connect` prend 4 arguments :
  - Un pointeur vers l'objet qui émet le signal.
  - Le nom du signal que l'on souhaite "intercepter".
  - Un pointeur vers l'objet qui contient le slot récepteur.
  - Le nom du slot qui doit s'exécuter lorsque le signal se produit.
- Il existe aussi une méthode `disconnect()` permettant de casser la connexion entre 2 objets.



# 6.4) Utilisation de la méthode `connect()` pour quitter

`connect()` est une méthode de la classe `QObject`. Comme la classe `MaFenetre` hérite de `QObject` indirectement, elle possède elle aussi cette méthode.

`m_bouton` : c'est un pointeur vers le bouton qui va émettre le signal.

`SIGNAL(clicked())` : `SIGNAL()` est une macro du pré processeur. Qt transformera en un code "acceptable" pour la compilation.

`qApp` : c'est un pointeur vers l'objet de type `QApplication` créé dans le main.

`SLOT(quit())` : c'est le slot qui doit être appelé lorsqu'on a cliqué sur le bouton.

```
#include "MaFenetre.h"
```

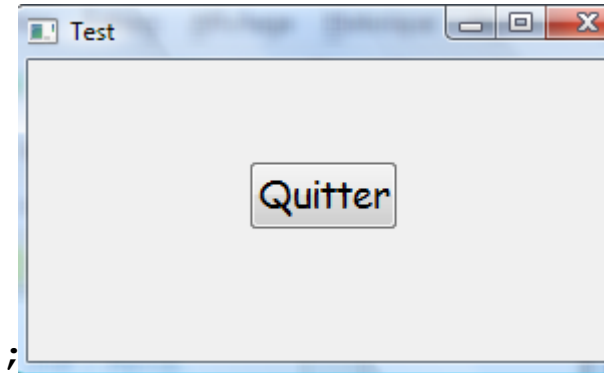
```
MaFenetre::MaFenetre() : QWidget()  
{
```

```
    setFixedSize(300, 150);
```

```
    m_bouton = new QPushButton("Quitter", this);  
    m_bouton->setFont(QFont("Comic Sans MS", 14));  
    m_bouton->move(110, 50);
```

```
    // Connexion du clic du bouton à la fermeture de l'application  
    QObject::connect(m_bouton, SIGNAL(clicked()), qApp, SLOT(quit()));
```

```
}
```



# 6.5) Utilisation de la méthode `connect()` pour afficher "A propos"

Créer un 2<sup>ème</sup> bouton qui se chargera d'afficher la fenêtre "A propos de Qt".

```
#include "MaFenetre.h"
```

```
MaFenetre::MaFenetre() : QWidget()
```

```
{  
    setFixedSize(300, 150);
```

```
    m_quitter = new QPushButton("Quitter", this);
```

```
    m_quitter->setFont(QFont("Comic Sans MS", 14));
```

```
    m_quitter->move(110, 50);
```

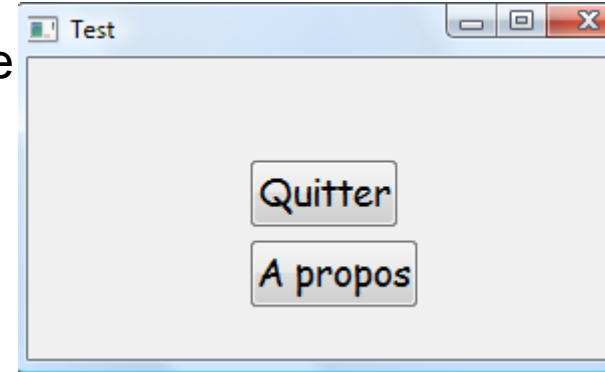
```
    QObject::connect(m_quitter, SIGNAL(clicked()), qApp, SLOT(quit()));
```

```
    m_aPropos = new QPushButton("A propos", this);
```

```
    m_aPropos->setFont(QFont("Comic Sans MS", 14));
```

```
    m_aPropos->move(110, 90);
```

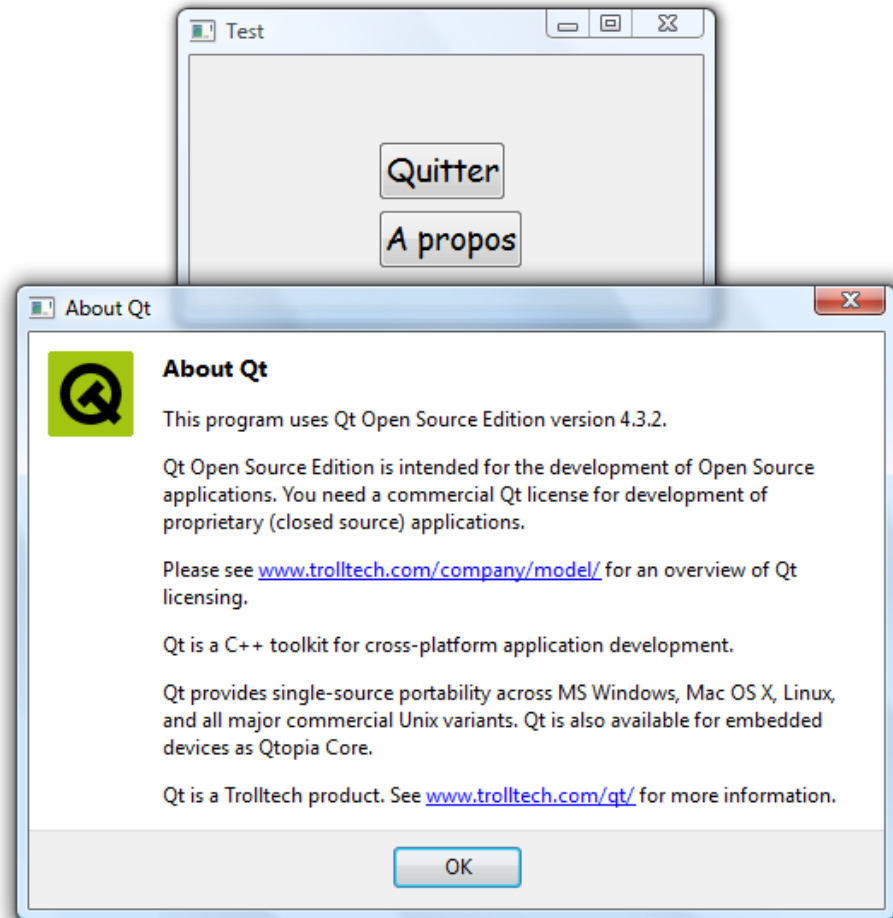
```
    QObject::connect(m_aPropos, SIGNAL(clicked()), qApp, SLOT(aboutQt()));  
}
```



# 6.5) Utilisation de la méthode `connect()` pour afficher "A propos"

Le bouton "Quit" ferme toujours l'application.

Quant à "A propos", il provoque l'ouverture de la fenêtre "A propos de Qt".



# 6.6) Des paramètres dans les signaux et slots

- Les signaux et les slots peuvent s'échanger des paramètres,
- De nouveaux widgets dans la fenêtre.
  - `QSlider` : un curseur qui permet de définir une valeur.
  - `QLCDNumber` : un widget qui affiche un nombre.

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE
```

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLCDNumber>
#include <QSlider>
```

```
class MaFenetre : public QWidget
{
    public:
        MaFenetre();

    private:
        QLCDNumber *m_lcd;
        QSlider *m_slider;
};
```

```
#endif
```



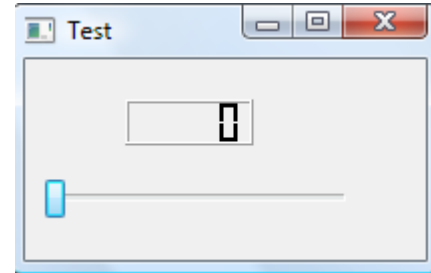
## 6.6) Des paramètres dans les signaux et slots

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

    m_lcd = new QLCDNumber(this);
    m_lcd->setSegmentStyle(QLCDNumber::Flat);
    m_lcd->move(50, 20);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setGeometry(10, 60, 150, 20);
}
```



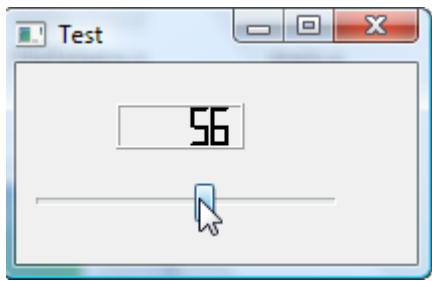

# 6.7) Connexion avec des paramètres

- On veut que l'afficheur LCD change de valeur en fonction de la position du curseur du slider.
- On dispose du signal et du slot suivant :
  - Le signal `valueChanged(int)` du `QSlider` : il est émis dès que l'on change la valeur du curseur du slider en le déplaçant. La particularité de ce signal est qu'il envoie un paramètre de type `int` (la nouvelle valeur du slider).
  - Le slot `display(int)` du `QLCDNumber` : il affiche la valeur qui lui est passée en paramètre.

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd,  
                 SLOT(display(int)));
```

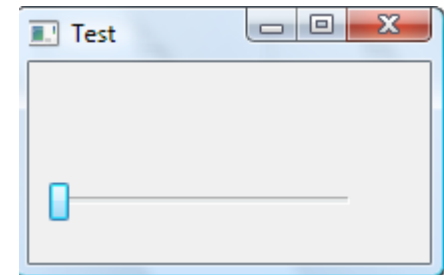
- Il suffit d'indiquer le type du paramètre envoyé, `int`, sans donner de nom à ce paramètre.
- Qt fait automatiquement la connexion entre le signal et le slot et "transmet" le paramètre au slot.

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd, SLOT(display(int)));
```



# 6.8) Créer ses propres signaux et slots

- Pour pouvoir créer son propre signal ou slot dans une classe, il faut que celle-ci dérive directement ou indirectement de `QObject`.
  - C'est le cas de la classe `MaFenetre` : elle hérite de `QWidget`, qui hérite de `QObject`.
- Créer son propre slot
  - **But**: le `QSlider` contrôle la largeur de la fenêtre.
  - **Solution**: le signal `valueChanged(int)` du `QSlider` puisse être connecté à un slot de la fenêtre (de type `MaFenetre`). Ce nouveau slot aura pour rôle de modifier la largeur de la fenêtre.
  - Comme il n'existe pas de slot "changerLargeur" dans la classe `QWidget`, il faut le créer.
  - Pour créer ce slot, il faut modifier la classe `MaFenetre`.



# 6.8) Créer ses propres signaux et slots

```
class MaFenetre : public QWidget
```

```
{
```

```
    Q_OBJECT
```



La macro porte le nom de `Q_OBJECT` (tout en majuscules) et doit être placée tout au début de la déclaration de la classe :

```
    public:
```

```
    MaFenetre();
```

```
    public slots:
```

```
    void changerLargeur(int largeur);
```

```
    private:
```

```
    QSlider *m_slider;
```

```
};
```

La macro `Q_OBJECT` "prépare" le compilateur à accepter un nouveau mot-clé : "**slot**". Il est possible de créer une section "slots":

## 6.8) Créer ses propres signaux et slots

```
void MaFenetre::changerLargeur(int largeur)
{
    setFixedSize(largeur, 100);
}
```

Le slot prend en paramètre un entier : la nouvelle largeur de la fenêtre. Il appelle la méthode `setFixedSize` de la fenêtre et lui envoie la nouvelle largeur reçue.

```
MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setRange(200, 600);
    m_slider->setGeometry(10, 60, 150, 20);

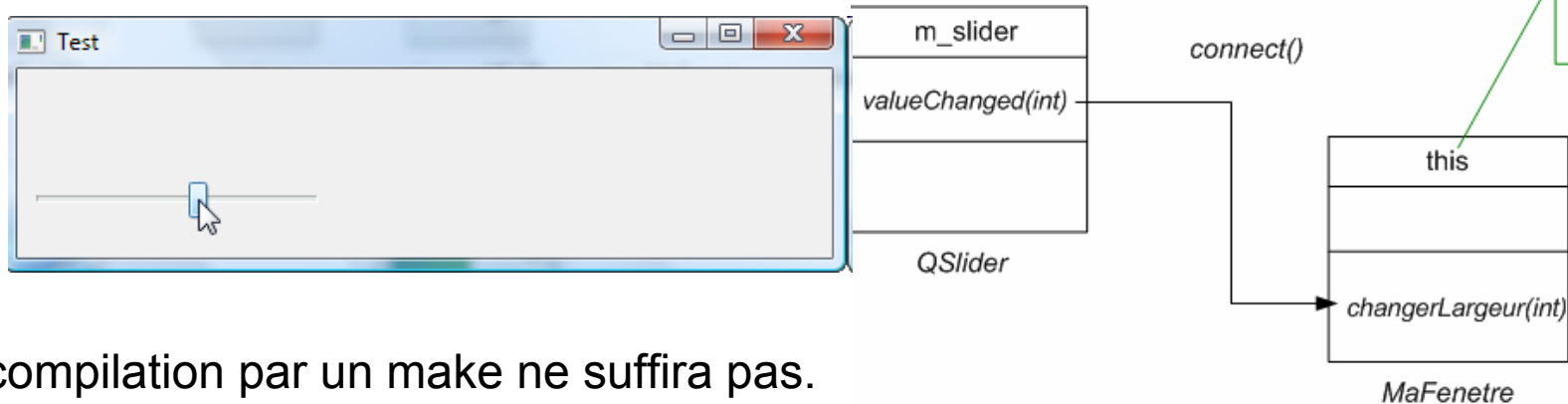
    QObject::connect(m_slider, SIGNAL(valueChanged(int)), this,
                     SLOT(changerLargeur(int)));
}
```

Le slider est limité entre 200 et 600

# 6.8) Créer ses propres signaux et slots

La connexion se fait entre le signal `valueChanged(int)` du `QSlider`, et le slot `changerLargeur(int)` de la classe `MaFenetre`.

- il faut pouvoir indiquer un pointeur vers l'objet actuel (la fenêtre)



la compilation par un `make` ne suffira pas.

Avec la macro `Q_OBJECT`, Qt a besoin d'appeler un pré-compilateur qui lui est propre appelé le `moc` (Meta-Object Compiler).

- Relancez `qmake` avant de faire votre `make`, et Qt fera le travail de "traduction" du slot en quelque chose de compréhensible pour le compilateur C++.
- `qmake` a provoqué la création d'un fichier intermédiaire `moc_MaFenetre.cpp`, ce fichier fournit des informations indispensables au compilateur.
- faire un `make`, la compilation doit bien se passer.

# 6.9) Créer son propre signal

- si le slider horizontal arrive à sa valeur maximale (600), alors on émet un signal "agrandissementMax".
  - La fenêtre doit pouvoir émettre l'information comme quoi elle est agrandie au maximum.
  - Après, nous connecterons ce signal à un slot pour vérifier que notre programme réagit correctement.

```
class MaFenetre : public QWidget {  
    Q_OBJECT  
  
public:  
    MaFenetre();  
  
public slots:  
    void changerLargeur(int largeur);  
  
signals:  
    void agrandissementMax();  
  
private:  
    QSlider *m_slider;  
};
```

Les signaux se présentent en pratique sous forme de méthodes (comme les slots) à la différence près qu'on ne les implémente pas dans le .cpp.

C'est Qt qui fournit l'implémentation.

Si vous tentez d'implémenter un signal, il y a une erreur "Multiple definition of...".

Un signal peut passer un ou plusieurs paramètres. Un signal doit toujours renvoyer void.

## 6.9) Créer son propre signal

- Maintenant que le signal est défini, il faut que la classe puisse l'émettre à un moment.
- Quand sait on que la fenêtre a été agrandie au maximum ? Dans le slot `changerLargeur` ! Il suffit de tester dans ce slot si la largeur correspond au maximum (600), et d'émettre alors le signal.

```
void MaFenetre::changerLargeur(int largeur) {  
    setFixedSize(largeur, height());  
  
    if (largeur == 600) {  
        emit agrandissementMax();  
    }  
}
```

La méthode s'occupe toujours de redimensionner la fenêtre, mais vérifie en plus si la largeur a atteint le maximum (600). Si c'est le cas, elle émet le signal `agrandissementMax()`.

Pour émettre un signal, on utilise le mot-clé `emit`. On comprend "Émettre le signal `agrandissementMax()`".

Il suffit d'appeler le signal comme ceci : `emit monSignal(parametre1, parametre2, ...);`

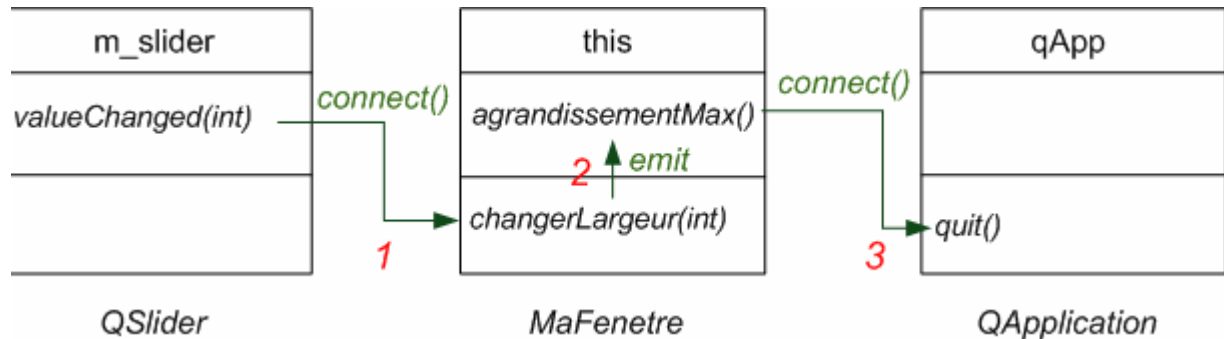


# 6.9) Créer son propre signal

- Il reste à connecter le nouveau signal à un slot. Il est possible de connecter ce signal à l'application (à l'aide du pointeur global `qApp`) pour provoquer l'arrêt du programme.
- Dans le constructeur de `MaFenetre`:

```
QObject::connect(this, SIGNAL(agrandissementMax()), qApp,  
                SLOT(quit()));
```

- normalement le programme s'arrête quand la fenêtre est agrandie au maximum.
- Le schéma des signaux qu'on vient d'émettre et connecter est le suivant :



- + Le signal `valueChanged` du slider a appelé le slot `changerLargeur` de la fenêtre.
- + Le slot a fait changer la largeur de la fenêtre et a vérifié si la fenêtre était arrivée à sa taille maximale. Lorsque cela a été le cas, le signal personnalisé `agrandissementMax()` a été émis.
- + Le signal `agrandissementMax()` de la fenêtre était connecté au slot `quit()` de l'application, ce qui a provoqué la fermeture du programme.

# 7) Utilisation de XML avec QT

- Un rappel sur XML.
- Les parsers DOM et SAX.
- Parsing de documents XML en utilisant QT.

# 7.1) XML

- Qt propose le support de XML comme module
- Qt offre 2 manières d'interagir avec le contenu XML: DOM et SAX.
  - SAX est le plus simple des deux, à lire et analyser.
  - DOM lit le fichier XML entièrement dans un arbre en mémoire. Cet arbre peut alors être lu et manipulé avant d'être mis de côté ou écrit de nouveau sur disque.
- SAX rend difficile la modification des données fournies. En revanche, il exige un espace mémoire très faible tout en étant aussi utile que DOM dans beaucoup de situations.
- DOM exige plus de mémoire. Le document entier doit être stocké en mémoire.
  - cela donne la capacité de modifier et travailler avec le document librement en mémoire et de le mettre alors de nouveau sur disque.

```
<tag attribute="value" />  
<tagWithData attribute="value" anotherAttribute="value">  
data  
</tagWithData>
```

# 7.2) Écrire en utilisant DOM

## 1. Créer un document (un QDomDocument).

- "AdBookML" est par convention le nom choisi pour le document.

```
QDomDocument doc( "AdBookML" );
```

## 2. Créer un élément racine.

- un élément racine doit être l'endroit à partir duquel commencer. l'élément racine s'appellera adbook.

```
QDomElement root = doc.createElement( "adbook" );
```

```
doc.appendChild( root );
```

## 3. Pour chaque contact, l'insérer dans le document.

```
QDomElement ContactToNode( QDomDocument &d, const Contact &c )
```

```
{
```

```
    QDomElement cn = d.createElement( "contact" );
```

```
    cn.setAttribute( "name", c.name );
```

```
    cn.setAttribute( "phone", c.phone );
```

```
    cn.setAttribute( "email", c.eMail );
```

```
    return cn;
```

```
}
```

- Chaque appel à cette fonction finira par l'ajout d'un élément à la racine du document.

# 7.2) Écrire en utilisant DOM

4.Écrire le résultat dans un fichier.

- en ouvrant un fichier, en créant un flux de texte et puis en appelant la méthode toString() du document DOM

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

```
    QDomDocument doc( "AdBookML" );  
    QDomElement root = doc.createElement( "adbook" );  
    doc.appendChild( root );
```

```
    Contact c;  
    c.name = "Kal";  
    c.eMail = "kal@goteborg.se";  
    c.phone = "+46(0)31 123 4567";
```

```
    root.appendChild( ContactToNode( doc, c ) );  
    c.name = "Ada";  
    c.eMail = "ada@goteborg.se";  
    c.phone = "+46(0)31 765 1234";  
    root.appendChild( ContactToNode( doc, c ) );
```

```
    QFile file( "test.xml" );  
    if( !file.open( IO_WriteOnly ) )  
        return -1;  
    QTextStream ts( &file );  
    ts << doc.toString();  
    file.close();  
    return 0;
```

```
}
```

```
<!DOCTYPE AdBookML>
```

```
<adbook>
```

```
    <contact email="kal@goteborg.se"  
phone="+46(0)31 123 4567" name="Kal" />
```

```
    <contact email="ada@goteborg.se"  
phone="+46(0)31 765 1234" name="Ada" />  
</adbook>
```

# 7.3) Lire en utilisant DOM

## 1. Créer le document DOM à partir d'un fichier.

- Un document vide est créé et le contenu du fichier lui est assigné (si le fichier est ouvert correctement). Après ceci, le fichier peut-être abandonné puisque le document entier a été lu en mémoire.

```
QDomDocument doc( "AdBookML" );
QFile file( "test.xml" );
if( !file.open( IO_ReadOnly ) )
    return -1;
if( !doc.setContent( &file ) )
{
    file.close();
    return -2;
}
file.close();
```

## 2. Trouver la racine et s'assurer que c'est un carnet d'adresses.

- on le vérifie pour s'assurer qu'il s'agit d'un élément "adbook" et rien d'autre.

```
QDomElement root = doc.documentElement();
if( root.tagName() != "adbook" )
    return -3;
```

# 7.3) Lire en utilisant DOM

3.Trouver tous les contacts.

4.Trouver tous les attributs intéressants de chaque élément de contact.

- Chaque élément est vérifié, si c'est un contact, ses attributs sont analysés, autrement il est ignoré

```
QDomNode n = root.firstChild();
while( !n.isNull() ) {
    QDomElement e = n.toElement();
    if( !e.isNull() ) {
        if( e.tagName() == "contact" ) {
            Contact c;

            c.name = e.attribute( "name", "" );
            c.phone = e.attribute( "phone", "" );
            c.eMail = e.attribute( "email", "" );

            QMessageBox::information( 0, "Contact", c.name + "\n" + c.phone +
                "\n" + c.eMail );
        }
    }

    n = n.nextSibling();
}
```

```
class AdBookParser : public QDomDefaultHandler {
```

```
public:
```

```
    bool startDocument() {
        inAdBook = false;
        return true;
    }

    bool endElement(const QString&, const QString&, const QString &name) {
        if( name == "adbook" )
            inAdBook = false;
        return true;
    }
}
```

```
    bool startElement(const QString&, const QString&, const QString &name,
const QDomAttributes &attrs) {
```

```
        if( inAdBook && name == "contact" ) {
            QString name, phone, email;
            for( int i=0; i<attrs.count(); i++) {
                if( attrs.localName( i ) == "name" )
                    name = attrs.value( i );
                else if( attrs.localName( i ) == "phone" )
                    phone = attrs.value( i );
                else if( attrs.localName( i ) == "email" )
                    email = attrs.value( i );
            }
            QMessageBox::information( 0, "Contact", name + "\n" + phone + "\n" + email )
        }
        else if( name == "adbook" )
            inAdBook = true;
        return true;
    }
}

private:
    bool inAdBook;
```

## 7.4) Lire en utilisant SAX



## 7.4) Lire en utilisant SAX

- La méthode `startDocument` est appelée d'abord quand le document commence.
- Pour chaque balise qui s'ouvre, la méthode `startElement` est appelée.
- Elle s'assure qu'une balise `adbook` est trouvée, l'état est modifié, et si une balise `contact` est trouvée alors qu'on est à l'intérieur d'une balise `adbook` les attributs sont lus.
- La méthode `endElement` est appelée chaque fois qu'une balise fermante est rencontrée. Si c'est une balise `adbook` qui se ferme, l'état est mis à jour.

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

    AdBookParser handler;
    QFile file( "test.xml" );
    QXmlInputSource source( file );

    QXmlSimpleReader reader;
    reader.setContentHandler( &handler );
    reader.parse( source );
    return 0;
}
```

## 7.5) Exemple avec XML

```
class Contact {  
public:  
    QString name, eMail, phone;  
    Contact(QString iName = "", QString iPhone = "", QString iEMail = "" )  
    Contact(const QDomElement &e );  
  
    QDomElement createXMLNode( QDomDocument &d );  
};
```

un constructeur pour créer un Contact depuis un QDomElement

# 7.5) Exemple avec XML

```
#include "contact.h"
```

```
Contact::Contact( QString iName, QString iPhone, QString iEMail )
{
    name = iName;
    phone = iPhone;
    eMail = iEMail;
}
```

```
Contact::Contact( const QDomElement &e )
{
    name = e.attribute( "name", "" );
    phone = e.attribute( "phone", "" );
    eMail = e.attribute( "email", "" );
}
```

```
QDomElement Contact::createXMLNode( QDomDocument &d )
{
    QDomElement cn = d.createElement( "contact" );

    cn.setAttribute( "name", name );
    cn.setAttribute( "phone", phone );
    cn.setAttribute( "email", eMail );

    return cn;
}
```

```
        !m_main->load( const QString::fromStdString( filename ), {
```

```
QFile file( filename );
```

```
if( !file.open( IO_ReadOnly ) ) {
```

```
    QMessageBox::warning( this, "Loading", "Failed to load file." );
```

```
    return;
```

```
}
```

```
QDomDocument doc( "AdBookML" );
```

```
if( !doc.setContent( &file ) ) {
```

```
    QMessageBox::warning( this, "Loading", "Failed to load file." );
```

```
    file.close();
```

```
    return;
```

```
}
```

```
file.close();
```

```
QDomElement root = doc.documentElement();
```

```
if( root.tagName() != "adbook" ) {
```

```
    QMessageBox::warning( this, "Loading", "Invalid file." );
```

```
    return;
```

```
}
```

```
m_contacts.clear();
```

```
lvContacts->clear();
```

```
QDomNode n = root.firstChild();
```

```
while( !n.isNull() ) {
```

```
    QDomElement e = n.toElement();
```

```
    if( !e.isNull() ) {
```

```
        if( e.tagName() == "contact" ) {
```

```
            Contact c( e );
```

```
            m_contacts.append( c );
```

```
            lvContacts->insertItem(new QListViewItem(lvContacts,c.name,c.eMail,c.phone)
```

```
        }
```

```
    }
```

```
    n = n.nextSibling();
```

## 7.5) Exemple avec XML

## 7.5) Exemple avec XML

```
void frmMain::save( const QString &filename ) {
    QDomDocument doc( "AdBookML" );
    QDomElement root = doc.createElement( "adbook" );
    doc.appendChild( root );
    for( QValueList<Contact>::iterator it = m_contacts.begin();
                                                it != m_contacts.end(); ++it )
        root.appendChild( (*it).createXMLNode( doc ) );

    QFile file( filename );
    if( !file.open( IO_WriteOnly ) ) {
        QMessageBox::warning( this, "Saving", "Failed to save file." );
        return;
    }

    QTextStream ts( &file );
    ts << doc.toString();
    file.close();
}
```

L'interface utilisateur doit pouvoir demander à l'utilisateur des noms de fichier, pour le chargement et la sauvegarde

## 7.5) Exemple avec XML

```
void frmMain::loadFile() {
    QString filename = QFileDialog::getOpenFileName( QString::null,
        "Addressbooks (*.adb)", this, "file open", "Addressbook File Open" )
    if ( !filename.isEmpty()) {
        m_filename = filename;
        load( filename );
    }
}

void frmMain::saveFile() {
    if( m_filename.isEmpty()) {
        saveFileAs();
        return;
    }
    save( m_filename );
}

void frmMain::saveFileAs() {
    QString filename = QFileDialog::getSaveFileName( QString::null,
        "Addressbooks (*.adb)", this, "file save as", "Addressbook Save As" )
    if ( !filename.isEmpty()) {
        m_filename = filename;
        save( m_filename );
    }
}
```

## 8) Le système de plugin de QT

- Comprendre ce que sont les plugins avec QT.
- Les différentes classes de plugin (`QStylePlugin ...`).
- Les éléments nécessaires à la mise en place d'un plugin pour QT.
- Mise au point d'application gérant des plug-ins.

# 8.1) Définition

- En informatique, un plugin est un moyen de rajouter de nouvelles fonctionnalités à un logiciel de base.
- Ces modules sont en général utilisés pour qu'un programme puisse évoluer facilement.
  - De plus, d'autres personnes voulant aider le logiciel peuvent à leur tour en créer pour ajouter de nouvelles fonctionnalités.
- On peut rendre un programme totalement modulaire.
- Pour créer un plugin, il suffit de créer une définition puis de l'implémenter.
- Il faut donc aussi modifier la base du programme en lui donnant un moyen de charger les plugins.



## 8.2) Créer une définition

- Un plugin doit avoir un but minimal et une sorte de définition.
- La définition est la classe qui lie le projet mère à ses plugins.
  - pour pouvoir concevoir un plugin, il y aura besoin de cette définition.
  - pour que le projet utilise les plugins, il a besoin de savoir comment ils fonctionnent.
- La définition est donc dans les 2 projets (projet mère et projet de chaque plugin).
- Cette définition est une simple interface, c'est-à-dire une classe abstraite contenant une liste de fonctions que tous les plugins du même type auront.
- Par exemple la classe `Animal` qui permettra de créer des plugins de type `Animal` aura comme méthodes :

- `void manger(const Nourriture &repas) ;`
- `void bouger(const Position &destination) ;`
- `void attaquer(const Cible &cible) ;`
- ...

```
class MaClass {
public:
    virtual ~MaClass() {}

    virtual void maFonction() = 0;
    virtual void maFonctConst() const = 0;
};
```

## 8.2) Créer une définition

- Qt veut que nous déclarions explicitement une interface. Pour cela, on utilise la macro-définition `Q_DECLARE_INTERFACE`.
- Elle prend en paramètre 2 choses.
  - La 1ère est le nom de la classe interface
  - La 2ème est l'identifiant de l'interface sous forme de chaîne de caractères. Attention, celle-ci doit finir par le nom de la classe et être unique.
- En général, cette macro-définition est située juste après la déclaration de la classe.
- il faut que la macro-définition soit appelée hors du namespace.
  - on utilise l'opérateur de portée « `::` » (par exemple : « `MonNamespace::MaClasse` »).

## 8.2) Créer une définition

```
#ifndef __INTER_H__
#define __INTER_H__
#include <QtPlugin>
#include "EtreVivant.h"
class Animal : public EtreVivant {
    public:
        virtual ~Inter1() { }
        virtual void mange(const EtreVivant &proie) = 0;
        virtual void attaquer(const EtreVivant &victime) = 0;
        // On peut mettre des variables.
    protected:
        int m_Faim;
        int m_Vie;
};
Q_DECLARE_INTERFACE(Animal, "Mon programme.Animal")
namespace Out {
    class Console { // Ici on a l'interface dans le namespace.
    public:
        virtual Console() {}
        virtual print(const std::string &str) = 0;
    };
}
Q_DECLARE_INTERFACE(Out::Console, "Mon programme.Console")
#endif
```

## 8.3) Création de plug in

- Un plugin est avant tout un objet qui respecte une définition.
  - pour cela, il suffit de le faire hériter de l'interface.
- Puisqu'un plugin est aussi un objet Qt, alors il doit hériter de `QObject` (directement ou indirectement).

```
class MyPlugin : public QObject, public MyInter { };
```

- Le fait de ne pas dériver du plugin de `QObject` avant toute autre classe entraînera forcément une erreur.
  - faire hériter l'interface dans un 1<sup>er</sup> temps de `QObject`, puis de toute autre classe.
- Il faut ensuite dire à Qt que cette classe n'est pas comme les autres.
  - c'est un `QObject`, donc on peut mettre la macro-définition `Q_OBJECT`.
  - il faut signaler à Qt que l'on utilise une ou des interfaces.  
on utilise la macro-définition `Q_INTERFACES`. Elle attend la liste des interfaces séparées par des espaces.

## 8.3) Création de plug in

```
class Chien : public QObject, public Animal {  
    // Un chien est un animal, il dérive donc de l'interface Animal.  
    Q_OBJECT  
    Q_INTERFACES (Animal)  
    // Si notre plugin dériverait de plusieurs interfaces,  
    // il aurait fallu donner la liste à la macro-définition  
    // avec des espaces entre chaque nom. Comme cela :  
    // Q_INTERFACES (MyInter1 MyInter2 ...)  
};
```

- Pour pouvoir utiliser un plugin, il faut qu'il soit concret.
  - un plugin est avant tout une classe. Or si une classe est abstraite, elle ne peut pas être instanciée.
  - comme le plugin hérite d'une interface abstraite, il faudra donc définir toutes les méthodes virtuelles pures.
- Si le plugin a besoin de plus de fonctions, on peut en rajouter.
  - le programme de base ne pourra toucher qu'aux méthodes définies dans l'interface.

## 8.3) Création de plug in

- Dans l'implémentation, il faut rajouter la macro-définition `Q_EXPORT_PLUGIN2(nomPlugin, nomClass)`.
- Le nom du plugin est celui que l'on définira dans le `.pro` du plugin.
  - Le nom de la classe est celui de la classe d'entrée du plugin, donc celle qui hérite de l'interface.
  - On peut donc mettre plusieurs classes dans un plugin.

```
Q_EXPORT_PLUGIN2(nom_plugin, MyPlugin)  
// Pas de point-virgule après la macro-définition.
```

- Il faut maintenant changer le `.pro` du projet.
  - En effet, notre « projet » ne contient pas de main, donc le compilateur ne trouvera pas de point d'entrée s'il en cherche.
  - Il faut donc le signaler en lui disant que l'on veut faire une bibliothèque. Pour cela, on utilise la variable `TEMPLATE` du `.pro` comme ceci :

```
TEMPLATE = lib
```

## 8.3) Création de plug in

- Il existe différents types de bibliothèques : les dynamiques, les statiques et aussi les plugins.  
On utilise alors `CONFIG` pour montrer à Qt que c'est un plugin :

```
CONFIG += plugin
```

- Pour finir, on définit le nom de la cible (c'est-à-dire le plugin) avec `TARGET`.  
C'est aussi le nom du plugin qu'on avait mis dans la macro-définition `Q_EXPORT_PLUGIN2` :

```
TARGET = nom_plugin
```

- Il ne reste plus que la partie sur la création d'un chargeur de plugins et la partie théorique est finie.

## 8.4) Faire un chargeur de plug in

- Il est relativement simple de charger un plugin car Qt a une classe prête pour cela : `QPluginLoader`.
- Pour en créer un, il suffit de donner le chemin vers le plugin.
  - on peut récupérer un pointeur vers le plugin avec la méthode `QPluginLoader::instance()` . Elle retourne un `QObject*`, donc il faudra la réinterpréter en `MyPlugin*`, par exemple.
  - Pour la réinterpréter, soit on utilise le cast C++ `reinterpret_cast<T>(obj)` , soit on utilise un cast défini par Qt : `qobject_cast<T>(obj)` .
- Puisque l'on utilise Qt et que `qobject_cast` utilise `reinterpret_cast`, le mieux est d'utiliser `qobject_cast`.

```
QPluginLoader loader("./cheminVersMonPlugin");  
    // On charge le plugin en lui donnant juste le chemin.  
if(QObject *plugin = loader.instance()) {  
    // On prend l'instance de notre plugin sous forme de QObject.  
    // On vérifie en même temps s'il n'y a pas d'erreur.  
    MyPlugin* myPlugin = qobject_cast<MyPlugin *>(plugin);  
    // On réinterprète alors notre QObject en MyPlugin  
}
```



## 8.4) Faire un chargeur de plug in

- Ce code permet de charger un plugin, mais il est assez basique. Un système de plugin doit permettre à une application d'évoluer facilement, elle doit donc savoir si un plugin a été ajouté. Si on met tous les plugins du même type dans le même dossier, alors on peut utiliser `QDir` pour lire ce dossier et donc connaître tous les plugins.
  - il faut créer un `QDir` en lui donnant le dossier du programme pour être sûr de connaître le bon chemin vers le plugin.
  - la fonction `qApp->applicationDirPath()` retourne le chemin vers l'application.
  - il faut déplacer le `QDir` dans le dossier du plugin avec la méthode « `cd` ».

```
QDir plugDir = QDir(qApp->applicationDirPath());  
    // On place le QDir dans le dossier de notre exécutable.  
plugDir.cd("../cheminSecretVersLeTresor");  
    // Puis on le déplace dans le dossier des plugins.
```

## 8.4) Faire un chargeur de plug in

- Il faut maintenant boucler avec la liste des fichiers se trouvant dans le répertoire. `QDir::entryList()` retourne la liste des fichiers avec comme arguments la macro `QDir::Files` qui est un filtre : il ne prendra que les fichiers du dossier, les répertoires ne seront pas listés. Elle retourne un `QStringList`.
- Puisqu'un `QStringList` est une `List`, il est alors possible d'utiliser un `foreach`.
  - Un `foreach` est une structure de `Qt` qui permet de faire des actions pour chaque élément d'une liste. D'où son nom.
  - Cela revient au même qu'une boucle itérative.

```
// T est un type défini.
```

```
T unElement;
```

```
QList<T> listElement;
```

```
foreach(unElement, listElement) {
```

```
// On prend chaque élément de listElement que l'on met
```

```
// l'un après l'autre dans la variable unElement.
```

```
// Les actions.
```

```
}
```

## 8.4) Faire un chargeur de plug in

- On se servira aussi de `QDir::absoluteFilePath(QString)` qui nous permet d'avoir le chemin absolu vers le fichier.

```
QList<MyPlugin *> m_LsPlugin;
    // On crée une liste de MyPlugin* qui contiendra nos plugins.
QDir plugDir = QDir(qApp->applicationDirPath());
    // Comme avant, on crée un QDir.
plugDir.cd("./Chemin");
    // On se déplace encore.

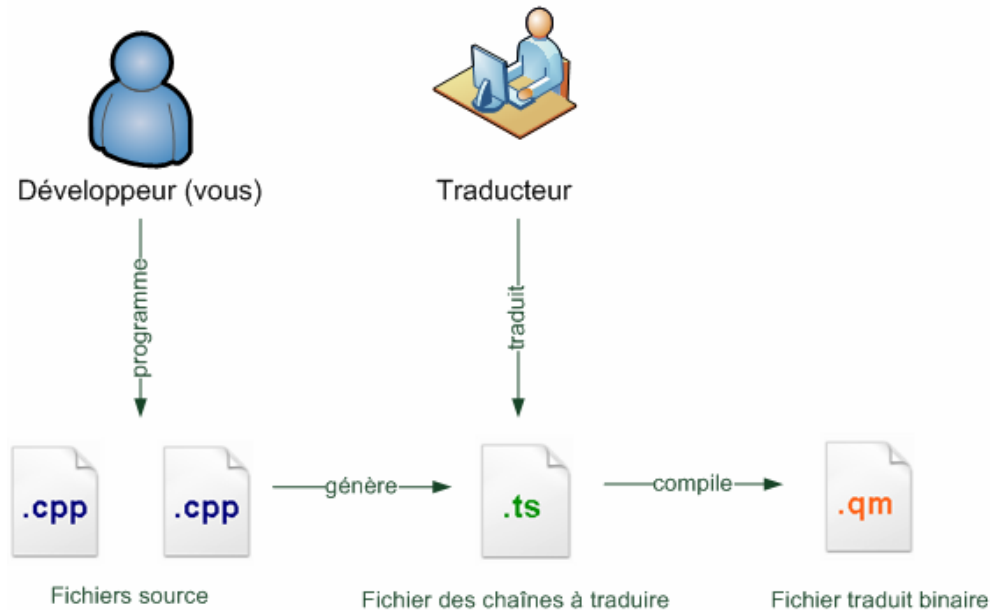
foreach(QString file, plugDir.entryList(QDir::Files)) {
    // Puis on utilise le foreach.
    QPluginLoader loader(plugDir.absoluteFilePath(file));
    // On fait ensuite la même chose que pour un seul plugin.
    if(QObject *plugin = loader.instance())
        MyPlugin* myPlugin = qobject_cast<MyPlugin *>(plugin);
        m_LsPlugin.push_back(myPlugin);
    // Vous pouvez maintenant les stocker ou directement les utiliser.
}
}
```

# 9) L'internationalisation

- Rappel sur Unicode.
- L'objet `QTranslator`.
- Mise en place de l'internationalisation dans l'application.
- L'application `QT Linguist`.

# 9.1) Etapes de traduction

- Qt suppose que les développeurs ne sont pas des traducteurs. Il suppose donc que ce sont 2 personnes différentes.
  - Tout a été fait pour que les traducteurs, même si ce ne sont pas des informaticiens, soient capables de traduire votre programme.



# 9.1) Etapes de traduction

1. il y a le développeur qui écrit normalement son programme, en rédigeant les messages dans le code source dans sa langue maternelle (le français).
2. on génère un fichier contenant les chaînes à traduire. Un programme livré avec Qt le fait automatiquement pour l'utilisateur. Ce fichier porte l'extension `.ts`, et est généralement de la forme : `nomduprogramme_langue.ts`.

Il faut connaître le symbole à 2 lettres de la langue de destination pour donner un nom correct au fichier `.ts`. Généralement c'est la même chose que les extensions des noms de domaine : fr (français), pl (polonais), ru (russe)...

3. Le traducteur récupère le ou les fichiers `.ts` à traduire (un par langue). Il les traduit via le programme Qt `Linguist`.
4. Une fois que le traducteur a fini, il retourne les fichiers `.ts` traduits au développeur, qui les "compile" en fichiers `.qm` binaires. La différence entre un `.ts` et un `.qm`, c'est un peu comme la différence entre un `.cpp` (la source) et un `.exe` (le programme binaire final).

Le `.qm` contenant les traductions au format binaire, Qt pourra le charger et le lire très rapidement lors de l'exécution du programme, ce qui fait qu'on ne sentira pas de ralentissement si on charge une version traduite du programme.

## 9.2) Préparer son code à la traduction

- Qt utilise exclusivement sa classe `QString` pour gérer les chaînes de caractères.
  - Cette classe, très complète, gère nativement l'Unicode.
- L'Unicode est une norme qui indique comment sont gérés les caractères à l'intérieur de l'ordinateur.
  - Elle permet à un ordinateur d'afficher sans problème tous types de caractères, en particulier les caractères étrangers.
  - `QString` n'a donc aucun problème pour gérer des alphabets cyrilliques ou arabes.

```
QString chaine = "Bonjour";  
// Bon : adapté pour la traduction  
char chaine[] = "Bonjour";  
// Mauvais : inadapté pour la traduction
```

## 9.2) Préparer son code à la traduction

- Utilisation basique
  - La méthode `tr()` permet d'indiquer qu'une chaîne devra être traduite. Par exemple, avant vous faisiez :

```
quitter = new QPushButton("&Quitter");
```

- Cela ne permettra pas de traduire le texte du bouton. En revanche, il faut d'abord appeler la méthode `tr()` :

```
quitter = new QPushButton(tr("&Quitter"));
```

- La méthode `tr()` est définie dans `QObject`.
  - C'est donc une méthode statique dont héritent toutes les classes de Qt, puisqu'elles dérivent de `QObject`.



# 9.3) Créer les fichiers de traduction

## .ts

- On souhaite que notre projet soit traduit dans les langues suivantes :
  - Anglais
  - Espagnol
- Nous devons générer 2 fichiers de traduction :
  - zeroclassgenerator\_en.ts pour l'anglais
  - zeroclassgenerator\_es.ts pour l'espagnol
- Il va falloir éditer le fichier .pro. Celui-ci se trouve dans le dossier de le projet et a normalement été généré automatiquement par Qt Creator.
- Ouvrez ce fichier (Projet1.pro) avec un éditeur de texte

```
#####  
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008  
#####
```

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

```
# Input  
HEADERS += FenCodeGenere.h FenPrincipale.h  
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp
```

Rajoutez à la fin une directive `TRANSLATIONS` en indiquant les noms des fichiers de traduction à générer. Ici, nous rajoutons un fichier pour la traduction anglaise, et un autre pour la traduction espagnole :

# 9.3) Créer les fichiers de traduction .ts

```
#####  
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008  
#####  
  
TEMPLATE = app  
TARGET =  
DEPENDPATH += .  
INCLUDEPATH += .  
  
# Input  
HEADERS += FenCodeGenere.h FenPrincipale.h  
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp  
TRANSLATIONS = Projet1_en.ts Projet1_es.ts
```

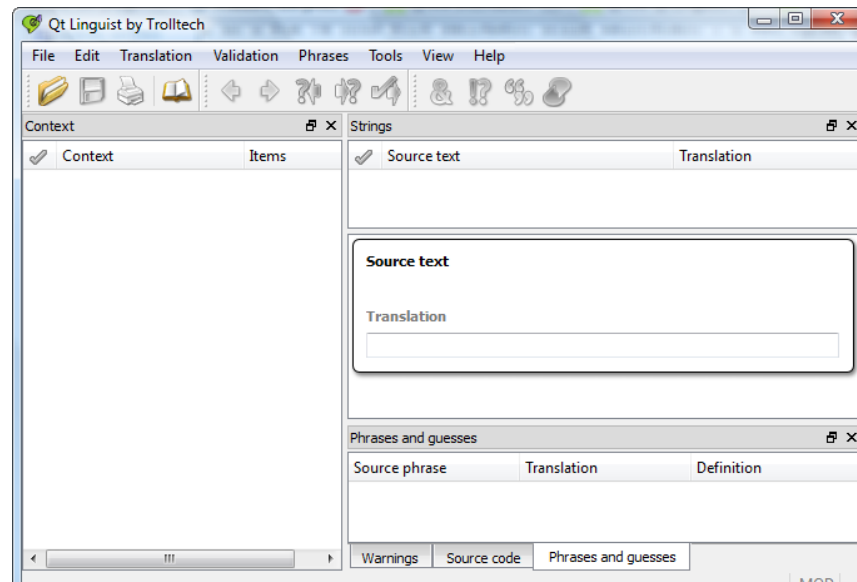
- faire appel à un programme en console de Qt qui permet de générer automatiquement les fichiers .ts.
- sous Windows, utilisez le raccourci Qt Command Prompt, allez dans le dossier de votre projet. Tapez :

```
lupdate NomDuProjet.pro
```

- `lupdate` est un programme qui va mettre à jour les fichiers de traduction .ts, ou les créer si ceux-ci n'existent pas.
- Ce programme est intelligent puisque, si vous l'exécutez une seconde fois, il ne mettra à jour que les chaînes qui ont changé. C'est très pratique, puisque cela permet d'avoir à faire traduire au traducteur seulement ce qui a changé par rapport à la version précédente de votre programme !
  - Il y a maintenant avoir 2 fichiers supplémentaires dans le dossier de votre projet :

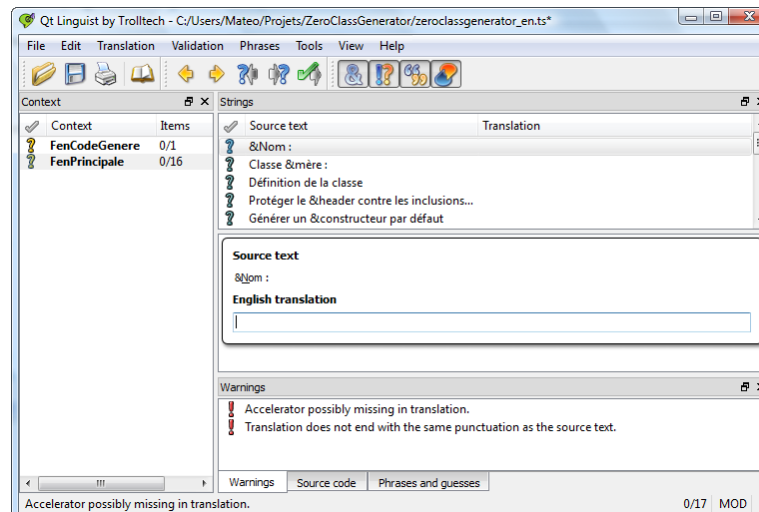
# 9.4) Traduire l'application sous Qt Linguist

- le traducteur a besoin de 2 choses :
  - Du fichier .ts à traduire
  - de Qt Linguist pour pouvoir le traduire !
- C'est une `QMainWindow`, avec une barre de menus, une barre d'outils, des docks, et un widget central
- Ouvrez un des fichiers .ts avec Qt Linguist, par exemple `Projet1_en.ts`. La fenêtre se remplit :



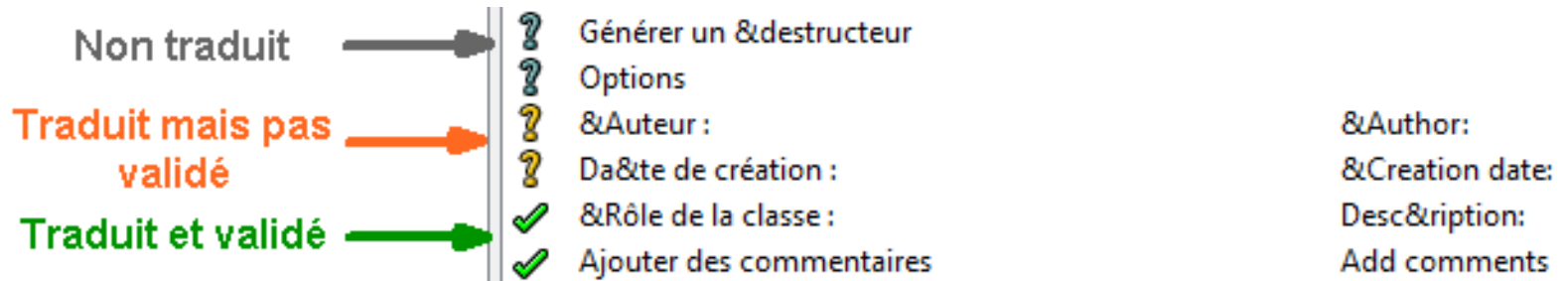
# 9.4) Traduire l'application sous Qt Linguist

- **Context**: affiche la liste des fichiers source qui contiennent des chaînes à traduire. Il s'agit des fenêtres FenCodeGenere et FenPrincipale. Le nombre de chaînes traduites est indiqué à droite.
- **Strings** : c'est la liste des chaînes à traduire pour le fichier sélectionné. Ces chaînes ont été extraites grâce à la présence de la méthode `tr()`.
- **Au milieu** : la version française de la chaîne, et on demande d'écrire la version anglaise. Notez que Qt a automatiquement détecté qu'il fallait traduire en anglais, grâce au nom du fichier qui contient "en".  
Si la chaîne à traduire peut être mise au pluriel, Qt Linguist vous demandera 2 traductions : une au singulier ("There is %n file") et une au pluriel ("There are %n files").
- **Warnings** : affiche des avertissements bien utiles, comme « il manque un & pour faire un raccourci », ou "La chaîne traduite ne se termine pas par le même signe de ponctuation" (ici un deux-points). Cette zone peut afficher aussi la chaîne à traduire dans son contexte du code source.



# 9.4) Traduire l'application sous Qt Linguist

- Lorsqu'il est sûr de sa traduction, il doit marquer la chaîne comme étant validée
  - en cliquant sur le petit "?" ou en faisant Ctrl + Entrée.
  - un symbole coché vert doit apparaître, et le dock context doit afficher que toutes les chaînes ont bien été traduites (16/16 par exemple).
- Voici les 3 états que peut avoir chaque message :
  - On procède donc en 2 temps : d'abord on traduit, puis ensuite on se relit et on valide. Lorsque toutes les chaînes sont validées (en vert), le traducteur vous rend le fichier .ts.
  - Il ne nous reste plus qu'une étape : compiler ce .ts en un .qm, et adapter notre programme pour qu'il charge automatiquement le programme dans la bonne langue.



# 9.5) Lancer l'application traduite

- Compiler le .ts en .qm
- Pour effectuer cette compilation, nous devons utiliser un autre programme de Qt : `lrelease`.
  - Ouvrez donc une console Qt (Qt Command Prompt), aller dans le dossier de votre projet, et tapez :

```
lrelease nomDuFichier.ts
```

- pour compiler le fichier .ts indiqué :

```
lrelease nomDuProjet.pro
```

- pour compiler tous les fichiers .ts du projet.

```
C:\Users\Laurent\Projets\Projet1>lrelease Projet1_en.ts
```

```
Updating 'Projet1_en.qm'...
```

```
Generated 17 translation(s) (17 finished and 0 unfinished)
```

- `lrelease` ne compile que les chaînes marquées comme terminées (celles qui ont le symbole vert dans Qt Linguist). Si certaines ne sont pas marquées comme terminées, elles ne seront pas compilées dans le .qm.
- Nous avons maintenant un fichier `Projet1_en.qm` dans le dossier de notre projet..

# 9.6) Charger un fichier de langue .qm dans l'application

- Le chargement d'un fichier de langue s'effectue au début de la fonction `main()`.

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();
    return app.exec();
}
```

- Juste après la création de l'objet de type `QApplication`, nous allons rajouter les lignes suivantes :

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QTranslator translator;
    translator.load("dictionary_en");
    app.installTranslator(&translator);

    FenPrincipale fenetre;
    fenetre.show();
    return app.exec();
}
```

# 9.6) Charger un fichier de langue .qm dans l'application

- Il y a moyen qu'elle s'adapte à la langue de l'utilisateur

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QString locale = QLocale::system().name().section('_', 0, 0);

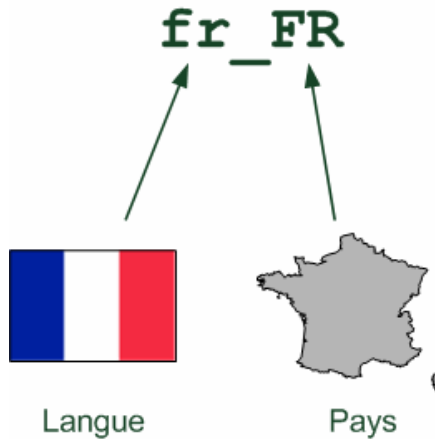
    QTranslator translator;
    translator.load(QString("dictionary_") + locale);
    app.installTranslator(&translator);

    FenPrincipale fenetre;
    fenetre.show();
    return app.exec();
}
```

- on veut récupérer le code à 2 lettres de la langue du PC de l'utilisateur. On utilise une méthode statique de QLocale pour récupérer des informations sur le système d'exploitation sur lequel le programme a été lancé.
- La méthode `QLocale::system().name()` renvoie un code ressemblant à ceci : "fr\_FR", où "fr" est la langue (français) et "FR" le pays (France).
- Si vous êtes québécois, vous aurez par exemple "fr\_CA" (français au Canada).



# 9.6) Charger un fichier de langue .qm dans l'application



- On veut juste récupérer les 2 premières lettres. On utilise la méthode `section()` pour couper la chaîne en deux autour de l'underscore "\_". Les 2 autres paramètres permettent d'indiquer qu'on veut le premier mot à gauche de l'underscore, à savoir le "fr".
  - Au final, la variable locale contiendra juste ce qu'on veut : la langue de l'utilisateur (par exemple "fr").
  - On combine cette variable avec le début du nom du fichier de traduction, comme ceci :

```
QString("dictionary_") + locale
```

Si locale vaut "fr", le fichier de traduction chargé sera "Projet\_fr".

Si locale vaut "en", le fichier de traduction chargé sera "Projet\_en".

# 10) En plus avec QT

- Développement d'applications multi plateformes.
- La gestion des threads (`QThread ...`).
- Les autres outils (`Qt Quick`, `Qt Assistant...`).
- `QML`.
- L'accès aux bases de données.
- Le développement mobile avec `QT Mobility`.

## 10.6) Qt Jambi

- Ce qui est faisable en C++ avec Qt l'est en Java avec Qt Jambi.
- Le point fort, c'est qu'il n'y a pas de pointeur à gérer et qu'il n'y a pas besoin de recompiler pour que ça fonctionne partout où il y a Qt Jambi et Java d'installés.
- Un projet en Java est composé d'au moins une classe contenant une fonction main.

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QPushButton;
...
public static void main(String[] args) {
    QApplication.initialize(args);

    QPushButton bouton = new QPushButton("Salut les Nouveaux ?");
    bouton.show();

    QApplication.exec();
}
```

# 10.6) Qt Jambi

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;

public class HelloWorld
{
    public static void main(String args[])
    {
        // Initialise l'application Qt
        QApplication.initialize(args);
        // Crée un bouton poussoir, bouton, dont le texte est Salut Nouveaux ?
        QPushButton bouton = new QPushButton("Salut les Nouveaux ?");
        // Fixe la taille du bouton à 120x40
        bouton.resize(120, 40);
        // Fixe la police de caractères en Times 18 Gras
        bouton.setFont(new QFont("Times", 18, QFont.Weight.Bold.value()));
        // Définit l'action du clic sur le bouton. (on quitte l'application)
        bouton.clicked.connect(QApplication.instance(), "quit()");
        // Fixe le titre de la fenêtre
        bouton.setWindowTitle("Hello World");
        // Affiche le bouton
        bouton.show();
        // Lance l'application //
        QApplication.exec();
    }
}
```

```
# Compiler
javac HelloWorld.java
# Lancer
java HelloWorld.java
```



# 10.1) Qt et les bases de donnée

```
CREATE TABLE Rubrique(  
    id_rubrique INT NOT NULL,  
    libelle_rubrique VARCHAR(50)  
;
```

```
CREATE TABLE Article(  
    id_article INT NOT NULL,  
    titre_article VARCHAR(50) NOT NULL,  
    nom_auteur VARCHAR(30) NOT NULL,  
    id_rubrique INT NOT NULL
```

```
CREATE TABLE Chapitre(  
    id_chapitre INT NOT NULL,  
    titre_chapitre VARCHAR(50) NOT NULL,  
    id_article INT NOT NULL
```

```
ALTER TABLE Rubrique  
ADD
```

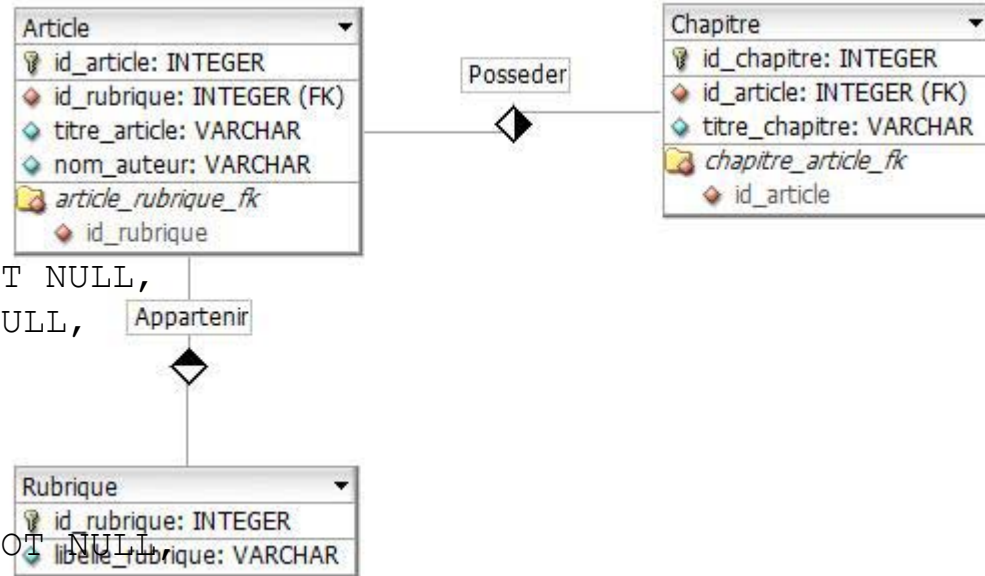
```
    CONSTRAINT PK_Rubrique PRIMARY KEY (id_rubrique);
```

```
ALTER TABLE Article  
ADD
```

```
    CONSTRAINT PK_Article PRIMARY KEY (id_article),  
    CONSTRAINT FK_Article_Rubrique FOREIGN KEY (id_rubrique)  
        REFERENCES Rubrique (id_rubrique) ON DELETE CASCADE;
```

```
ALTER TABLE Chapitre  
ADD
```

```
    CONSTRAINT PK_Chapitre PRIMARY KEY (id_chapitre),  
    CONSTRAINT FK_Chapitre_Article FOREIGN KEY (id_article)  
        REFERENCES Article (id_article) ON DELETE CASCADE;
```



# 10.1) Qt et les bases de donnée

- modifier le .pro pour que les classes d'accès aux données soient accessibles.

```
QT += sql
```

- 
- Les connexions s'utilisent au travers de la classe `QSqlDatabase`.
- Cette classe possède une méthode statique `QSqlDatabase::addDatabase(const QString)` renvoyant une instance de `QSqlDatabase` et reçoit en paramètre une chaîne de caractères correspondant au driver utilisé.
- la première connexion
  - Si aucun message d'erreur n'apparaît, alors la connexion s'est bien déroulée.

# 10.1) Qt et les bases de donnée

```
#include <QApplication>
#include <QSqlDatabase>
#include <QSqlError>
#include <QMessageBox>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    win_Form *window = new win_Form();

    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN"); // DSN que nous venons de créer.
    db.setUserName("alain.defrance");
    db.setPassword("plop");

    if(!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               db.lastError().text());
    }

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

# 10.2) Requête sans retour de données

```
#include <QApplication>
#include <QSqlDatabase>
#include <QMessageBox>
#include <QSqlError>
#include <QSqlQuery>
#include "win_main.hpp"
```

```
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    win_Form *window = new win_Form();
    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");
    if(!db.open())    {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               db.lastError().text());
    }
    QSqlQuery requeteur;
    requeteur.exec("INSERT INTO Rubrique(id_rubrique,
        libelle_rubrique) VALUES (1, 'nouvelle rubrique')");

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



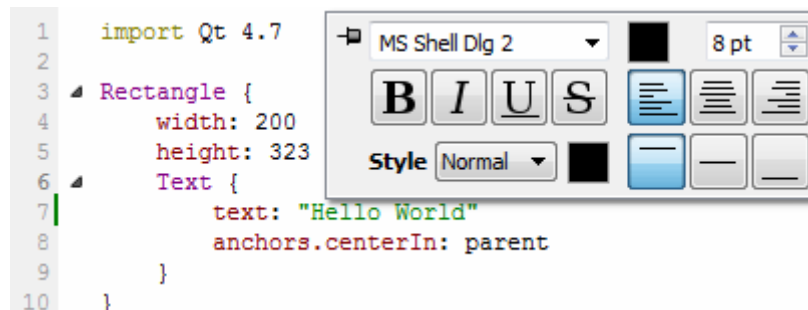
# 10.2) Requête sans retour de données

```
#include <QApplication>
#include <QSqlDatabase>
#include <QMessageBox>
#include <QSqlError>
#include <QSqlQuery>
#include "win_main.hpp"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    win_Form *window = new win_Form();
    QSqlDatabase db = QSqlDatabase::addDatabase("QODBC");
    db.setDatabaseName("monDSN");
    db.setPassword("plop");
    db.setUserName("alain.defrance");
    if(!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
db.lastError().text());
    }
    QSqlQuery requeteur;
    requeteur.exec("INSERT INTO Rubrique(id_rubrique,
libelle_rubrique) VALUES (1, 'nouvelle rubrique')");
    window->show();
    return app.exec();
}
```

## 10.3) Qt Quick

- Qt Creator 2.0.1 fournit un éditeur de texte complet pour QML et une intégration du qmlviewer.
- L'addition la plus visible dans l'éditeur de texte QML est une "barre d'outils Quick", qui permet de changer les propriétés diverses et variées des éléments QML à travers une interface utilisateur graphique.
- Par exemple, le changement des propriétés de police d'un élément `Text` est maintenant aussi simple que d'utiliser un éditeur de texte ordinaire.



## 10.4) Le Qt Quick Designer

- Qt Quick Designer est un éditeur visuel pour les fichiers QML (simples).
  - il est utilisable pour prototyper rapidement les choses.
  - par exemple prendre l'outil `gimp2qml` pour générer rapidement des fichiers `.qml` depuis GIMP et alors faire le layout final dans Qt Quick Designer.
- Le débogueur QML/JScript

# 10.5) Le développement et le déploiement d'applications C++/QML

- comment avoir une application QML sur votre téléphone sans un qmlviewer ?
- La manière la plus simple est d'écrire une application C++, via un nouvel assistant « d'applications QML ».
  - Ce nouveau projet va remplacer le format de fichier `.qmlproject`, qui nécessite la présence d'un `qmlviewer` lors de l'exécution.

