Librairie Qt

Table des matières

1. Introduction....3
1.1. Présentation....3
1.2. Installation de Qt....3
2. Mon premier programme....4
2.1. Création d'un projet Qt....4
2.2. Affichage d'un widget....8
2.3. Diffuser le programme....9
3. Personnaliser les widgets....9
3.1. Modifier les propriétés d'un widget....9
3.2. Qt et l'héritage....11
3.3. Widgets conteneurs....12
3.4. Hériter un widget....13
4. Les signaux et les slots....14
4.1. Le principe des signaux et slots....14
4.2. Connexion d'un signal à un slot simple....14
4.3. Paramètres dans les signaux et les slots....15
4.4. Créer ses propres signaux et slots....17
5. Les boîtes de dialogue usuelles....19
5.1. Afficher un message....19
5.2. Saisir une information....22
5.3. Sélectionner une police....23
5.4. Sélectionner une couleur....23
5.5. Sélection d'un fichier ou d'un dossier....24
5.5.1. Sélection d'un dossier....24
5.5.2. Ouverture d'un fichier....24
5.5.3. Enregistrement d'un fichier....25
6. Positionner les widgets....25
6.1. Le positionnement absolu....25
6.2. L'architecture des classes de layout....25
6.2.1. Les layouts horizontaux et verticaux....26
6.2.2. Le layout de grille....27
6.2.3. Le layout de formulaire....29
6.3. Combiner les layouts....30
7. Les principaux widgets....32
7.1. Les fenêtres....32
7.2. Les boutons....34
7.2.1. QpushButton....34
7.2.2. QcheckBox....34
7.2.2. QradioButton....34
7.3. Les afficheurs....35
7.3.1. QLabel....35
7.3.2. QprogressBar....36
7.4. Les champs....36

7.4.1. QLineEdit....36
7.4.2. QTextEdit....37
7.4.3. QSpinBox....37
7.4.4. QDoubleSpinBox....37
7.4.5. Qslider....37
7.4.6. QComboBox....37
7.5. Les conteneurs....38
8. La fenêtre principale....40
8.1. Présentation de QMainWindow....40
8.2. La zone centrale (SDI et MDI)....41
8.2.1. Définition de la zone centrale (type SDI)....41
8.2.2. Définition de la zone centrale (type MDI)....42
8.3. Les menus....43
8.4. La barre d'outils....44
9. Documentation Qt....45
9.1. Comprendre la documentation d'une classe....46
9.2. Lire et comprendre le prototype....48

Qt est une API orientée objet qui offre des composants d'interface graphique, d'accès aux données, de connexions réseaux,, d'analyse XML, etc. Qt est connu pour être la bibliothèque sur laquelle repose l'un des environnements de bureau les plus utilisés dans le monde GNU/Linux.

1. Introduction

1.1. Présentation

Chaque OS propose au moins une bibliothèque qui permet de créer des fenêtres. Le défaut de cette méthode est qu'en général, cette bibliothèque ne fonctionne que pour l'OS pour lequel elle a été créée. Il existe cependant des bibliothèques multiplate-forme qui permettent de s'affranchir de ce handicap. Voici quelques-unes des principales bibliothèques multiplate-forme :

- .NET (prononcez « Dot Net ») : développé par Microsoft pour succéder à la vieillissante API Win32. On l'utilise souvent en langage C#. On peut néanmoins utiliser .NET dans une multitude d'autres langages dont le C++.
- GTK+ : une des plus importantes bibliothèques utilisées sous Linux. Elle est portable, c'est-à-dire utilisable sous Linux, Mac OS X et Windows. GTK+ est utilisable en C mais il en existe une version C++ appelée GTKmm (on parle de wrapper ou encore de surcouche).
- wxWidgets : une bibliothèque objet très complète, comparable à Qt. wxWidgets n'est pas beaucoup plus compliquée que Qt. C'est la bibliothèque utilisée pour réaliser la GUI de l'IDE Code::Blocks.
- FLTK : contrairement à toutes les bibliothèques « poids lourds » précédentes, FLTK se veut légère. C'est une petite bibliothèque dédiée uniquement à la création d'interfaces graphiques multiplateforme.
- Qt très utilisée sous Linux, en particulier dans l'environnement de bureau KDE.

Qt est donc constituée d'un ensemble de bibliothèques, appelées « modules ». On peut y trouver entre autres ces fonctionnalités :

- Module GUI : c'est toute la partie création de fenêtres. Nous nous concentrerons surtout, dans ce cours, sur le module GUI.
- Module OpenGL : Qt peut ouvrir une fenêtre contenant de la 3D gérée par OpenGL.
- Module réseau : Qt fournit une batterie d'outils pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, un client Bittorent, un lecteur de flux RSS…
- Module XML : pour ceux qui connaissent le XML, c'est un moyen très pratique d'échanger des données à partir de fichiers structurés à l'aide de balises, comme le XHTML.
- Module SQL : permet d'accéder aux bases de données (MySQL, Oracle, PostgreSQL…).

1.2. Installation de Qt

Commencez par [télécharger Qt](#).

Bien qu'il soit possible de développer en C++ avec Qt en utilisant un IDE comme Code::Blocks, je vous recommande 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
- un éditeur de fenêtres, qui permet de dessiner facilement le contenu des interfaces à la souris

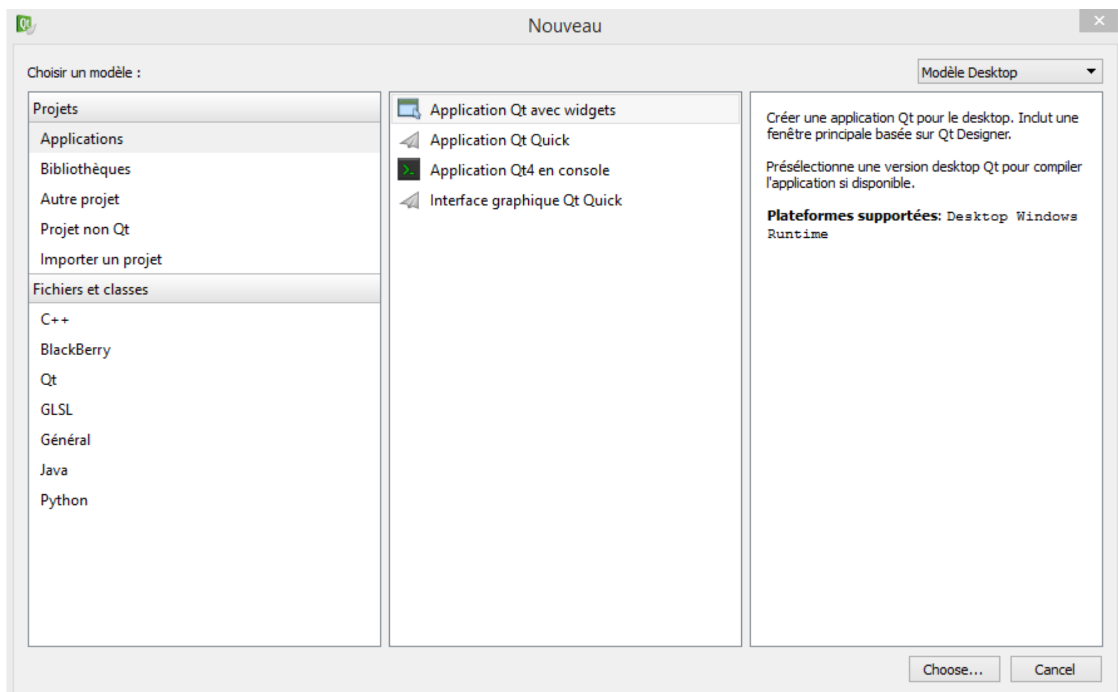
- une documentation indispensable

2. Mon premier programme

2.1. Création d'un projet Qt

Après avoir lancé Qtcreator, nous allons créer un nouveau projet en allant dans le menu Fichier > Nouveau fichier ou projet. On vous propose plusieurs choix selon le type de projet que vous souhaitez créer : application graphique pour ordinateur, application pour mobile, etc.

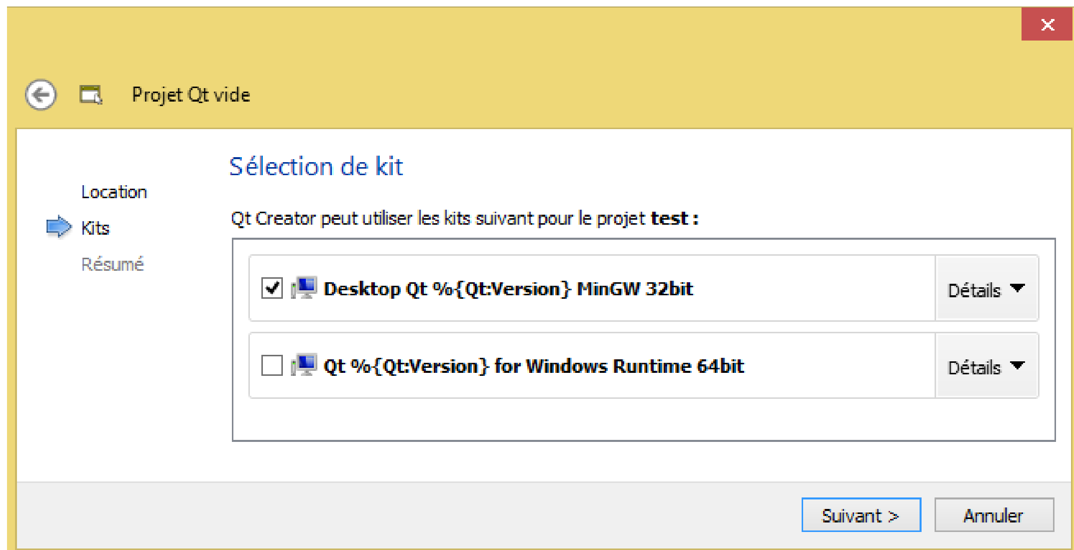
Choisissez les options « Autre projet », puis « Projet Qt vide » :



Un assistant s'ouvre alors pour vous demander le nom du projet et l'emplacement où vous souhaitez l'enregistrer :

Comme vous le voyez, je l'ai appelé « test », mais vous lui donner le nom que vous voulez.

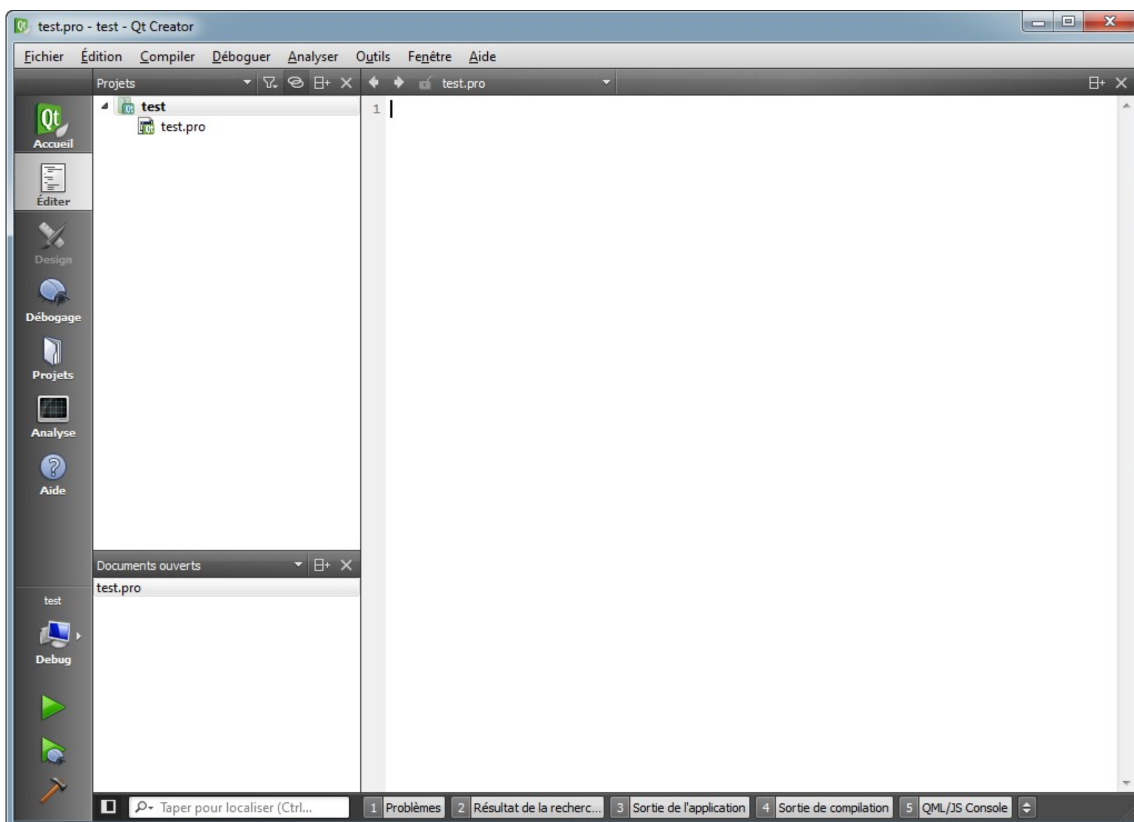
On vous demande ensuite quels "Kits" de compilation de Qt vous voulez utiliser. Pour le moment, contentez-vous de laisser les cases cochées comme elles sont :



La fenêtre suivante vous demande quelques informations dont vous avez peut-être moins l'habitude : vous pouvez lier votre projet à un gestionnaire de version.

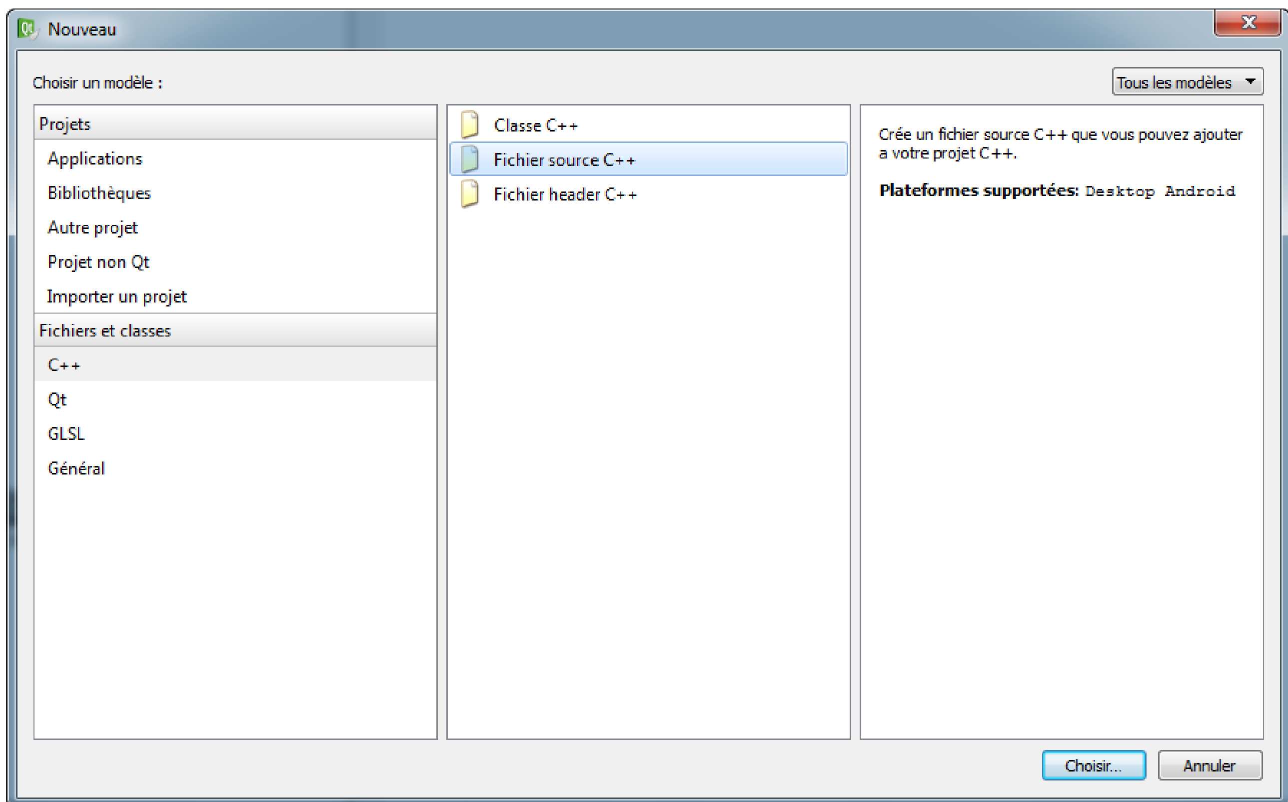
On peut associer le projet à un gestionnaire de version (comme SVN, Git). C'est un outil très utile, notamment si on travaille à plusieurs sur un code source…

Pour le moment, cliquez simplement sur « Suivant » :

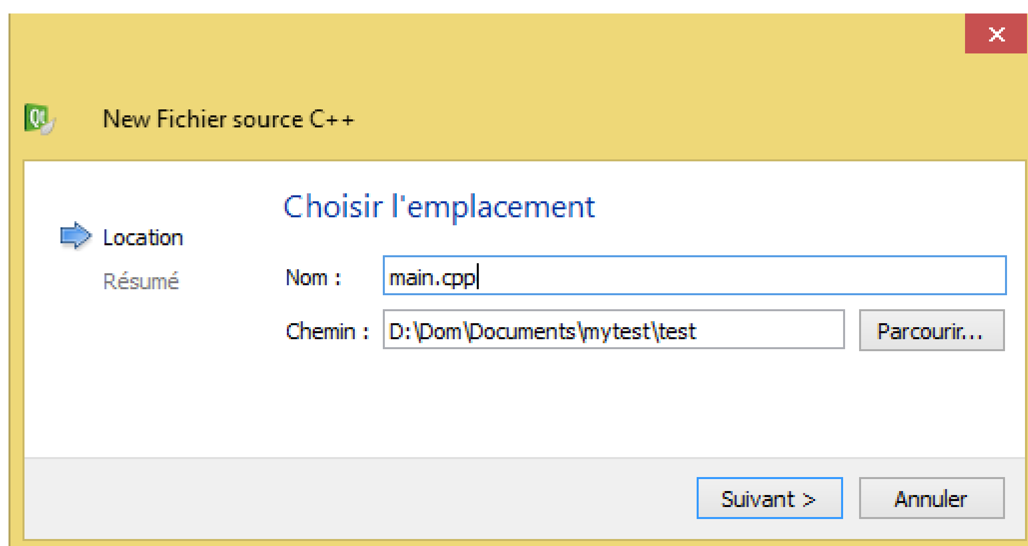


Le projet est constitué seulement d'un fichier .pro. Ce fichier, propre à Qt, sert à configurer le projet au moment de la compilation.

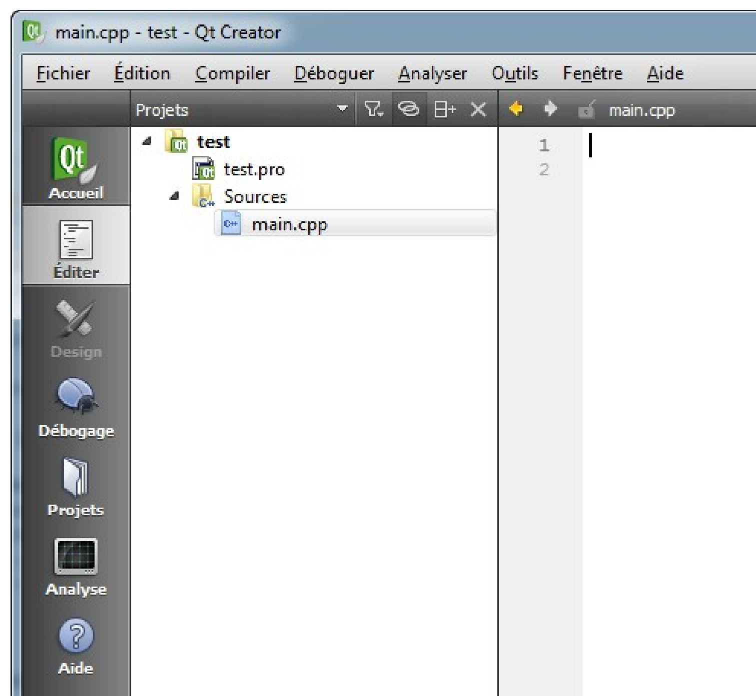
Il faut au moins un fichier main.cpp pour commencer à programmer : pour l'ajouter, retournez dans le menu Fichier > Nouveau fichier ou projet et sélectionnez cette fois C++ > Fichier source C++ pour ajouter un fichier .cpp :



On vous demande ensuite le nom du fichier à créer, indiquez main.cpp :



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



C'est dans ce fichier que nous écrirons nos premières lignes de code C++ utilisant Qt.

Saisissez le code suivant dans le fichier main.cpp :

```
#include <QApplication>

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

    return app.exec();
}
```

C'est le code minimal d'une application utilisant Qt. Analysons ce code :

```
#include <QApplication>
```

C'est le seul include dont vous avez besoin au départ. Il permet d'accéder à la classe QApplication, qui est la classe de base de tout programme Qt.

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

La première ligne du main() crée un nouvel objet de type QApplication. Cet objet est appelé app (mais vous pouvez l'appeler comme vous voulez). Le constructeur de QApplication exige que vous lui passiez les arguments du programme, c'est-à-dire les paramètres argc et argv que reçoit la fonction main.

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

Cette ligne fait 2 choses :

- Elle appelle la méthode exec de notre objet app. Cette méthode démarre le programme et lance donc l'affichage des fenêtres.
- Elle renvoie le résultat de app.exec() pour dire si le programme s'est bien déroulé ou pas. Le return provoque la fin de la fonction main, donc du programme.

2.2. Affichage d'un widget[1]

Dans la plupart des bibliothèques GUI[2], dont Qt fait partie, tous les éléments d'une fenêtre sont appelés des widgets : les boutons, les cases à cocher, les images…

Pour provoquer l'affichage d'une fenêtre, il suffit de demander à afficher n'importe quel widget. Ici par exemple, nous allons afficher un bouton :

```
#include <QApplication>
#include <QPushButton>     // permet de créer des objets QpushButton (des boutons)

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

    QPushButton bouton("La forme ?");  // constructeur
    bouton.show();         // permet d'afficher le bouton

    return app.exec();
}
```

Pour pouvoir compiler notre fenêtre, il faut d'abord effectuer une toute petite configuration de Qt. On va devoir lui dire en effet qu'on veut utiliser le module "QtWidgets" qui nous permet d'afficher des fenêtres.

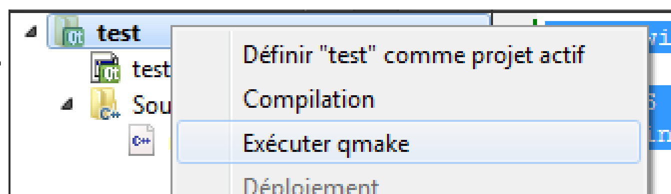
Ouvrez le fichier .pro de votre projet. Il devrait contenir la liste des fichiers de votre projet :

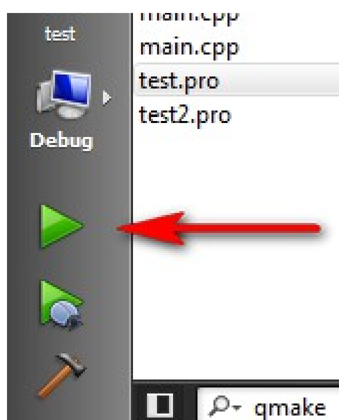
```
SOURCES += \

    main.cpp
```

Ajoutez-y une ligne QT += widgets pour demander à Qt de charger le module QtWidgets :

```
QT += widgets

SOURCES += \
    main.cpp
```

Ensuite, pour que Qt prenne en compte la modification de ce fichier .pro, il faut exécuter un petit utilitaire appelé qmake. Faites un clic droit sur le nom de votre projet et cliquez sur "Exécuter qmake" :

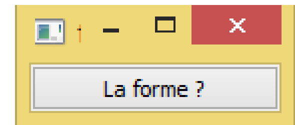




On peut maintenant compiler notre programme en cliquant sur "Exécuter" (la flèche verte).

1 composant d'interface graphique
2 Graphical User Interface

Le programme se lance alors…

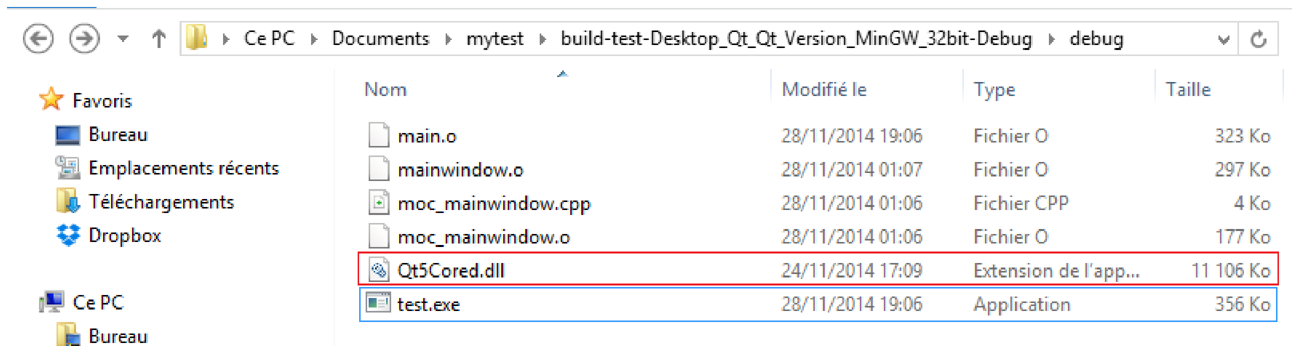


2.3. Diffuser le programme

L'exécutable qui a été généré ne pourra probablement pas lancer votre programme car il a besoin d'une série de fichiers DLL.

Quand vous exécutez votre programme depuis Qt Creator, la position des DLL est « connue », donc le programme se lance sans erreur.

Pour pouvoir lancer l'exécutable depuis l'explorateur, il faut placer les DLL qui manquent dans le même dossier que l'exécutable. À vous de les chercher sur votre disque.



3. Personnaliser les widgets

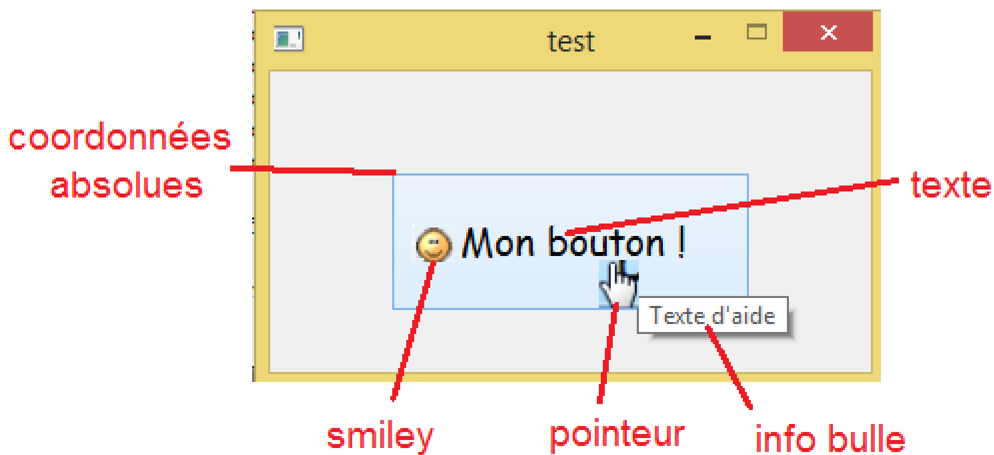
3.1. Modifier les propriétés d'un widget

Il existe pour chaque widget, y compris le bouton, un grand nombre de propriétés que l'on peut éditer. Pour chacune d'elle, la documentation de Qt nous indique de quel type est l'attribut, à quoi sert cet attribut et les accesseurs qui permettent de lire et modifier l'attribut.

Essayons donc de modifier le texte du bouton après sa création :

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

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

    QWidget fenetre;                            // création d'une fenêtre
    fenetre.setFixedSize(300, 150);             // taille de la fenêtre

    QPushButton bouton("La forme ?", &fenetre); // la fenêtre est le parent du bouton
    bouton.setText("Mon bouton !");             // le nouveau texte
    bouton.setToolTip("Texte d'aide");          // info bulle
    bouton.setFont(QFont("Comic Sans MS", 14)); // nouvelle police
    bouton.setCursor(Qt::PointingHandCursor);   // nouveau curseur
    bouton.setIcon(QIcon("smile.png"));         // ajout smiley devant texte
    bouton.setGeometry(60, 50, 180, 70);        // position absolue du bouton

    fenetre.show();                             // affichage
```

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



On créé une fenêtre à l'aide d'un objet de type Qwidget, puis on dimensionne le widget (la fenêtre) avec la méthode setFixedSize. La taille de la fenêtre étant fixée : on ne pourra pas la redimensionner.

On créé un bouton mais en indiquant un pointeur vers le widget parent (en l’occurrence la fenêtre) afin de faire contenir le bouton dans la fenêtre pour pouvoir le redimensionner.

La signature de la méthode setFont est : `void setFont(const QFont &)`

Le constructeur de QFont attend quatre paramètres. Voici son prototype :

```
QFont(const QString & family, int pointSize = -1, int weight = -1, bool italic = false)
```

Seul le premier argument est obligatoire : il s'agit du nom de la police à utiliser. Les autres, comme vous pouvez le voir, possèdent des valeurs par défaut. Nous ne sommes donc pas obligés de les indiquer. 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). Vous pouvez aussi utiliser la constante QFont::Bold qui correspond à une épaisseur de 75.
- italic : un booléen, pour indiquer si le texte doit être affiché en italique ou non.

Les attributs qui stockent du texte avec Qt est de type QString. En effet, Qt n'utilise pas la classe string standard du C++ mais sa propre version de la gestion des chaînes de caractères.

La méthode setIcon rajoute une icône au bouton et attend un objet de type QIcon.

Un QIcon se construit en donnant le nom du fichier image à charger :

- sous Windows, pour que cela fonctionne, votre icône smile.png doit se trouver dans le même dossier que l'exécutable (ou dans un sous-dossier si vous écrivez dossier/smile.png).
- sous Linux, il faut que votre icône soit dans votre répertoire HOME. Si vous voulez utiliser le chemin de votre application, écrivez : QIcon(QCoreApplication::applicationDirPath() + "/smile.png").

La méthode setGeometry permet, en plus de déplacer le widget, de lui donner des dimensions bien précise. La méthode setGeometry, qui prend 4 paramètres :

```
bouton.setGeometry(int abscisse, int ordonnee, int largeur, int hauteur)
```

À noter aussi, si vous voulez placer le bouton ailleurs dans la fenêtre, la méthode move :

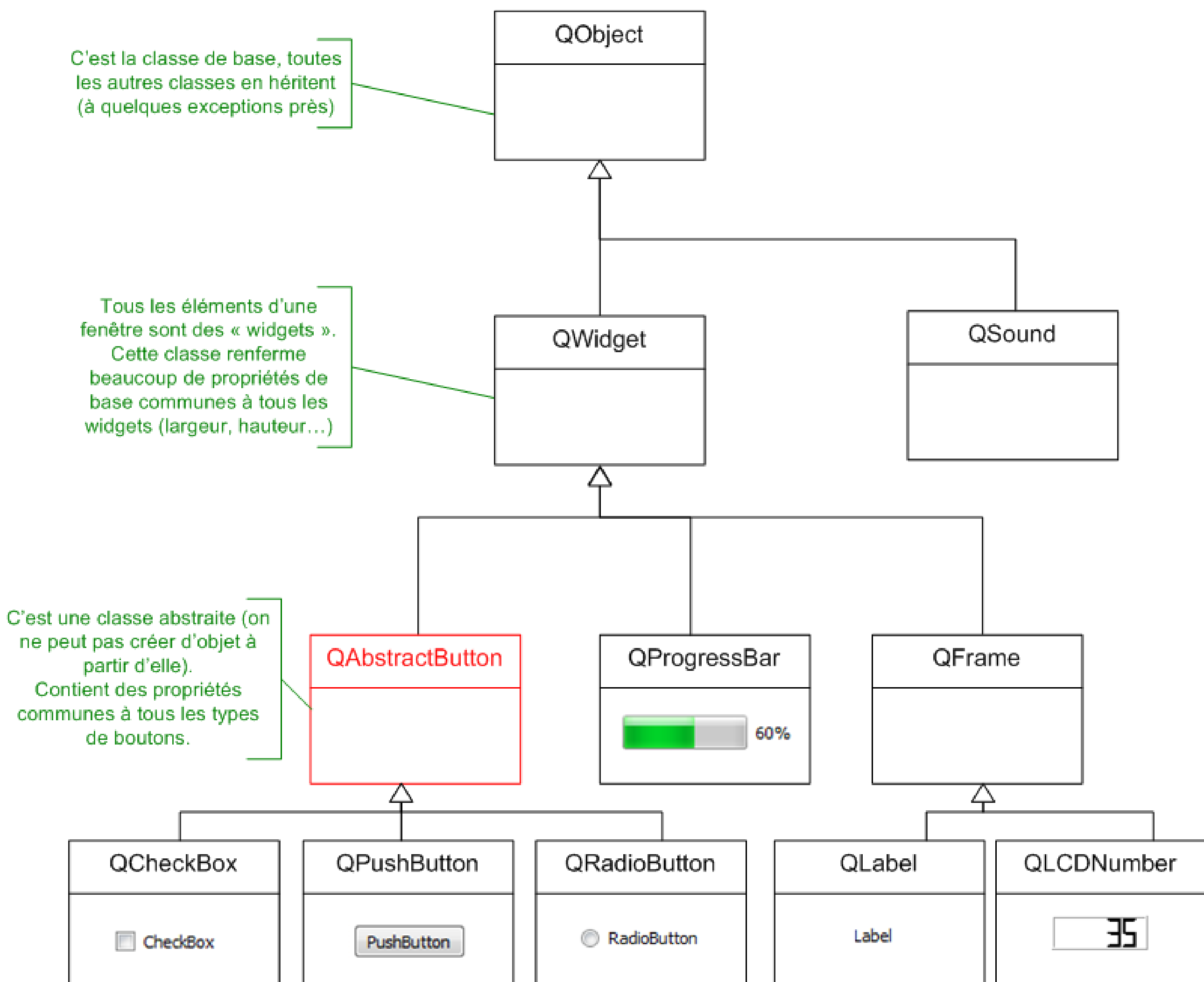
```
bouton.move(int abscisse, int ordonnee)
```

3.2. Qt et l'héritage

Le fait de pouvoir créer une classe de base, réutilisée par des sous-classes filles, qui ont elles-mêmes leurs propres sous-classes filles, cela donne à une bibliothèque comme Qt une puissance infinie.

QObject est la classe de base de tous les objets sous Qt.

Un schéma simplifié des héritages de Qt permet de mieux visualiser la hiérarchie des classes :



Dans une fenêtre, tout est considéré comme un widget (même la fenêtre est un widget). C'est pour cela qu'il existe une classe de base QWidget pour tous les widgets. Elle contient énormément de propriétés communes à tous les widgets, comme :

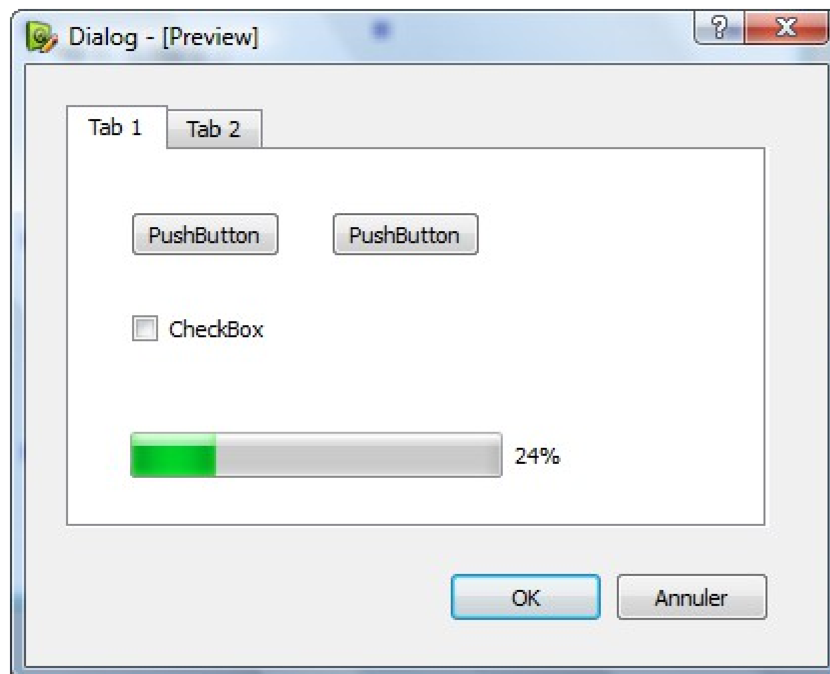
- la largeur

- 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.

Grâce à cette technique, il leur a suffit de définir une fois toutes les propriétés de base des widgets (largeur, hauteur…). Tous les widgets héritent de QWidget, donc ils possèdent toutes ces propriétés. Vous savez donc par exemple que vous pouvez retrouver la méthode setCursor dans la classe QProgressBar.

3.3. Widgets conteneurs

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



Sur cette capture, la fenêtre contient trois widgets :

- un bouton OK ;
- un bouton Annuler ;
- un conteneur avec des onglets.

Le conteneur avec des onglets est, comme son nom l'indique, un conteneur. Il contient à son tour des widgets :

- deux boutons ;
- une case à cocher (checkbox) ;
- une barre de progression.

Les widgets sont donc imbriqués les uns dans les autres suivant cette hiérarchie :

- QWidget (la fenêtre)
  - QPushButton
  - QpushButton
  - QTabWidget (le conteneur à onglets)
    - QPushButton
    - QPushButton
    - QCheckBox
    - QProgressBar

Tous les widgets possèdent un constructeur surchargé qui permet d'indiquer quel est le parent du widget que l'on crée. Il suffit de donner un pointeur pour que Qt sache « qui contient qui ». Quel que soit le widget, son constructeur accepte en dernier paramètre un pointeur vers un autre widget, pointeur qui indique quel est le parent.

3.4. Hériter un widget

Nous allons créer une nouvelle classe qui hérite de QWidget et représente notre fenêtre. Créer une classe pour gérer la fenêtre peut paraître un peu lourd au premier abord, mais c'est ainsi qu'on procède à chaque fois que l'on crée des GUI en POO. Cela nous donnera une plus grande souplesse par la suite :

- MaFenetre.h : contiendra la définition de la classe (Fichier/Nouveau Fichier/Fichier d'en-tête C++)
- MaFenetre.cpp : contiendra l'implémentation des méthodes (Fichier/Nouveau Fichier/Fichier source C++)

Voici le code du fichier MaFenetre.h :

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE      // évite que le header soit inclus plusieurs fois

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

class MaFenetre : public Qwidget  // On hérite de QWidget (IMPORTANT)
{
    public:
    MaFenetre();

    private:
    QPushButton *m_bouton;
};

#endif
```

Voici le code du fichier MaFenetre.cpp :

```
#include "MaFenetre.h"


MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);      // on définit la taille de la fenêtre de manière fixée

    // Construction du bouton
    // NB : le second paramètre du constructeur doit être un pointeur vers le widget parent
    m_bouton = new QPushButton("Salut !", 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);
}
```

Voici le code du fichier main.cpp :

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


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

    MaFenetre fenetre;           // on crée un objet de type MaFenetre
    fenetre.show();

    return app.exec();
}
```

Remarque : la destruction des widgets enfants est automatique. En effet, quand on supprime un widget parent (ici notre fenêtre), Qt supprime automatiquement tous les widgets qui se trouvent à l'intérieur (tous les widgets enfants).

4. Les signaux et les slots

4.1. Le principe des signaux et slots

Une application de type GUI réagit à partir d'évènements. Dans Qt, on parle de signaux et de slots :

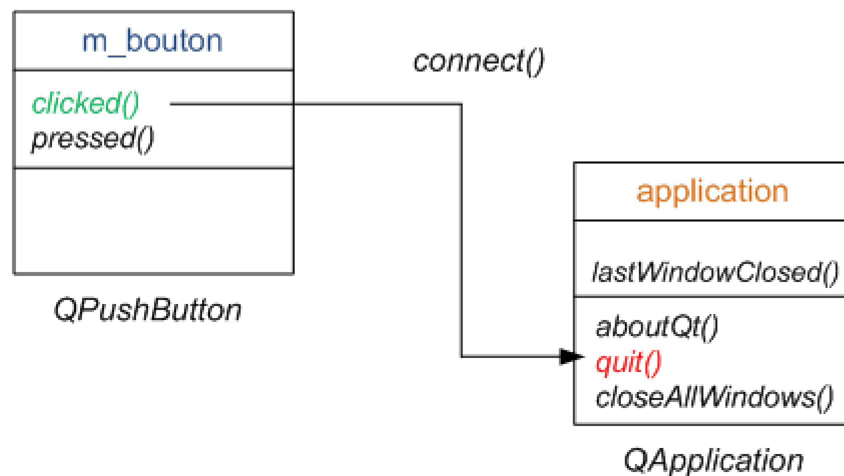
- Un signal : c'est un message envoyé par un widget lorsqu'un évènement se produit.
- Un slot : c'est la fonction qui est appelée lorsqu'un évènement s'est produit. On dit que le signal appelle le slot. Concrètement, un slot est une méthode d'une classe.

Exemple : le slot quit() de la classe QApplication 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.

4.2. Connexion d'un signal à un slot simple

Exemple : un clic sur un bouton doit provoquer l'arrêt de l'application.



Pour cela, nous devons utiliser une méthode statique de 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

Voici le nouveau code du fichier MaFenetre.cpp :

```
#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,            // pointeur vers le bouton qui va émettre le signal
       SIGNAL(clicked()),   // SIGNAL() est une macro du préprocesseur
       qApp,                // pointeur vers l'objet de type Qapplication créé dans le main
       SLOT(quit()));       // macro appelée lorsqu'on a cliqué sur le bouton
}
```

4.3. Paramètres dans les signaux et les slots

D'autres bibliothèques, comme wxWidgets, utilisent à la place de nombreuses macros et se servent du mécanisme un peu complexe et délicat des pointeurs de fonction (pour indiquer l'adresse de la fonction à appeler en mémoire).

Qt offre la possibilité aux signaux et aux slots de s'échanger des paramètres !

Pour illustrer ce propos, nous allons utiliser deux nouveaux widgets :

- QSlider : un curseur qui permet de définir une valeur
- QLCDNumber : un widget qui affiche un nombre

Voici le nouveau code du fichier MaFenetre.h :

```
#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
```

Et le fichier MaFenetre.cpp :

```
#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);

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

On a modifié le type d'afficheur LCD pour qu'il soit plus lisible (avec setSegmentStyle). Quant au slider, on a rajouté un paramètre pour qu'il apparaisse horizontalement (sinon il est vertical).

L'afficheur LCD change de valeur en fonction de la position du curseur du slider avec le code suivant :

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

On dispose du signal et du slot suivants :

- 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.

Il suffit d'indiquer le type du paramètre envoyé, ici un 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.

Le type des paramètres doit absolument correspondre : vous ne pouvez pas connecter un signal qui envoie (int, double) à un slot qui reçoit (int, int). C'est un des avantages du mécanisme des signaux et des slots : il respecte le type des paramètres.

En revanche, un signal peut envoyer plus de paramètres à un slot que celui-ci ne peut en recevoir. Dans ce cas, les paramètres supplémentaires seront ignorés.

4.4. 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.

Ex : nous allons faire en sorte qu'un QSlider contrôle la largeur de la fenêtre. Si le slider arrive à sa valeur maximale (600 dans notre cas), alors on émet un signal agrandissementMax et on quitte l'application.

Voici le code du fichier MaFenetre.h :

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLCDNumber>
#include <Qslider>

class MaFenetre : public QWidget
{
    Q_OBJECT    // macro à placer tout au début pour personnaliser un slot

    public:
    MaFenetre();

    public slots:
    void changerLargeur(int largeur);    // nouveau slot personnalisé

    signals:
    void agrandissementMax();

    private:
    QSlider *m_slider;
};

#endif
```

Et le code du fichier MaFenetre.cpp :

```
#include "MaFenetre.h"

void MaFenetre::changerLargeur(int largeur)
{
    // Le slot prend en paramètre la nouvelle largeur de la fenêtre
    setFixedSize(largeur, height());
    if (largeur == 600)
    {
        // pour envoyer un paramètre : emit monSignal(parametre1, parametre2, …);
        emit agrandissementMax();
    }
}

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

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setRange(200, 600);       // le slider est limité entre 200 et 600 px
    m_slider->setGeometry(10, 60, 150, 20);
```

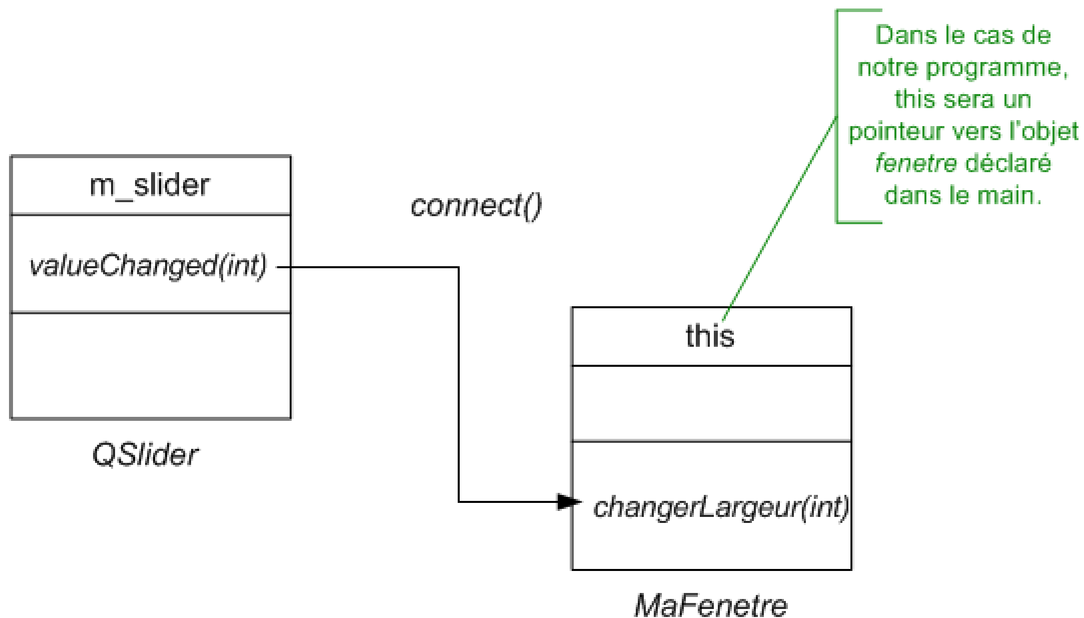
```
    QObject::connect(this, SIGNAL(agrandissementMax()), qApp, SLOT(quit()));
    QObject::connect(m_slider, SIGNAL(valueChanged(int)), this, SLOT(changerLargeur(int)));
}
```

Le signal valueChanged(int) du QSlider doit être connecté à un slot de la fenêtre qui aura pour rôle de modifier la largeur de la fenêtre.

Comme il n'existe pas de slot changerLargeur dans la classe QWidget, nous allons devoir le créer.

La connexion se fait entre le signal valueChanged(int) de notre QSlider et le slot changerLargeur(int) de notre classe MaFenetre.

Schématiquement, on a réalisé la connexion présentée à la figure suivante :



On a ajouté une section signals. 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. En effet, c'est Qt qui le fait pour nous. Si vous tentez d'implémenter un signal, vous aurez une erreur du genre « Multiple definition of… ».

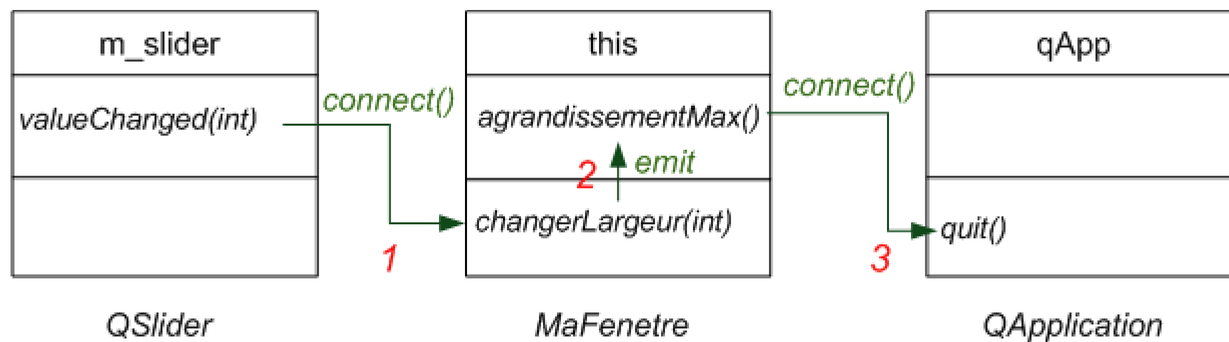
- Un signal peut passer un ou plusieurs paramètres (dans notre cas, il n'en envoie aucun).
- Un signal doit toujours renvoyer void.

Pour émettre un signal, on utilise le mot-clé emit de la bibliothèque Qt qui n'existe pas en C++.

Il ne reste plus qu'à connecter le nouveau signal à un slot dans le constructeur de MaFenetre :

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

Le schéma des signaux qu'on vient d'émettre et de connecter est présenté en figure suivante :



Dans l'ordre, voici ce qui s'est passé :

1. Le signal valueChanged du slider a appelé le slot changerLargeur de la fenêtre.
2. Le slot a fait ce qu'il avait à faire (changer la largeur de la fenêtre) et a vérifié que la fenêtre était arrivée à sa taille maximale. Lorsque cela a été le cas, le signal personnalisé agrandissementMax() a été émis.
3. Le signal agrandissementMax() de la fenêtre était connecté au slot quit() de l'application, ce qui a provoqué la fermeture du programme.

Remarque : il arrive parfois d'obtenir une erreur du style « erreur : undefined reference to `vtable for' » quand on utilise « signal » et « slot » via la méthode connect. Pour une raison ou une autre, le compilateur n'arrive pas à créer le ou les fichiers « .moc » nécessaire à ce mécanisme.

Pour y remédier :

1. Clic droit sur le projet : « Nettoyer le projet '…' »
2. Clic droit sur le projet : « Excécuter qmake »
3. Clic droit sur le projet : « Recompiler le projet '…' »

5. Les boîtes de dialogue usuelles

5.1. Afficher un message

Les boîtes de dialogue « afficher un message » sont contrôlées par la classe QMessageBox. Nous allons créer dans la boîte de dialogue un bouton qui appelle le slot personnalisé ouvrirDialogue(). C'est dans ce slot que nous nous chargerons d'ouvrir une boîte de dialogue.

```
//----- MaFenetre.h

#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QmessageBox>  // à inclure pour déclarer la classe  QMessageBox

class MaFenetre : public QWidget
{
    Q_OBJECT

    public:
    MaFenetre();
```

```
    public slots:
    void ouvrirDialogue();

    private:
    QPushButton *m_boutonDialogue;
};

#endif

//----- MaFenetre.cpp

#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()
{
    // Vous insérerez ici le code d'ouverture des boîtes de dialogue
    QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et bienvenue !");
}
```

La classe QMessageBox permet de créer des objets de type QmessageBox, mais on utilise majoritairement ses méthodes statiques, pour des raisons de simplicité.

La méthode statique information() permet d'ouvrir une boîte de dialogue constituée d'une icône « information ».

```
StandardButton information (
      QWidget * parent,
      const QString & title,
      const QString & text,
      StandardButtons buttons = Ok,
      StandardButton defaultButton= NoButton );
```

Seuls les trois premiers paramètres sont obligatoires, les autres ayant des valeurs par défaut :

- parent : un pointeur vers la fenêtre parente (qui doit être de type QWidget ou hériter de QWidget). Vous pouvez envoyer NULL en paramètre si vous ne voulez pas que la boîte de dialogue ait une fenêtre parente.
- 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.

L'appel de la méthode statique se fait comme celui d'une fonction classique, à la différence près qu'il faut mettre en préfixe le nom de la classe dans laquelle elle est définie (QMessageBox).

NB : lorsque la boîte de dialogue est ouverte, on ne peut plus accéder à sa fenêtre parente en arrière-plan. On dit que la boîte de dialogue est une fenêtre modale : c'est une fenêtre qui « bloque » temporairement son parent en attente d'une réponse de l'utilisateur.

Remarque : il est possible d'enrichir le message à l'aide de balises (X)HTML.

Il existe trois autre méthode qui s'utilise de la même manière que QMessageBox::information dont l'icône change :

```
// Boîte de dialogue attention
QMessageBox::warning(this, "Titre de la fenêtre", "Fichier introuvable !");
// Boîte de dialogue erreur critique
QMessageBox::critical(this, "Titre de la fenêtre", "Fichier corrompu !");
// Boîte de dialogue question
QMessageBox::question(this, "Titre de la fenêtre", "Effacer ce fichier ?");
```

Pour personnaliser les boutons de la boîte de dialogue, il faut utiliser le quatrième paramètre de la méthode statique. Ce paramètre accepte une combinaison de valeurs prédéfinies, séparées par un OR (la barre verticale |). On appelle cela des flags.

La liste des flags disponibles est donnée par la documentation :

- boutons « Oui » : QMessageBox::Yes
- boutons « Non » : QMessageBox::No
- boutons « Annuler » : QMessageBox::Cancel

Pour traduire le texte des boutons de la boîte de dialogue, il faut utiliser l'objet QTranslator :

```
// main.cpp

#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();
}
```

Les méthodes statiques que nous venons de voir renvoient un entier (int). On peut tester facilement la signification de ce nombre à l'aide des valeurs prédéfinies par Q :

```
void MaFenetre::ouvrirDialogue()
{
    int reponse = QMessageBox::question(this, "Avertissement", "Voulez-vous effacer ce 
fichier ?", QMessageBox ::Yes | QMessageBox::No);

    if (reponse == QMessageBox::Yes)
    {
        QMessageBox::information(this, "Confirmation", "Demande enregistrée.");
    }
    else if (reponse == QMessageBox::No)
    {
        QMessageBox::critical(this, "Attention", "Le traitement sera interrompu !");
    }
}
```

5.2. Saisir une information

Pour saisir une information, ou faire un choix parmi une liste, les boîtes de dialogue de saisie sont idéales. Elles sont gérées par la classe QinputDialog. Les boîtes de dialogue « saisir une information » peuvent être de quatre types :

- saisir un texte : QInputDialog::getText() ;
- saisir un entier : QInputDialog::getInteger() ;
- saisir un nombre décimal : QInputDialog::getDouble() ;
- choisir un élément parmi une liste : QInputDialog::getItem().

La méthode statique QInputDialog::getText() a pour prototype :

```
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 );
```

Les paramètres signifient, dans l'ordre :

- 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. Permet de dire si on veut que les lettres s'affichent quand on tape, ou si elles doivent être remplacées par des astérisques (pour les mots de passe) ou si aucune lettre ne doit s'afficher. Par défaut, les lettres s'affichent normalement (QLineEdit::Normal).
- text : texte par défaut dans la zone de saisie.
- ok : 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.

Exemple :

```
void MaFenetre::ouvrirDialogue()
{
    bool      ok     = false;     //  vérifie si le bouton OK a été cliqué
    Qstring   pseudo = QInputDialog::getText(this, "Pseudo", "Entrez votre pseudo",
QLineEdit::Normal, QString(), &ok);

    if ( ok && !pseudo.isEmpty() )      //  isEmpty() vérifie si la saisie n'est pas vide
    {
        QMessageBox::information(this, "Pseudo", "Bonjour " + pseudo + ", ça va ?");
    }
    else
    {
        QMessageBox::critical(this, "Pseudo", "Pseudo non renseigné.");
    }
}
```

Si un pseudo a été saisi et que l'utilisateur a cliqué sur OK, alors une boîte de dialogue lui souhaite la bienvenue. Sinon, une erreur est affichée.

5.3. Sélectionner une police

La boîte de dialogue de sélection de police est gérée par la classe QFontDialog. Celle-ci propose en gros une seule méthode statique surchargée :

```
QFont getFont (
       bool * ok,
       const QFont & initial,
       QWidget * parent,
       const QString & caption )
```

- le pointeur vers un booléen ok, permet de savoir si l'utilisateur a cliqué sur OK ou a annulé.
- on peut spécifier une police par défaut (initial), il faudra envoyer un objet de type QFont.
- la chaîne caption correspond au message qui sera affiché en haut de la fenêtre.
- la méthode renvoie un objet de type QFont correspondant à la police qui a été choisie.

Exemple : la police sélectionnée sera immédiatement appliquée au texte d'un bouton, par l'intermédiaire de la méthode setFont().

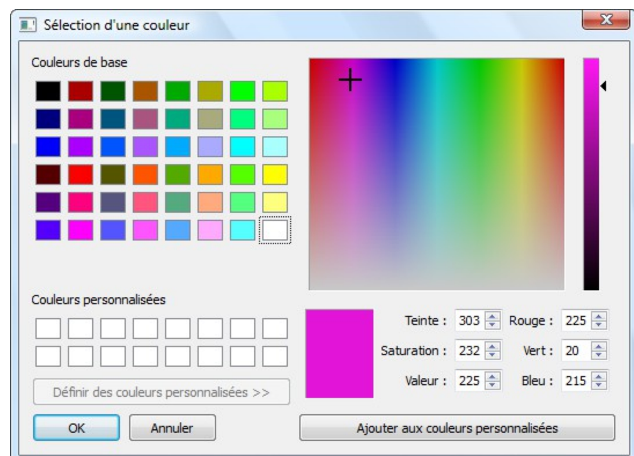
```
void MaFenetre::ouvrirDialogue()
{
    bool     ok     = false;

    Qfont    police = QFontDialog::getFont(&ok, m_boutonDialogue->font(), this,
"Choisissez une police");

    if ( ok )
    {
        m_boutonDialogue->setFont(police);
    }
}
```

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

On teste si l'utilisateur a bien validé la fenêtre et, si c'est le cas, on applique au bouton la police qui vient d'être choisie.

5.4. Sélectionner une couleur

Dans la même veine que la sélection de police, nous connaissons probablement tous la boîte de dialogue « Sélection de couleur » :



Utilisez la classe QColorDialoget et sa méthode statique getColor().

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

Elle renvoie un objet de type QColor. Vous pouvez préciser une couleur par défaut, en envoyant un objet de type QColor ou en utilisant une des constantes prédéfinies.

Pour appliquer une couleur au bouton, il fat utiliser la méthode setPalette qui sert à indiquer une palette de couleurs. Il faudra vous renseigner dans ce cas sur la classe QPalette.

```
void MaFenetre::ouvrirDialogue()
{
    QColor couleur = QColorDialog::getColor(Qt::white, this);

    QPalette palette;
    palette.setColor(QPalette::ButtonText, couleur);
    m_boutonDialogue->setPalette(palette);
}
```

Le code ouvre une boîte de dialogue de sélection de couleur, puis crée une palette où la couleur du texte correspond à la couleur qu'on vient de sélectionner, et enfin applique cette palette au bouton.

5.5. Sélection d'un fichier ou d'un dossier

La sélection de fichiers et de dossiers est gérée par la classe QFileDialog qui propose des méthodes statiques.

5.5.1. Sélection d'un dossier

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

La méthode renvoie un QString contenant le chemin complet vers le dossier demandé.

5.5.2. Ouverture d'un fichier

La boîte de dialogue « Ouverture d'un fichier » est gérée par getOpenFileName().

Sans paramètre particulier, la boîte de dialogue permet d'ouvrir n'importe quel fichier. Vous pouvez néanmoins créer un filtre (en dernier paramètre) pour afficher par exemple uniquement les images.

```
void MaFenetre::ouvrirDialogue()
{
    QString fichier = QfileDialog::getOpenFileName(
       this,
       "Ouvrir un fichier",
       Qstring(),   // nom du répertoire par défaut dans lequel l'utilisateur est placé
       "Images (*.png *.gif *.jpg *.jpeg)");

    QMessageBox::information(this, "Fichier", "Vous avez sélectionné :\n" + fichier);
}
```

Ce 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 :

Le troisième paramètre de getOpenFileName est le nom du répertoire par défaut dans lequel l'utilisateur est placé. La valeur par défaut (QString(), équivaut à écrire ""), donc la boîte de dialogue affiche par défaut le répertoire dans lequel est situé le programme.

Grâce au quatrième paramètre, seules les images de type PNG, GIF, JPG et JPEG s'afficheront.

Le principe de cette boîte de dialogue est de vous donner le chemin complet vers le fichier, mais pas d'ouvrir ce fichier. C'est à vous, ensuite, de faire les opérations nécessaires pour ouvrir le fichier et l'afficher dans votre programme.

5.5.3. Enregistrement d'un fichier

C'est le même principe que la méthode précédente, à la différence près que, pour l'enregistrement, la personne peut cette fois spécifier un nom de fichier qui n'existe pas. Le bouton « Ouvrir » est remplacé par « Enregistrer » :

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

6. Positionner les widgets

On distingue deux techniques différentes pour positionner des widgets :

- Le positionnement absolu : avec l'appel à la méthode setGeometry (ou move)... Ce positionnement est très précis car on place les widgets au pixel près.
- Le positionnement relatif : c'est le plus souple.

6.1. Le positionnement absolu

Nous allons créer une fenêtre avec un bouton placé aux coordonnées (70, 60) sur la fenêtre :

```
#include <QApplication>
#include <QpushButton>

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    Qwidget         fenetre;

    QPushButton bouton("Bonjour", &fenetre);
    bouton.move(70, 60);

    fenetre.show();

    return app.exec();
}
```

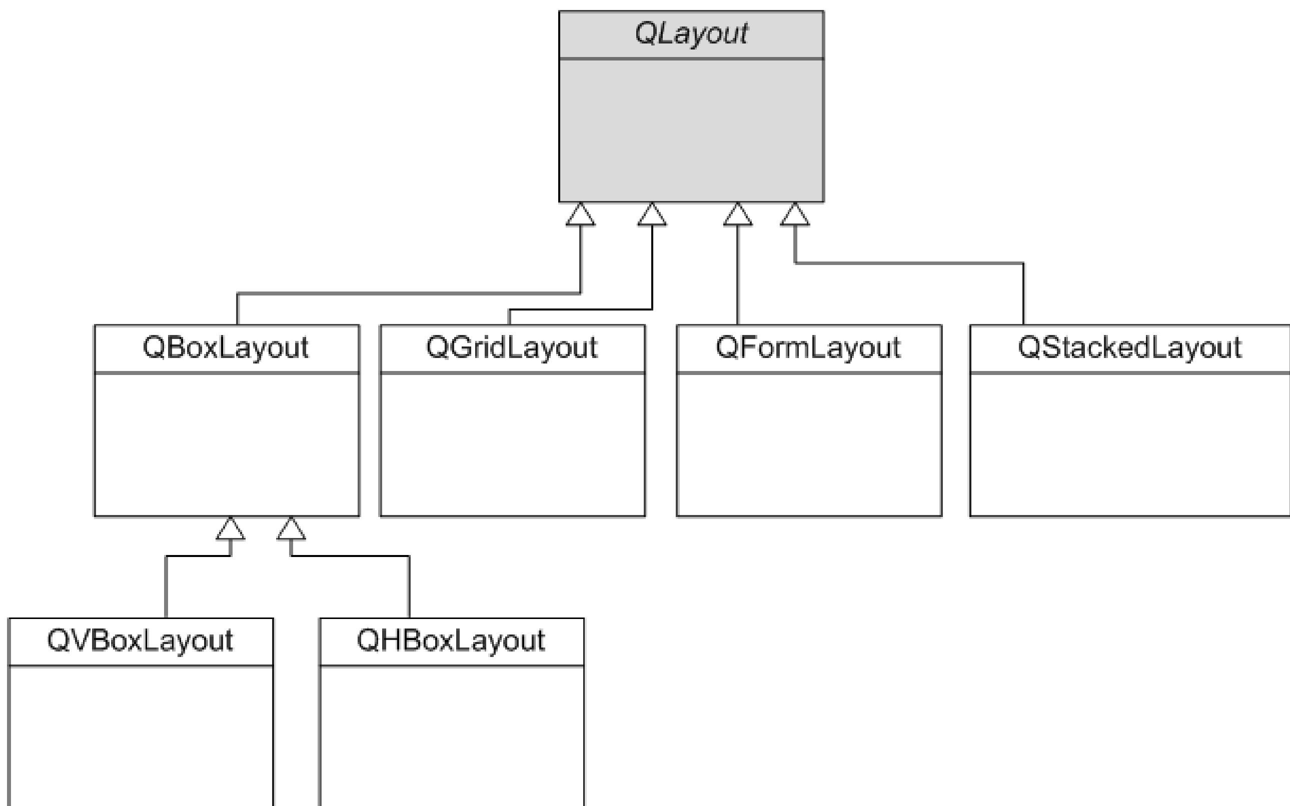
Le bouton est positionné de manière absolue à l'aide de l'instruction bouton.move(70, 60);

Si la fenêtre est redimensionnée, le bouton ne bouge pas. Le positionnement absolu sert parfois quand on a vraiment besoin de positionner au pixel près.

6.2. L'architecture des classes de layout

Le positionnement relatif, consiste à expliquer comment les widgets sont agencés les uns par rapport aux autres. Le positionnement relatif est géré dans Qt par des layouts : ce sont des conteneurs de widgets.

Il existe par exemple des classes gérant le positionnement horizontal et vertical des widgets ou encore le positionnement sous forme de grille.



Toutes les classes héritent de la classe de base Qlayout (classe abstraite).

On compte donc en gros les classes :

- QBoxLayout
- QHBoxLayout
- QVBoxLayout
- QGridLayout
- QFormLayout
- QStackedLayout

6.2.1. Les layouts horizontaux et verticaux

Nous allons travailler sur deux classes :

- QHBoxLayout ;
- QVBoxLayout.

QHBoxLayout et QVBoxLayout héritent de QBoxLayout.

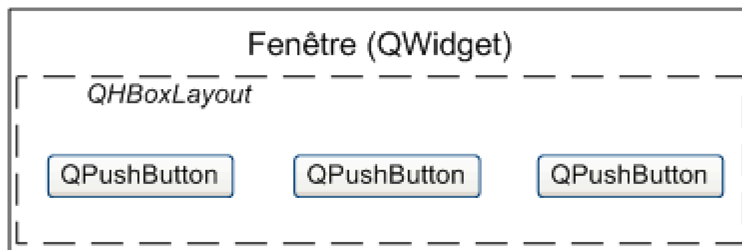
L'utilisation d'un layout se fait en 3 temps :

- On crée les widgets
- On crée le layout et on place les widgets dedans
- On dit à la fenêtre d'utiliser le layout qu'on a créé

Exemple :

```
#include <QApplication>
#include <QPushButton>
#include <QHBoxLayout>     // ne pas oublier !

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    Qwidget         fenetre;

    // création des widgets bouton
    QPushButton *bouton1 = new QPushButton("Un");
    QPushButton *bouton2 = new QPushButton("Deux");
    QPushButton *bouton3 = new QpushButton("Trois");

    // création d'un layout horizontal
    QHBoxLayout *layout = new QHBoxLayout;

    // placement des widgets dans le layout horizontal
    layout->addWidget(bouton1);
    layout->addWidget(bouton2);
    layout->addWidget(bouton3);

    //  placement du layout dans la fenêtre
    fenetre.setLayout(layout);

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

La fenêtre contient le layout qui contient les widgets. Le layout se charge d'organiser les widgets.



Le layout QHBoxLayout organise les widgets horizontalement. Il y en a un autre qui les organise verticalement : QVBoxLayout.

Pour utiliser un layout vertical, il suffit de remplacer QHBoxLayout par QVBoxLayout dans le code précédent.

Remarque : les widgets sont placés dans un layout, qui est lui-même placé dans la fenêtre. Lorsque la fenêtre est supprimée, tous les widgets contenus dans son layout sont supprimés par Qt.

6.2.2. Le layout de grille

QGridLayout offre une disposition en grille, comme un tableau avec des lignes et des colonnes :

Pour placer un widget en haut à gauche, il faudra le placer à la case de coordonnées (0, 0).

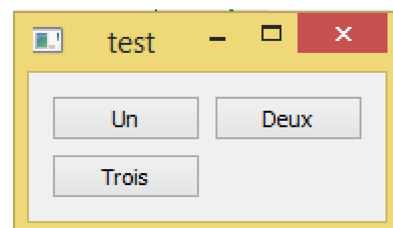
Pour en placer un autre en-dessous, il faudra utiliser les coordonnées (1, 0) et ainsi de suite.

0, 0	0, 1	0, 2	...
1, 0	1, 1	1, 2	...
2, 0	2, 1	2, 2	...
...	...	...	...

Utilisation basique de la grille :

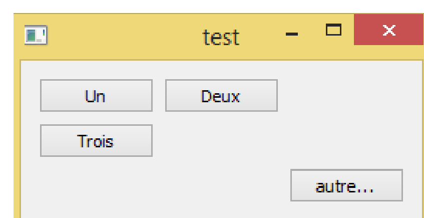
```
#include <QApplication>
#include <QPushButton>
#include <QGridLayout>  // ne pas oublier !

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

    // création des widgets bouton
    QPushButton *bouton1 = new QPushButton("Un");
    QPushButton *bouton2 = new QPushButton("Deux");
    QPushButton *bouton3 = new QpushButton("Trois");

    // création d'un layout grille
    QGridLayout *layout = new QgridLayout;


    // placement des widgets dans la grille
    layout->addWidget(bouton1, 0, 0);   // addWidget deux paramètres supplémentaires :
    layout->addWidget(bouton2, 0, 1);   // les coordonnées du widget sur la grille
    layout->addWidget(bouton3, 1, 0);

    //  placement du layout dans la fenêtre
    fenetre.setLayout(layout);

    fenetre.show();

    return app.exec();
}
```

Qt « sait » quel est le widget à mettre en bas à droite en fonction des coordonnées des autres widgets. Le widget qui a les coordonnées les plus élevées sera placé en bas à droite.



Pour décaler un bouton encore plus en bas à droite dans une nouvelle ligne et une nouvelle colonne :

```
QPushButton *bouton4 = new QPushButton("autre...");
layout->addWidget(bouton4, 2, 2);
```



Dans une disposition en grille, un widget peut occuper plusieurs cases à la fois. On parle de

spanning.

Pour ce faire, il faut appeler une version surchargée de addWidget qui accepte deux paramètres supplémentaires : le rowSpan et le columnSpan.

- rowSpan : nombre de lignes qu'occupe le widget (par défaut 1) ;
- columnSpan : nombre de colonnes qu'occupe le widget (par défaut 1).

Imaginons un widget placé en haut à gauche, aux coordonnées (0, 0). Si on lui donne un rowSpan de 2, il occupera alors l'espace indiqué à la figure ci-contre.

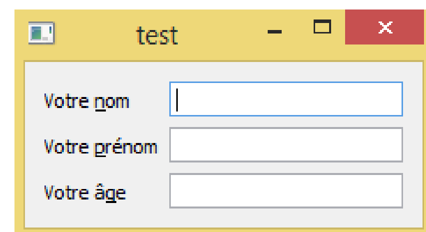
Si on lui donne un columnSpan de 3, il occupera l'espace indiqué sur la figure ci-contre.

columnSpan = 3			...
			...
rowSpan = 2			...
			...

L'espace pris par le widget au final dépend de la nature du widget (les boutons s'agrandissent en largeur mais pas en hauteur par exemple) et du nombre de widgets sur la grille.

```
layout->addWidget(bouton3, 1, 0, 1, 2);
```

Les deux derniers paramètres correspondent respectivement au rowSpan et au columnSpan. Le bouton va donc « occuper » 2 colonnes (figure ci-contre).



6.2.3. Le layout de formulaire

Le layout de formulaire QFormLayout est un layout spécialement conçu pour les fenêtres hébergeant des formulaires.

Votre prénom :	Anna
Votre nom :	Conda
Votre âge :	26

La disposition correspond à celle d'un QGridLayout à 2 colonnes et plusieurs lignes. En effet, le QFormLayout n'est en fait rien d'autre qu'une version spéciale du QGridLayout pour les formulaires, avec quelques particularités : il s'adapte aux habitudes des OS pour, dans certains cas, aligner les libellés à gauche, dans d'autres à droite, etc.

Au lieu d'utiliser une méthode addWidget, nous allons utiliser une méthode addRow qui prend deux paramètres :

- le texte du libellé ;
- un pointeur vers le champ du formulaire.

```
#include <QApplication>
#include <QLineEdit>
#include <QformLayout>

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    QWidget         fenetre;

    QLineEdit *nom    = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age    = new QlineEdit;

    QFormLayout *layout = new QformLayout;

    // le symbole « & » permet de définir des raccourcis clavier
    layout->addRow("Votre &nom", nom);
    layout->addRow("Votre &prénom", prenom);
    layout->addRow("Votre â&ge", age);

    fenetre.setLayout(layout);
    fenetre.show();

    return app.exec();
}
```

On peut aussi définir des raccourcis clavier pour accéder rapidement aux champs du formulaire. Pour ce faire, placez un symbole « & » devant la lettre du libellé que vous voulez transformer en raccourci.

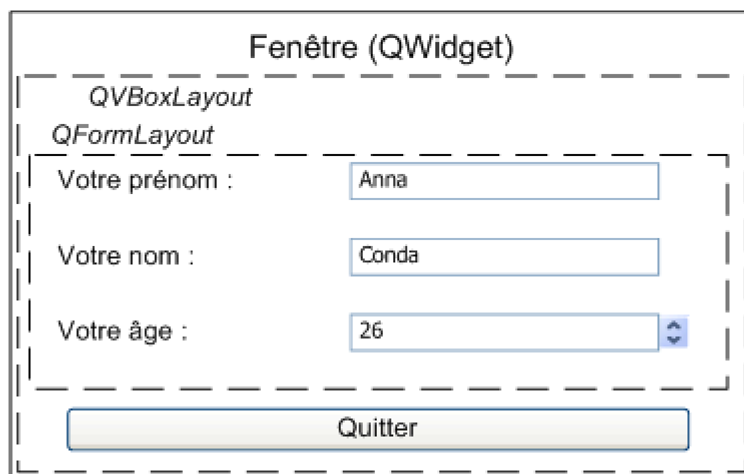
L'utilisation du raccourci dépend de votre système d'exploitation. Sous Windows, il faut appuyer sur la touche Alt puis la touche raccourci. Lorsque vous appuyez sur Alt, les lettres raccourcis apparaissent soulignées (figure suivante).

NB : pour afficher un symbole & dans un libellé, tapez-en deux : « && ».

6.3. Combiner les layouts

Prenons l'exemple du formulaire et ajoutons un bouton « Quitter ».

Il faut d'abord créer un layout vertical (QVBoxLayout) et placer à l'intérieur notre layout de formulaire puis notre bouton « Quitter ».



QVBoxLayout contient deux choses, dans l'ordre :

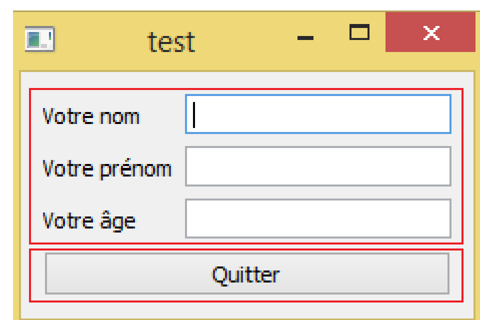
- Un QFormLayout (qui contient lui-même d'autres widgets) ;
- Un QPushButton.

Un layout peut donc contenir aussi bien des layouts que des widgets. Pour insérer un layout dans un autre, on utilise addLayout :

```
#include <QApplication>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QformLayout>

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    Qwidget         fenetre;

    // Création du layout de formulaire et de ses widgets
    QLineEdit *nom    = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age    = new QlineEdit;

    QFormLayout *formLayout = new QformLayout;

    formLayout->addRow("Votre &nom", nom);
    formLayout->addRow("Votre &prénom", prenom);
    formLayout->addRow("Votre â&ge", age);

    // Création du layout principal de la fenêtre (vertical)
    QVBoxLayout *layoutPrincipal = new QVBoxLayout;
    layoutPrincipal->addLayout(formLayout);          // Ajout du layout de formulaire

    QPushButton *boutonQuitter = new QpushButton("Quitter");
    layoutPrincipal->addWidget(boutonQuitter);       // Ajout du bouton

    // traitement du signal du bouton quitter
    QWidget::connect(boutonQuitter, SIGNAL(clicked()), &app, SLOT(quit()));

    fenetre.setLayout(layoutPrincipal);
    fenetre.show();

    return app.exec();
}
```

On crée d'abord les widgets et layouts « enfants » (le QFormLayout) ; ensuite on crée le layout principal (le QVBoxLayout) et on y insère le layout enfant créé.

On ne le voit pas, mais la fenêtre contient d'abord un QVBoxLayout, qui contient lui-même un layout de formulaire et un bouton (figure ci-contre).



7. Les principaux widgets

7.1. Les fenêtres

Avec Qt, tout élément de la fenêtre est appelé un widget. La fenêtre elle-même est considérée comme un widget. Toutefois, il y a deux classes de widgets à mettre en valeur :

- QmainWindow : c'est un widget spécial qui permet de créer la fenêtre principale de l'application. Une fenêtre principale peut contenir des menus, une barre d'outils, une barre d'état, etc.
- Qdialog : c'est une classe de base utilisée par toutes les classes de boîtes de dialogue. On peut aussi s'en servir directement pour ouvrir des boîtes de dialogue personnalisées.

La classe QDialog hérite de QWidget comme tout widget et y ajoute peu de choses, parmi lesquelles la gestion des fenêtres modales.

Les QWidget disposent de beaucoup de propriétés et de méthodes. Donc tous les widgets disposent de ces propriétés et méthodes. On peut découper les propriétés en deux catégories :

- celles qui valent pour tous les types de widgets et pour les fenêtres ;
- celles qui n'ont de sens que pour les fenêtres.

Voici quelques propriétés les plus intéressantes :

- cursor : curseur de la souris à afficher lors du survol du widget. La méthode setCursor attend que vous lui envoyiez un objet de type QCursor. Certains curseurs classiques (comme le sablier) sont prédéfinis dans une énumération.
- enabled : indique si le widget est activé, si on peut le modifier. Un widget désactivé est généralement grisé. Si vous appliquez setEnabled(false) à toute la fenêtre, c'est toute la fenêtre qui devient inutilisable.
- height : hauteur du widget.
- size : dimensions du widget. Vous devrez indiquer la largeur et la hauteur.
- visible : contrôle la visibilité du widget.
- width : largeur.

NB : pour modifier une de ces propriétés, préfixez la méthode par un set. Exemple :

Ces propriétés sont donc valables pour tous les widgets, y compris les fenêtres. Si vous appliquez un setWidth sur un bouton, cela modifiera la largeur du bouton. Si vous appliquez cela sur une fenêtre, c'est la largeur de la fenêtre qui sera modifiée.

Les propriétés utilisables uniquement sur les fenêtres commencent toutes par window.

Elles n'ont de sens que si elles sont appliquées aux fenêtres :

- windowFlags : une série d'options contrôlant le comportement de la fenêtre. Il faut consulter l'énumération Qt::WindowType pour connaître les différents types disponibles.

  Par exemple pour afficher une fenêtre de type « Outil » avec une petite croix et pas de possibilité d'agrandissement ou de réduction (figure suivante) :

  ```
  fenetre.setWindowFlags(Qt::Tool);
  ```

C'est par là aussi qu'on passe pour que la fenêtre reste par-dessus toutes les autres fenêtres du système (avec le flag Qt::WindowStaysOnTopHint).

- windowIcon: l'icône de la fenêtre. Il faut envoyer un objet de type QIcon, qui lui-même accepte un nom de fichier à charger situé dans le même dossier que l'application.
- windowTitle: le titre de la fenêtre, affiché en haut.

QDialog est un widget spécialement conçu pour générer des fenêtres de type « boîte de dialogue ».

Les QDialog sont rarement utilisées pour gérer la fenêtre principale. Pour cette fenêtre, on préférera utiliser QWidget ou carrément QMainWindow .

Les QDialog peuvent être de 2 types :

- modales : on ne peut pas accéder aux autres fenêtres de l'application lorsqu'elles sont ouvertes ;
- non modales : on peut toujours accéder aux autres fenêtres.

Par défaut, les QDialog sont modales.

Elles disposent en effet d'une méthode exec() qui ouvre la boîte de dialogue de manière modale.

Exemeple : nous allons ouvrir une fenêtre principale QWidget qui contiendra un bouton. Lorsqu'on cliquera sur ce bouton, il ouvrira une fenêtre secondaire de type Qdialog qui affichera une image.

```
#include <QApplication>
#include <QtWidgets>

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

    Qwidget         fenetre;
    // place le bouton dans la fenêtre principale
    QpushButton     *bouton = new QPushButton("Ouvrir la fenêtre", &fenetre);

    Qdialog         secondeFenetre (&fenetre);
    QVBoxLayout     *layout = new QVBoxLayout;
    // place QLabel dans la fenêtre secondaire
    Qlabel          *image  = new Qlabel(&secondeFenetre);

    // pcharge et affiche une image
    image->setPixmap(QPixmap("icone.png"));
    layout->addWidget(image);
    secondeFenetre.setLayout(layout);

    QWidget::connect(bouton, SIGNAL(clicked()), &secondeFenetre, SLOT(exec()));

    fenetre.show();

    return app.exec();
}
```

Au départ, la fenêtre principale s'affiche.

Si vous cliquez sur le bouton, la boîte de dialogue s'ouvre.

7.2. Les boutons

Il existe 3 types de widgets « boutons » :

- QpushButton : un bouton classique.
- QcheckBox : une case à cocher.
- QradioButton : un bouton « radio », pour un choix à faire parmi une liste.

Tous ces widgets héritent de QAbstractButton (classe abstraite) qui lui-même hérite de QWidget, qui finalement hérite de Qobject.

7.2.1. QpushButton

QPushButton est l'élément le plus classique qui contient peu de méthodes qui lui sont propres.

setEnabled permet d'activer/désactiver le bouton.

Un bouton émet un signal clicked() quand on l'active, pressed() quand on l'enfonce et released() quand on le relâche relâche.

7.2.2. QcheckBox

Une case à cocher QCheckBox est généralement associée à un texte de libellé. On définit le libellé de la case lors de l'appel du constructeur :

```
#include <QApplication>
#include <QWidget>
#include <QcheckBox>

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    Qwidget         fenetre;

    QCheckBox *checkbox = new QCheckBox("J'aime les frites", &fenetre);

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

La case à cocher émet le signal stateChanged(bool) lorsqu'on modifie son état. Le booléen en paramètre nous permet de savoir si la case est maintenant cochée ou décochée.

Pour vérifier si la case est cochée, appelez isChecked() qui renvoie un booléen.

7.2.2. QradioButton

Les boutons radio QradioButton qui ont le même widget parent sont mutuellement exclusifs. Si vous en cochez un, les autres seront automatiquement décochés.

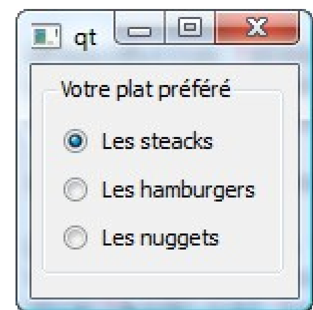
En général, on place les boutons radio dans une QgroupBox :

```
#include <QApplication>
#include <QtWidgets>

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

    QgroupBox       *groupbox = new QGroupBox("Votre plat préféré", &fenetre);
```

```
    QradioButton    *steacks    = new QRadioButton("Les steacks");
    QradioButton    *hamburgers = new QRadioButton("Les hamburgers");
    QradioButton    *nuggets    = new QRadioButton("Les nuggets");

    QVBoxLayout *vbox = new QVBoxLayout;

    vbox->addWidget(steacks);
    vbox->addWidget(hamburgers);
    vbox->addWidget(nuggets);

    steacks->setChecked(true);

    groupbox->setLayout(vbox);
    groupbox->move(5, 5);

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

Les boutons radio sont placés dans un layout qui est lui-même placé dans la groupbox, qui est elle-même placée dans la fenêtre.

Il n'y a pas de layout pour la fenêtre, ce qui fait que la taille initiale de la fenêtre est un peu petite.



7.3. Les afficheurs

Parmi les widgets afficheurs, on compte principalement :

- Qlabel : le plus important, un widget permettant d'afficher du texte ou une image ;
- QprogressBar : une barre de progression.

7.3.1. QLabel

QLabel hérite de QFrame, qui est un widget de base permettant d'afficher des bordures. Par défaut, un QLabel n'a pas de bordure.

Un QLabel peut afficher plusieurs types d'éléments :

- du texte (simple ou enrichi) ;
- une image.

Pour afficher un texte simple, on utilise setText() :

```
label->setText("Bonjour !");
```

Ou dès l'appel au constructeur :

```
QLabel *label = new QLabel("Bonjour !", &fenetre);
```

On peut écrire du code HTML dans le libellé pour lui appliquer une mise en forme (texte en gras, liens hypertexte, etc.).

Pour afficher une image, on utilise la méthode setPixmap qui attend un objet de type Qpixmap.

```
QLabel *label = new QLabel(&fenetre);
label->setPixmap(QPixmap("icone.png"));
```

NB : l'icône doit se trouver dans le même dossier que l'exécutable.

7.3.2. QprogressBar

Les barres de progression sont gérées par QProgressBar qui permet d'indiquer à l'utilisateur l'avancement des opérations. Voici quelques propriétés utiles de la barre de progression :

- maximum: la valeur maximale que peut prendre la barre de progression ;
- minimum: la valeur minimale que peut prendre la barre de progression ;
- value: la valeur actuelle de la barre de progression.

On utilisera donc setValue pour changer la valeur de la barre de progression. Par défaut les valeurs sont comprises entre 0 et 100%.

Une QProgressBar envoie un signal valueChanged() lorsque sa valeur a été modifiée.

7.4. Les champs

Voici les principaux widgets qui permettent de saisir des données :

- QLineEdit: champ de texte à une seule ligne ;
- QTextEdit: champ de texte à plusieurs lignes pouvant afficher du texte mis en forme ;
- QSpinBox: champ de texte adapté à la saisie de nombre entiers ;
- QDoubleSpinBox: champ de texte adapté à la saisie de nombre décimaux ;
- QSlider: curseur qui permet de sélectionner une valeur ;
- QComboBox: liste déroulante.

7.4.1. QLineEdit

Un QLineEdit est un champ de texte sur une seule ligne. Voici quelques propriétés à connaître :

- text : permet de récupérer/modifier le texte contenu dans le champ.
- Alignment : alignement du texte à l'intérieur.
- EchoMode : type d'affichage du texte. Il faudra utiliser l'énumération EchoMode pour indiquer le type d'affichage. Par défaut, les lettres saisies sont affichées mais on peut aussi faire en sorte que les lettres soient masquées, pour les mots de passe par exemple.

  ```
  lineEdit->setEchoMode(QLineEdit::Password);
  ```
- inputMask : permet de définir un masque de saisie, pour obliger l'utilisateur à fournir une chaîne répondant à des critères précis (par exemple, un numéro de téléphone ne doit pas contenir de lettres). Vous pouvez aussi jeter un coup d'œil aux validators qui sont un autre moyen de valider la saisie de l'utilisateur.
- maxLength : le nombre de caractères maximum qui peuvent être saisis.
- readOnly : le contenu du champ de texte ne peut être modifié. Cette propriété ressemble à enabled (définie dans QWidget) mais, avec readOnly, on peut quand même copier-coller le contenu du QLineEdit tandis qu'avec enabled, le champ est complètement grisé et on ne peut pas récupérer son contenu.

On note aussi plusieurs slots qui permettent de couper/copier/coller/vider/annuler le champ de texte.

Enfin, certains signaux comme returnPressed() (l'utilisateur a appuyé sur Entrée) ou textChanged() (l'utilisateur a modifié le texte) peuvent être utiles dans certains cas.

7.4.2. QTextEdit

Ce type de champ, similaire à QLineEdit, gère l'édition sur plusieurs lignes et autorise l'affichage de texte enrichi (HTML).

Les propriétés plainText et html permettent de récupérer et modifier le contenu respectivement sous forme de texte simple et sous forme de texte enrichi en HTML.

7.4.3. QSpinBox

Une QSpinBox est un champ de texte (type QLineEdit) qui permet d'entrer uniquement un nombre entier et qui dispose de petits boutons pour augmenter ou diminuer la valeur.

Voici quelques propriétés intéressantes :

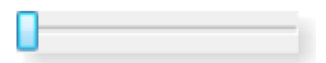
- accelerated: permet d'autoriser la spinbox a accélérer la modification du nombre si on appuie longtemps sur le bouton.
- minimum: valeur minimale que peut prendre la spinbox.
- maximum: valeur maximale que peut prendre la spinbox.
- singleStep: pas d'incrémentation (par défaut de 1). Si vous voulez que les boutons fassent varier la spinbox de 100 en 100, c'est cette propriété qu'il faut modifier.
- value: valeur contenue dans la spinbox.
- prefix: texte à afficher avant le nombre.
- suffix: texte à afficher après le nombre.

7.4.4. QDoubleSpinBox

Le QDoubleSpinBoxest très similaire au QSpinBox, à la différence près qu'il travaille sur des nombres décimaux.

7.4.5. Qslider

Un QSlider se présente sous la forme d'un curseur permettant de sélectionner une valeur numérique. Beaucoup de propriétés sont les mêmes que QspinBox.



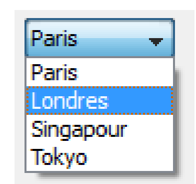
La propriété orientation qui permet de définir l'orientation du slider (verticale ou horizontale).

On connecte le signal valueChanged(int) au slot d'autres widgets pour répercuter la saisie de l'utilisateur.

7.4.6. QComboBox

Une QComboBox est une liste déroulante. On ajoute des valeurs à la liste déroulante avec la méthode addItem :

```
QComboBox *liste = new QComboBox(&fenetre);

liste->addItem("Paris");
liste->addItem("Londres");
liste->addItem("Singapour");
liste->addItem("Tokyo");
```



On dispose de propriétés permettant de contrôler le fonctionnement de la QComboBox :

- count: nombre d'éléments dans la liste déroulante.
- currentIndex: numéro d'indice de l'élément actuellement sélectionné. Les indices commencent à 0. Ainsi, si currentIndex renvoie 2, c'est que « Singapour » a été sélectionné dans l'exemple précédent.
- currentText: texte correspondant à l'élément sélectionné. Si on a sélectionné « Singapour », cette propriété contient donc « Singapour ».
- editable: indique si le widget autorise l'ajout de valeurs personnalisées ou non. Par défaut, l'ajout de nouvelles valeurs est interdit.

Si le widget est éditable, l'utilisateur pourra entrer de nouvelles valeurs dans la liste déroulante. Elle se comportera donc aussi comme un champ de texte. L'ajout d'une nouvelle valeur se fait en appuyant sur la touche « Entrée ». Les nouvelles valeurs sont placées par défaut à la fin de la liste

La QComboBox émet des signaux comme currentIndexChanged() qui indique qu'un nouvel élément a été sélectionné et highlighted() qui indique l'élément survolé par la souris (ces signaux peuvent envoyer un int pour donner l'indice de l'élément ou un QString pour le texte).

7.5. Les conteneurs

Certains widgets ont été créés spécialement pour pouvoir en contenir d'autres :

- QFrame : un widget pouvant avoir une bordure ;
- QGroupBox : un widget adapté à la gestion des groupes de cases à cocher et de boutons radio ;
- QTabWidget : un widget gérant plusieurs pages d'onglets.

QTabWidget propose une gestion de plusieurs pages de widgets, organisées sous forme d'onglets. Ce widget-conteneur ne peut contenir qu'un widget par page.

Ce conteneur doit être utilisé de la façon suivante :

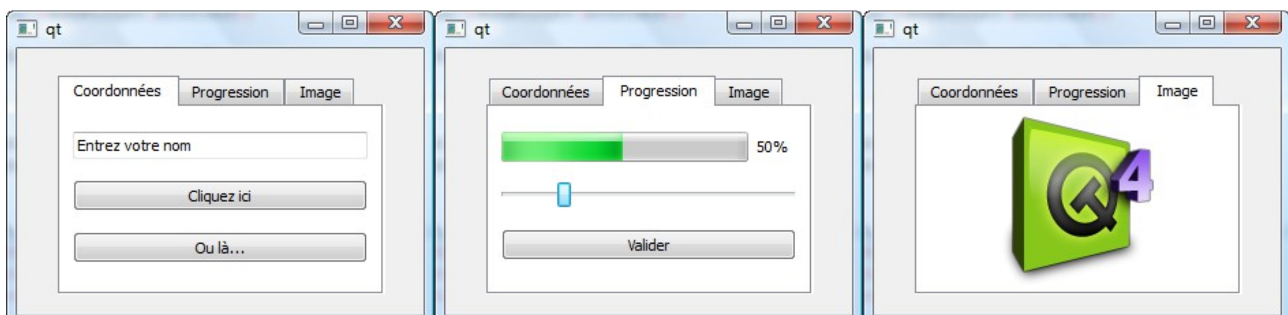
1. Créer un QTabWidget.
2. Créer un QWidget pour chacune des pages (chacun des onglets) du QTabWidget, sans leur indiquer de widget parent.
3. Placer des widgets enfants dans chacun de ces QWidget pour peupler le contenu de chaque page. Utiliser un layout pour positionner les widgets de préférence.
4. Appeler plusieurs fois addTab() pour créer les pages d'onglets en indiquant l'adresse du QWidget qui contient la page à chaque fois.

Exemple :

```
#include <QApplication>
#include <QtWidgets>

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

    // 1 : Créer le QTabWidget
    QTabWidget *onglets = new QTabWidget(&fenetre);
```

```
    onglets->setGeometry(30, 20, 240, 160);

    // 2 : Créer les pages, en utilisant un widget parent pour contenir chacune des pages
    QWidget *page1 = new QWidget;
    QWidget *page2 = new QWidget;
    QLabel *page3 = new Qlabel;   // QLabel hérite QWidget, il peut servir de page

    // 3 : Créer le contenu des pages de widgets
        //--- Page 1
        QLineEdit *lineEdit = new QLineEdit("Entrez votre nom");
        QPushButton *bouton1 = new QPushButton("Cliquez ici");
        QPushButton *bouton2 = new QPushButton("Ou là…");
        QVBoxLayout *vbox1 = new QVBoxLayout;
        vbox1->addWidget(lineEdit);
        vbox1->addWidget(bouton1);
        vbox1->addWidget(bouton2);
        page1->setLayout(vbox1);

        //--- Page 2
        QProgressBar *progress = new QProgressBar;
        progress->setValue(50);
        QSlider *slider = new QSlider(Qt::Horizontal);
        QPushButton *bouton3 = new QPushButton("Valider");
        QVBoxLayout *vbox2 = new QVBoxLayout;
        vbox2->addWidget(progress);
        vbox2->addWidget(slider);
        vbox2->addWidget(bouton3);
        page2->setLayout(vbox2);

        //--- Page 3 (on affiche une image, pas besoin de layout)
        page3->setPixmap(QPixmap("icone.png"));
        page3->setAlignment(Qt::AlignCenter);

    // 4 : ajouter les onglets au QTabWidget, en indiquant la page qu'ils contiennent
    onglets->addTab(page1, "Coordonnées");
    onglets->addTab(page2, "Progression");
    onglets->addTab(page3, "Image");

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

1. création de QtabWidget, positionné ici de manière absolue sur la fenêtre
2. création des pages pour chacun des onglets. Ces pages sont matérialisées par des QWidget.
3. création du contenu de chacune de ces pages à l'aide de layouts verticaux
4. ajout des onglets avec la méthode addTab(). On doit indiquer le libellé de l'onglet ainsi qu'un pointeur vers le « widget-page » de cette page.



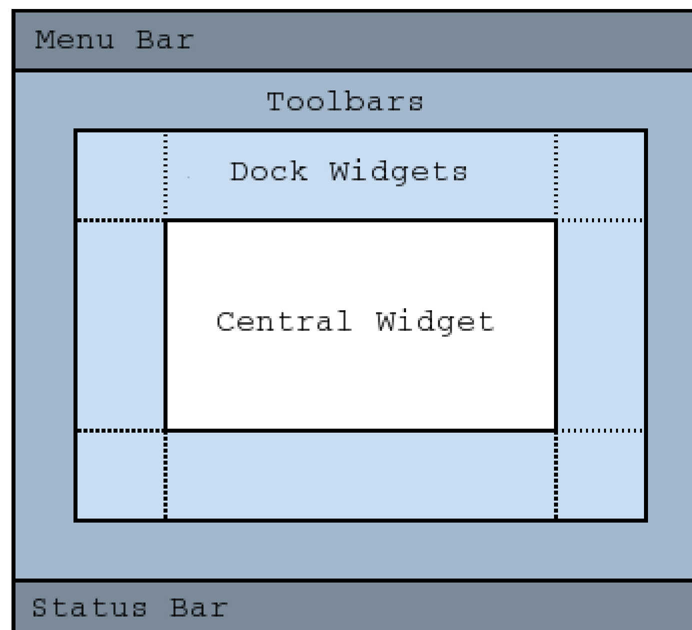
8. La fenêtre principale

La classe QMainWindow a été spécialement conçue pour gérer la fenêtre principale de votre application quand celle-ci est complexe. Parmi les fonctionnalités qui nous sont offertes par la classe QMainWindow, on trouve notamment les menus, la barre d'outils et la barre d'état.

8.1. Présentation de QMainWindow

La classe QMainWindow hérite directement de QWidget. C'est un widget généralement utilisé une seule fois par programme et qui sert uniquement à créer la fenêtre principale de l'application.

Structure de la QmainWindow :



Une fenêtre principale peut être constituée de tout ces éléments :

- Menu Bar : c'est la barre de menus. C'est là que vous allez pouvoir créer votre menu Fichier, Édition, Affichage, Aide, etc.
- Toolbars : les barres d'outils. Dans un éditeur de texte, on a par exemple des icônes pour créer un nouveau fichier, pour enregistrer, etc.
- Dock Widgets : ces docks sont des conteneurs que l'on place autour de la fenêtre principale. Ils peuvent contenir des outils, par exemple les différents types de pinceaux que l'on peut utiliser quand on fait un logiciel de dessin.
- Central Widget : c'est le cœur de la fenêtre, là où il y aura le contenu proprement dit.
- Status Bar : c'est la barre d'état. Elle affiche en général l'état du programme (« Prêt/Enregistrement en cours », etc.).

Nous allons créer notre propre classe de fenêtre principale qui héritera de QmainWindow. Ce projet contiendra trois fichiers :

- main.cpp : la fonction main() ;
- FenPrincipale.h : définition de notre classe FenPrincipale, qui héritera de QMainWindow ;

- FenPrincipale.cpp : implémentation des méthodes de la fenêtre principale.

```
//====main.cpp
#include <QApplication>
#include <QtWidgets>
#include "FenPrincipale.h"

int main(int argc, char *argv[])
{
    Qapplication    app(argc, argv);
    FenPrincipale   fenetre;

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

//====FenPrincipale.h
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtWidgets>
class FenPrincipale : public QMainWindow
{
    public:
    FenPrincipale();
    private:
};

#endif

//====FenPrincipale.cpp
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
}
```

8.2. La zone centrale (SDI et MDI)

On distingue deux types de QMainWindow :

- Les SDI (Single Document Interface) : elles ne peuvent afficher qu'un document à la fois.
- Les MDI (Multiple Document Interface) : elles peuvent afficher plusieurs documents à la fois. Elles affichent des sous-fenêtres dans la zone centrale.

La zone centrale de la fenêtre principale est prévue pour contenir un et un seul widget.

C'est le même principe que les onglets. On y insère un QWidget (ou une de ses classes filles) et on s'en sert comme conteneur pour mettre d'autres widgets à l'intérieur, si besoin est.

8.2.1. Définition de la zone centrale (type SDI)

On utilise la méthode setCentralWidget() de la QMainWindow pour indiquer quel widget contiendra la zone centrale. Faisons cela dans le constructeur de FenPrincipale :

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;
    setCentralWidget(zoneCentrale);
}
```

On a maintenant un QWidget qui sert de conteneur pour les autres widgets de la zone centrale de la fenêtre. On peut donc y insérer des widgets au milieu :

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;
    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;
    QFormLayout *layout = new QformLayout;

    layout->addRow("Votre nom", nom);
    layout->addRow("Votre prénom", prenom);
    layout->addRow("Votre âge", age);

    zoneCentrale->setLayout(layout);
    setCentralWidget(zoneCentrale);
}
```

8.2.2. Définition de la zone centrale (type MDI)

On utilise QMdiArea comme widget conteneur pour la zone centrale :

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;
    setCentralWidget(zoneCentrale);
}
```

La fenêtre est maintenant prête à accepter des sous-fenêtres. On crée celles-ci en appelant la méthode addSubWindow() du QMdiArea. Cette méthode attend en paramètre le widget que la sous-fenêtre doit afficher à l'intérieur.

Là encore, vous pouvez créer un QWidget générique qui contiendra d'autres widgets, eux-mêmes organisés selon un layout.

Dans notre exemple : les sous-fenêtres contiennent juste un QtextEdit.

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;
    QTextEdit *zoneTexte1 = new QTextEdit;
    QTextEdit *zoneTexte2 = new QtextEdit;

    QMdiSubWindow *sousFenetre1 = zoneCentrale->addSubWindow(zoneTexte1);
    QMdiSubWindow *sousFenetre2 = zoneCentrale->addSubWindow(zoneTexte2);

    setCentralWidget(zoneCentrale);
}
```

Ces fenêtres peuvent êtres réduites ou agrandies à l'intérieur même de la fenêtre principale.

On peut leur attribuer un titre et une icône avec les méthodes setWindowTitle, setWindowIcon, etc.

Vous remarquerez que addSubWindow() renvoie un pointeur sur une QMdiSubWindow : ce pointeur représente la sous-fenêtre qui a été créée. Cela peut être une bonne idée de garder ce pointeur pour la suite. Vous pourrez ainsi supprimer la fenêtre en appelant removeSubWindow().

On peut retrouver la liste des sous-fenêtres créées en appelant subWindowList(). Cette méthode

renvoie la liste des QMdiSubWindow contenues dans la QMdiArea.

8.3. Les menus

La barre de menus est accessible depuis la méthode menuBar(). Cette méthode renvoie un pointeur sur un QMenuBar, qui vous propose une méthode addMenu(). Cette méthode renvoie un pointeur sur le QMenu créé.

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // le symbole & pour définir des raccourcis clavier
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");
}
```

Un élément de menu est représenté par une action, gérée par la classe Qaction.

Pour créer une action vous avez deux possibilités :

- soit vous la créez d'abord, puis vous créez l'élément de menu qui correspond ;
- soit vous créez l'élément de menu directement et celui-ci vous renvoie un pointeur vers la QAction créée automatiquement.

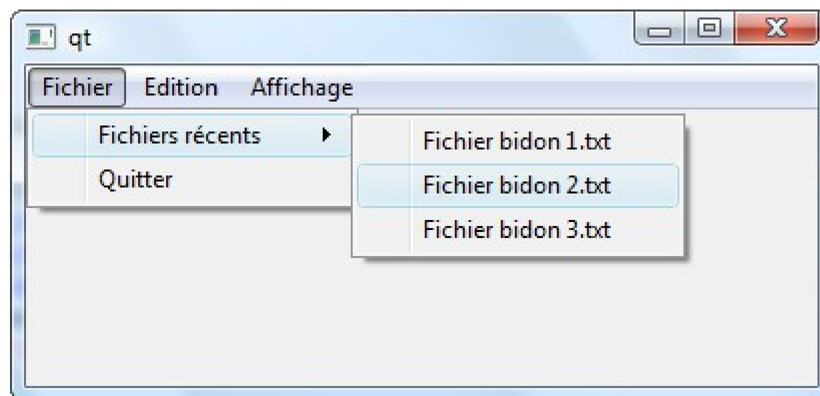
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = new QAction("&Quitter", this);

    menuFichier->addAction(actionQuitter);

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");
}
```

- Dans l'exemple de code ci-dessus, nous créons d'abord une QAction correspondant à l'action « Quitter ». Nous définissons en second paramètre de son constructeur un pointeur sur la fenêtre principale (this), qui servira de parent à l'action.
- Puis, nous ajoutons l'action au menu « Fichier ».

Les sous-menus sont gérés par la classe QMenu.

Au lieu d'appeler addAction() de la QMenuBar, appelez cette fois addMenu() qui renvoie un pointeur vers un QMenu :

```
QMenu *fichiersRecents = menuFichier->addMenu("Fichiers &récents");

fichiersRecents->addAction("Fichier bidon 1.txt");
fichiersRecents->addAction("Fichier bidon 2.txt");
fichiersRecents->addAction("Fichier bidon 3.txt");
```



Pour connecter les signaux des actions à des slots, il faut récupérer le pointeur vers les QAction créées à chaque fois.

Le premier rôle d'une QAction est de générer des signaux, que l'on aura connectés à des slots.

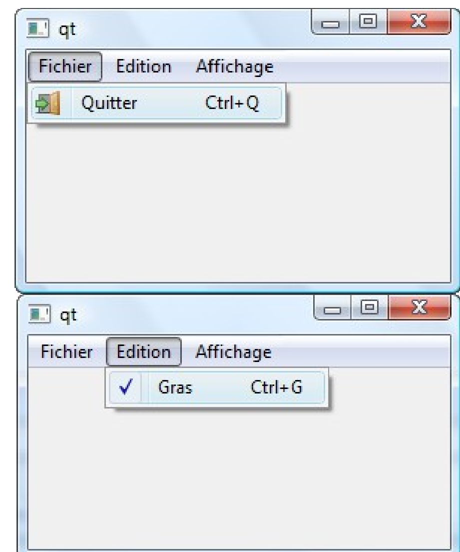
La QAction propose plusieurs signaux intéressants. Le plus utilisé d'entre eux est triggered() qui indique que l'action a été choisie par l'utilisateur.

On peut connecter notre action « Quitter » au slot quit() de l'application :

```
connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
```

Désormais, un clic sur « Fichier > Quitter » fermera l'application.

Remarque : l'évènement hovered() s'active lorsqu'on passe la souris sur l'action.



Chaque action peut avoir une icône. Lorsque l'action est associée à un menu, l'icône est affichée à gauche de l'élément de menu. Pour ajouter une icône, appelez setIcon() et envoyez-lui un Qicon :

```
actionQuitter->setIcon(QIcon("quitter.png"));
```

Lorsqu'une action peut avoir deux états (activée, désactivée), vous pouvez la rendre « cochable » grâce à setCheckable().

```
actionGras->setCheckable(true);
```

On vérifiera dans le code si l'action est cochée avec isChecked().

8.4. La barre d'outils

La barre d'outils est généralement constituée d'icônes et située sous les menus. Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de la barre.

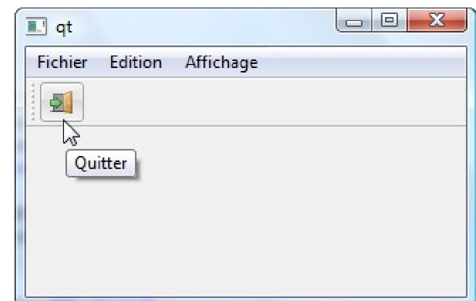
Pour ajouter une barre d'outils, vous devez tout d'abord appeler la méthode addToolBar() et donner un nom à la barre d'outils, même s'il ne s'affiche pas.

Le plus simple est d'ajouter une action à la QToolBar. On utilise la méthode appelée addAction()

qui prend comme paramètre une QAction.

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Création des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Création de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

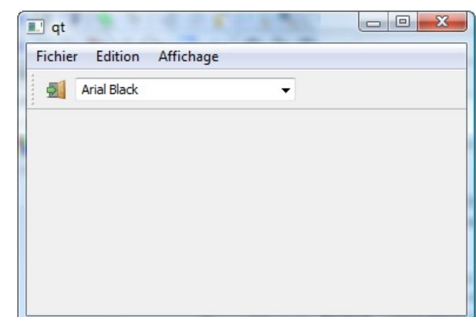
Dans ce code, on voit qu'on crée d'abord une QAction pour un menu, puis plus loin on réutilise cette action pour l'ajouter à la barre d'outils :



La QToolBar gère justement tous types de widgets. Vous pouvez ajouter des widgets avec la méthode addWidget(), comme vous le faisiez avec les layouts :

```
QFontComboBox *choixPolice = new QFontComboBox;
toolBarFichier->addWidget(choixPolice);
```

Ici, on insère une liste déroulante. Le widget s'insère alors dans la barre d'outils :



La méthode addWidget() crée une QAction automatiquement. Elle renvoie un pointeur vers cette QAction créée.

Si votre barre d'outils commence à comporter trop d'éléments, cela peut être une bonne idée de les séparer. C'est pour cela que Qt propose des séparateurs. Il suffit d'appeler la méthode addSeparator() à l'endroit où vous voulez insérer un séparateur :

```
toolBarFichier->addSeparator();
```

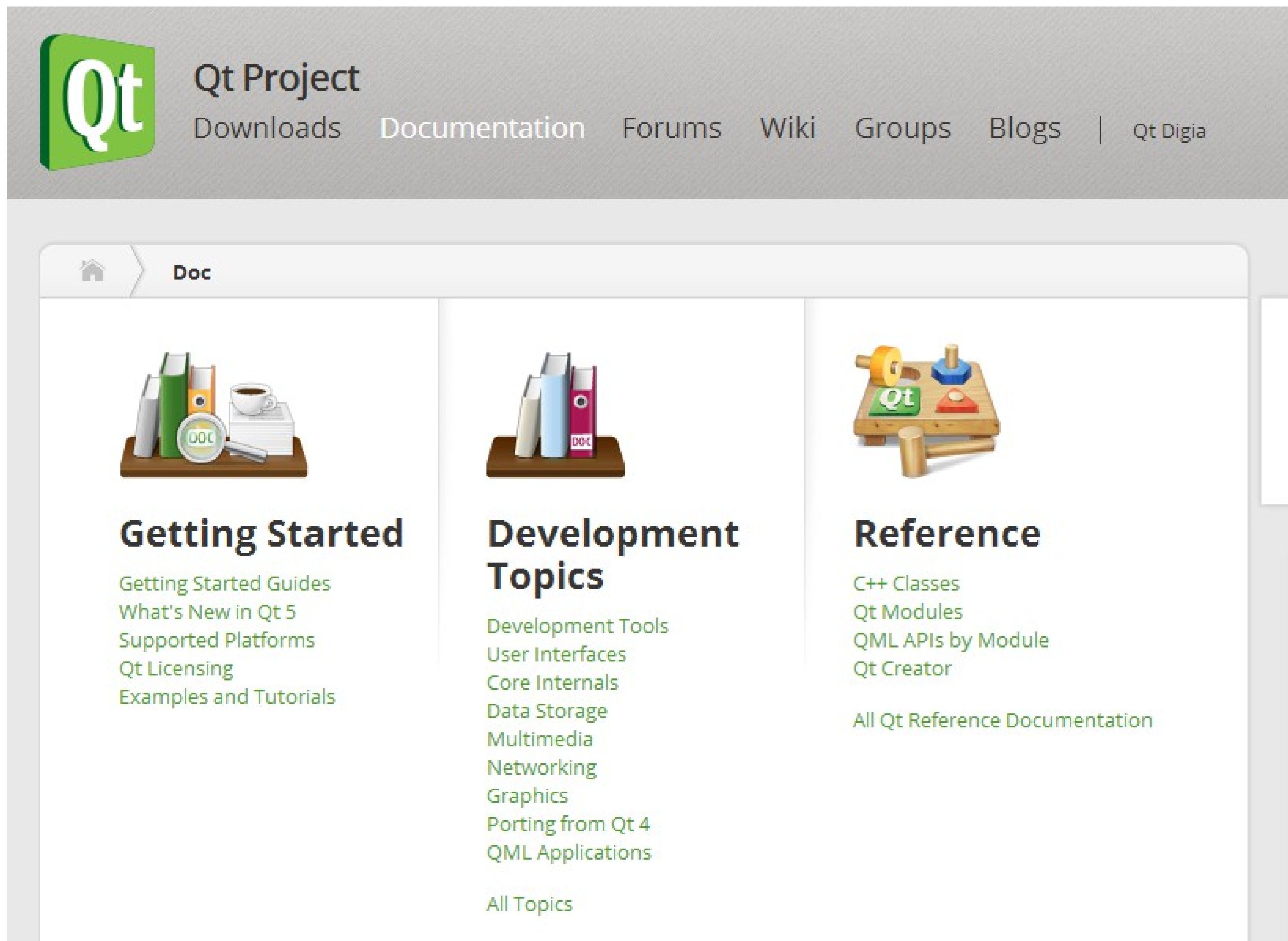
9. Documentation Qt

Il y a deux moyens d'accéder à la documentation :

- si vous avez accès à Internet : vous pouvez aller sur le site de Digia (l'entreprise qui édite Qt) ;

- si vous n'avez pas d'accès à Internet : vous pouvez utiliser le programme Qt Creator qui contient toute la documentation.

Lorsque vous arrivez sur la documentation en ligne, la page de la figure suivante s'affiche.



Les deux sections qui vont nous intéresser le plus sont Development Topics et Reference. Vous pouvez aussi regarder la section "Getting Started" qui contient des exemples et des tutoriels.

La section Reference décrit toutes les fonctionnalités de Qt :

- C++ Classes : la liste de toutes les classes proposées par Qt.
- Qt Modules : la liste des classes de Qt en fonction des modules. Qt étant découpé en plusieurs modules (Qt Core, Qt Widgets…), cela donne une vision de l'architecture globale de Qt.
- Qt Creator : une documentation détaillée sur Qt Creator.

La section Development Topics donne des pages de guide qui servent d'introduction à Qt, mais aussi de conseils pour ceux qui veulent utiliser Qt le mieux possible, thématique par thématique.

9.1. Comprendre la documentation d'une classe

Chaque classe possède sa propre page. Lorsque vous connaissez le nom de la classe et que vous voulez lire sa documentation, utilisez le champ de recherche en haut du menu. Vous pouvez aussi passer par le lien All Classes depuis le sommaire.

Voici un exemple sur la documentation de QlineEdit :

Qt Project
Downloads Documentation Forums Wiki Groups Blogs | Qt Digia

Docs > Qt 5.1 > qtwidgets > **QLineEdit Class | QtWidgets 5.1** Docmark

QLineEdit Class

The QLineEdit widget is a one-line text editor. More...

```
#include <QLineEdit>
```

Inherits: QWidget.

- List of all members, including inherited members

Contents

- Public Types
- Properties
- Public Functions
- Public Slots
- Signals
- Protected Functions
- Detailed Description

Public Types

enum **EchoMode** { Normal, NoEcho, Password, PasswordEchoOnEdit }

Properties

- **acceptableInput** : const bool
- **alignment** : Qt::Alignment
- **cursorMoveStyle** : Qt::CursorMoveStyle
- **cursorPosition** : int
- **displayText** : const QString
- **dragEnabled** : bool
- **echoMode** : EchoMode
- **inputMask** : QString
- **maxLength** : int
- **modified** : bool
- **placeholderText** : QString
- **readOnly** : bool
- **redoAvailable** : const bool
- **selectedText** : const QString

Chaque documentation de classe suit exactement la même structure. Vous retrouverez donc les mêmes sections, les mêmes titres, etc.

Introduction

Au tout début, vous pouvez lire une très courte introduction qui explique en quelques mots à quoi sert la classe.

Ici, nous avons : « The QLineEdit widget is a one-line text editor. », ce qui signifie que ce widget est un éditeur de texte sur une ligne.

Le lien « More… » amène vers une description plus détaillée de la classe. En général, il s'agit d'un mini-tutoriel pour apprendre à l'utiliser.

Ensuite, on donne le header à inclure pour pouvoir utiliser la classe dans votre code. En l'occurrence il s'agit de : #include <QLineEdit>

Puis la classe dont hérite votre classe. Ici, on voit que QWidget est le parent de QLineEdit. Donc QLineEdit récupère toutes les propriétés de QWidget.

Public Types

Les classes définissent parfois des types de données personnalisés, sous la forme d'énumérations.

Ici, QLineEdit définit l'énumération EchoMode qui propose plusieurs valeurs : Normal, NoEcho, Password, etc. Pour envoyer la valeur Password, il faudra écrire : setEchoMode(QLineEdit::Password)

Properties

Les attributs pour lesquels Qt définit des accesseurs. Qt suit cette convention pour le nom des accesseurs :

- propriete() : c'est la méthode accesseur qui vous permet de lire la propriété ;
- setPropriete() : c'est la méthode accesseur qui vous permet de modifier la propriété.

Prenons par exemple la propriété text. C'est la propriété qui stocke le texte rentré par l'utilisateur dans le champ de texte QlineEdit. Pour récupérer le texte, on utilisera setText.

Dans la documentation, la propriété text est un lien. Cela vous amènera plus bas sur la même page vers une description de la propriété.

On vous y donne aussi le prototype des accesseurs :

- `QString text () const`
- `void setText ( const QString & )`

NB : dans la liste des propriétés en haut de la page, notez les mentions 56 properties inherited from QWidget et 1 property inherited from QObject. Comme QLineEdit hérite de QWidget, qui lui-même hérite de QObject, il possède ainsi toutes les propriétés et toutes les méthodes de ses classes parentes !

Les propriétés affichées ne représentent qu'un tout petit bout des possibilités offertes par QLineEdit. Si vous cliquez sur le lien QWidget, vous êtes conduits à la liste des propriétés de QWidget. Vous disposez aussi de toutes ces propriétés dans un QLineEdit !

Public Functions

Méthodes publiques de la classe :

- le (ou les) constructeur(s), pour créer un objet à partir de cette classe ;
- les accesseurs, basés sur les attributs ;
- d'autres méthodes publiques qui ne sont ni des constructeurs ni des accesseurs et qui effectuent diverses opérations sur l'objet.

Cliquez sur le nom d'une méthode pour en savoir plus sur son rôle et son fonctionnement.

9.2. Lire et comprendre le prototype

Le prototype donne une grosse quantité d'informations sur la méthode.

Prenons l'exemple du constructeur QlineEdit :

- `QLineEdit ( QWidget * parent = 0 )`
- `QLineEdit ( const QString & contents, QWidget * parent = 0 )`

Si vous cliquez sur un de ces constructeurs, on vous explique la signification de chacun de ces

paramètres.

Dans la documentation de QlineEdit, la méthode setAlignment() demande un paramètre de type Qt::Alignment. Pour savoir comment envoyer un paramètre de type Qt::Alignment, il suffit de cliquer dans la documentation sur le lien Qt::Alignment.

Dans le cas des énumérations, il suffit d'écrire la valeur (par exemple Qt::AlignCenter). Mais quand la méthode attend un objet issu d'une classe précise pour travailler, il va falloir créer un objet de cette classe et l'envoyer à la méthode.

Prenons par exemple setValidator, qui attend un pointeur vers un QValidator. La méthode setValidator indique qu'elle permet de vérifier si l'utilisateur a saisi un texte valide. Si vous cliquez sur le lien QValidator, vous êtes conduit à la page qui explique comment utiliser la classe QValidator.

Dans l'exemple, QValidator est une classe abstraite et il faut utiliser une de ses classes filles.

Au tout début, la page de la documentation de QValidator vous dit Inherited by QDoubleValidator, QIntValidator, and QRegExpValidator. Cela signifie que ces classes héritent de QValidator et que vous pouvez les utiliser aussi.

Pour saisir un nombre entier, il faudra utiliser QintValidator et créer un objet de type QintValidator :

```
QValidator *validator = new QIntValidator(0, 150, this);
monChamp.setValidator(validator);
```

Public Slots

Tous les slots offerts par QLineEdit ne figurent pas dans la liste ; il faut penser à regarder les mentions comme 19 public slots inherited from QWidget, qui vous invitent à aller voir les slots de QWidget auxquels vous avez aussi accès.

C'est ainsi que vous découvrirez que l'on dispose du slot hide() qui permet de masquer QLineEdit.

Signals

Attention à bien regarder les signaux hérités de QWidget et QObject, car ils appartiennent aussi à la classe QLineEdit.

Additional Inherited Members

Si des éléments hérités n'ont pas été listés jusqu'ici, on les retrouvera dans cette section à la fin.

Par exemple, la classe QLineEdit ne définit pas de méthode statique, mais elle en possède quelques-unes héritées de QWidget et QObject, comme number() qui convertit le nombre donné en une chaîne de caractères de type QString.

```
QString maChaine = QString::number(12);
```

Detailed description

C'est une description détaillée du fonctionnement de la classe. On y accède notamment en cliquant sur le lien « More… » après la très courte introduction du début.