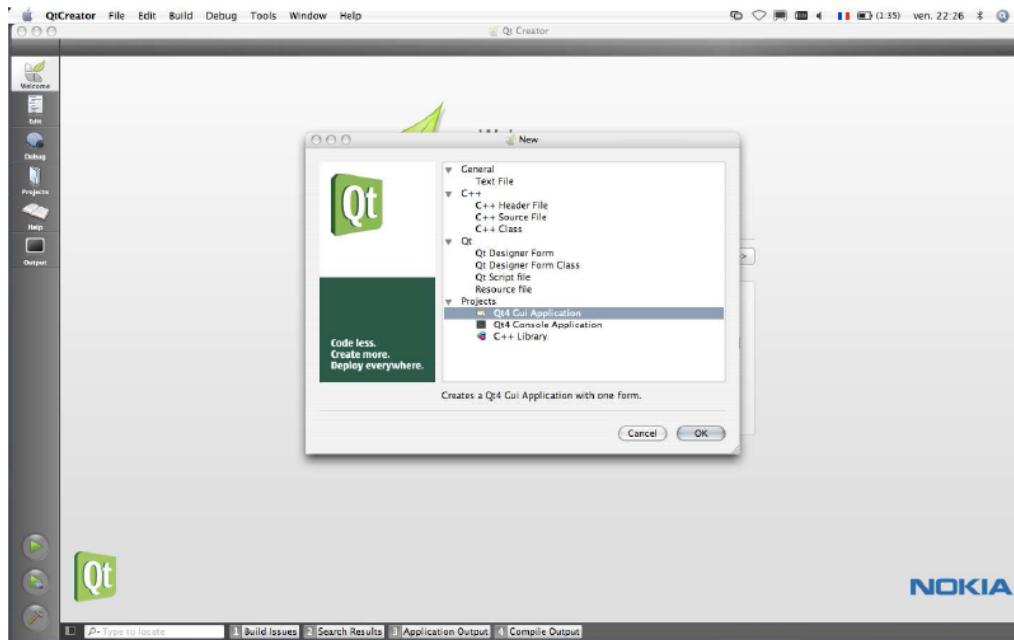


Utilisation de QT

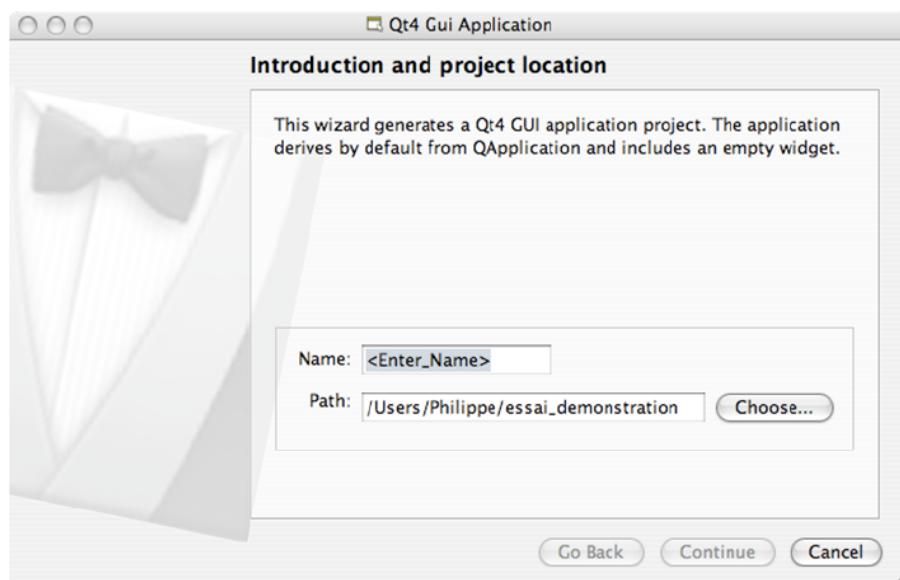
1) Création d'un projet

En utilisant l'icône QtCreator, lancez l'application et créez un projet de type GUI.



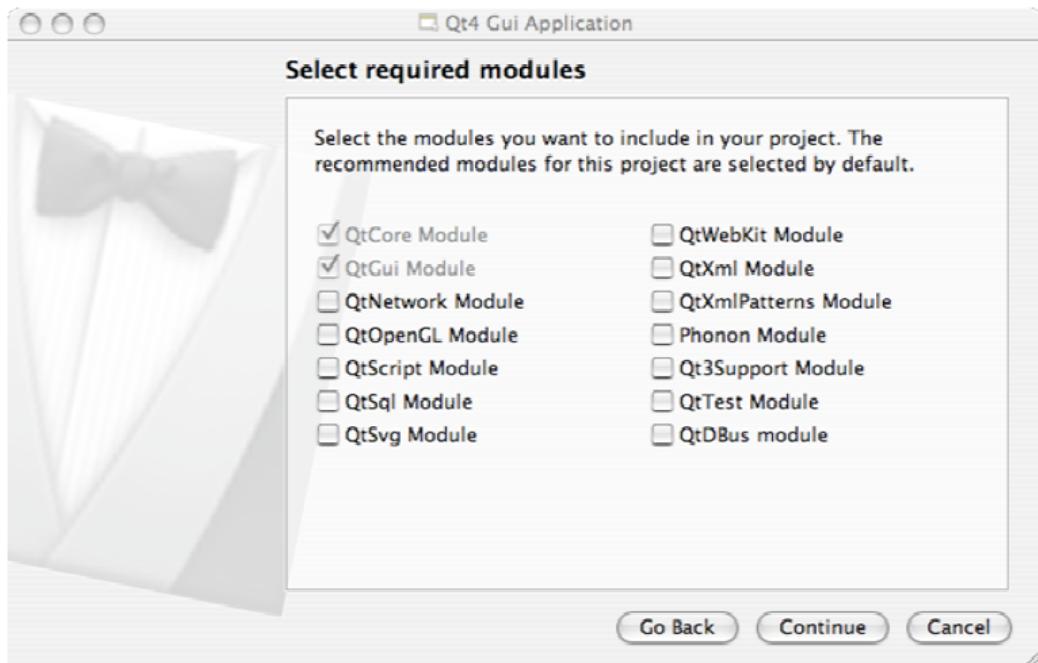
Une succession d'écrans vous permettant alors de créer un projet.

Dans un premier temps, il suffit de choisir un nom de projet et de choisir un répertoire de travail. Attention, ne mettez qu'un seul projet par répertoire car un projet se compose d'un ensemble de fichiers et il est important d'éviter les conflits.



CQT Exercices

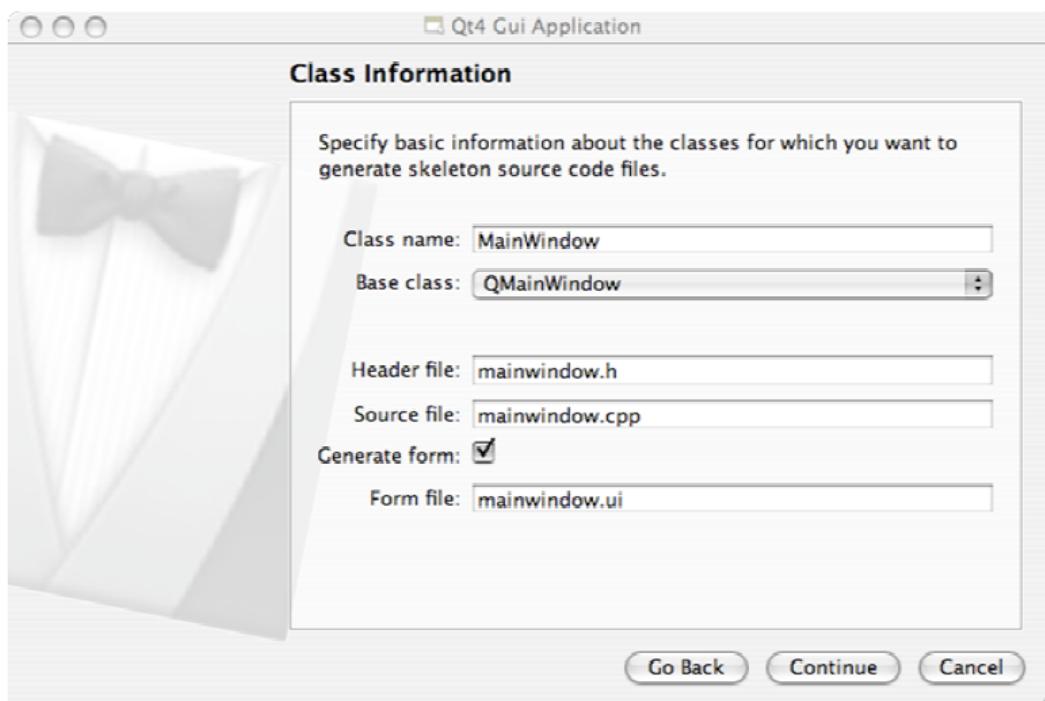
Il est important ensuite de choisir les modèles Qt à inclure dans votre projet. Par défaut seul les modules minimaux sont inclus et en particulier le module permettant de gérer les interfaces graphiques.



Le dernier écran vous permet de modifier d'une part le nom de la classe principale et par conséquent les noms des fichiers C++ correspondant. Par défaut :

- mainwindow.cpp
- mainwindow.h

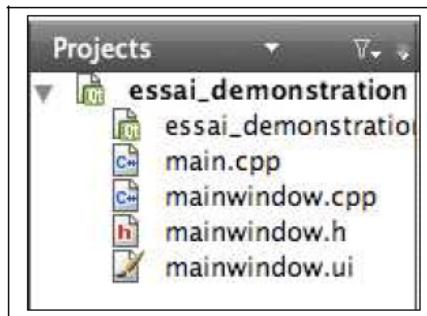
Notons la présence d'un fichier **mainwindow.ui** qui correspond au fichier de l'interface graphique c'est-à-dire permettant au générateur d'interface de fonctionner.



2) 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**.



Le fichier main.cpp

On n'a vraiment besoin de le modifier que si on souhaite modifier la fenêtre de démarrage ou bien inclure de nouvelles librairies Qt.

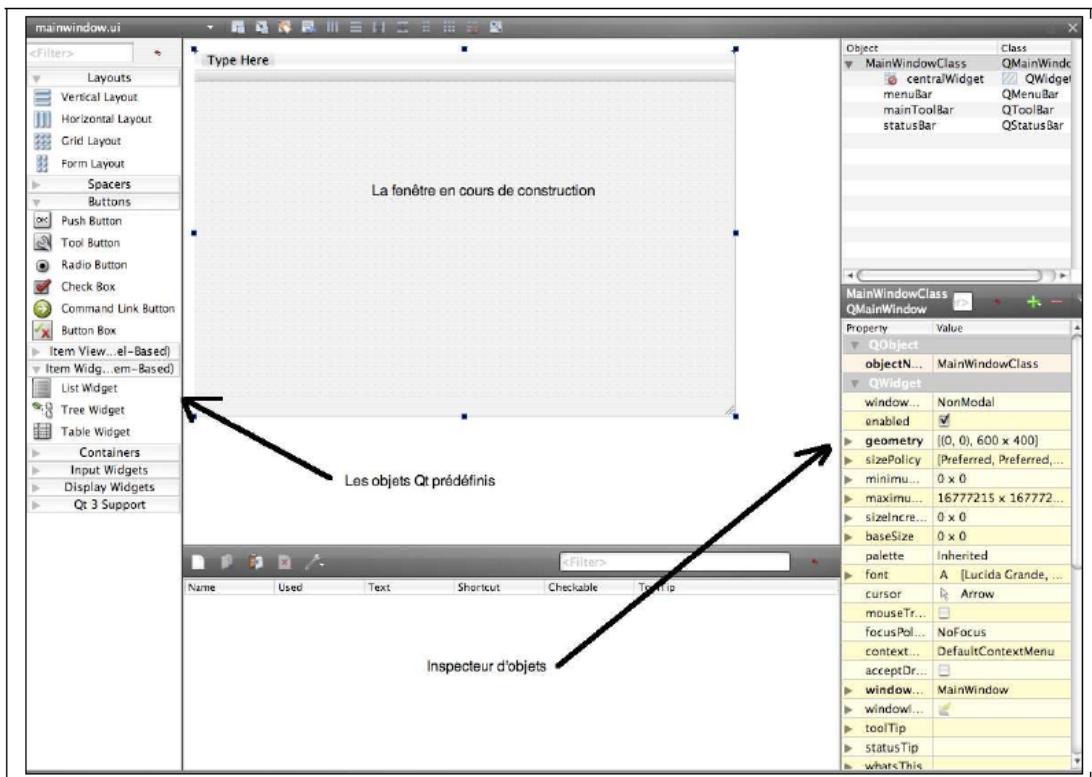
Les fichiers mainwindow.cpp et mainwindow.h

Ces fichiers devront être souvent modifiés pendant la conception de l'interface graphique pour ajouter de nouveau slot ou signaux et ainsi faire communiquer les objets de l'interface.

3) Le générateur d'interface

Un double click sur **mainwindow.gui** lance automatique le générateur d'interface.

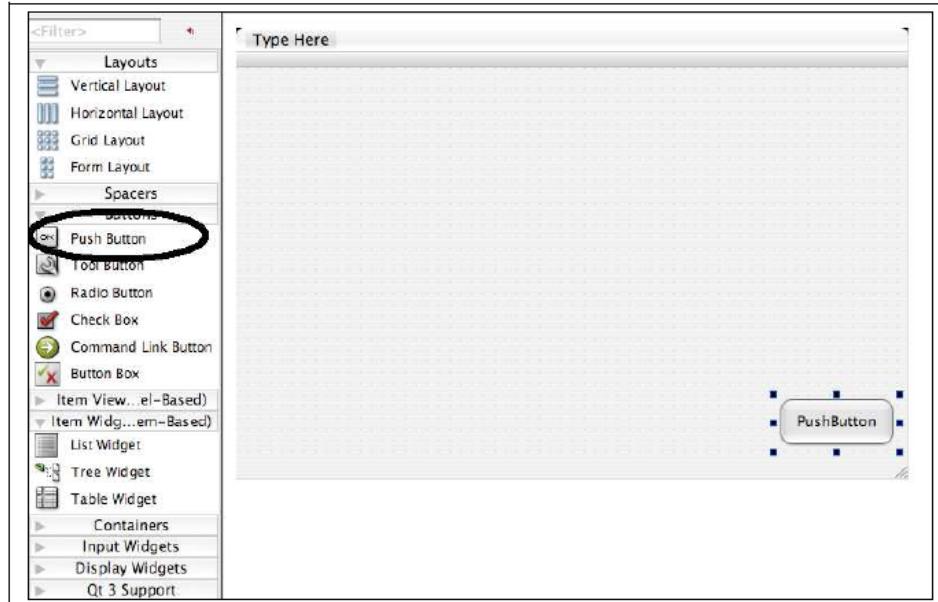
CQT Exercices



3.1. Création d'un bouton « Quitter »

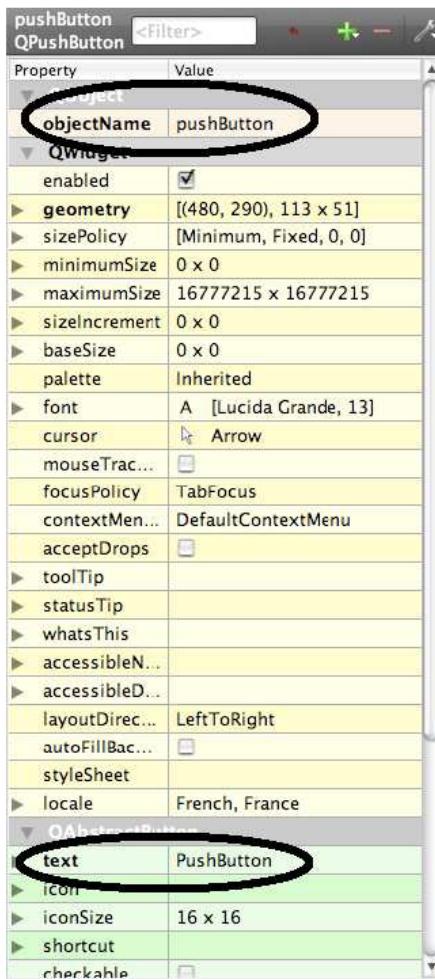
Etape 1. Créeation d'un bouton dans la fenêtre.

Dans la section Buttons, par click à la souris, insérer un bouton sur la fenêtre, par exemple à bas à droite.



L'inspecteur d'objet affiche alors les caractéristiques de l'objet bouton qui vient d'être crée.

CQT Exercices

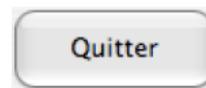


Deux propriétés sont importantes :

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.

Modifions cette propriété en « Quitter ». Le bouton ressemble alors à ce qui suit :



Etape 2. Attacher du code sur l'événement click sur le bouton.

Pour cela dans le fichier `mainwindow.h` créer une nouvelle section nommée « Private Slots » et définir une procédure nommée par exemple « `BoutonQuitter` ».

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui/QMainWindow>

namespace Ui
{
    class MainWindowClass;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindowClass *ui;

private slots:
    void BoutonQuitter();
};

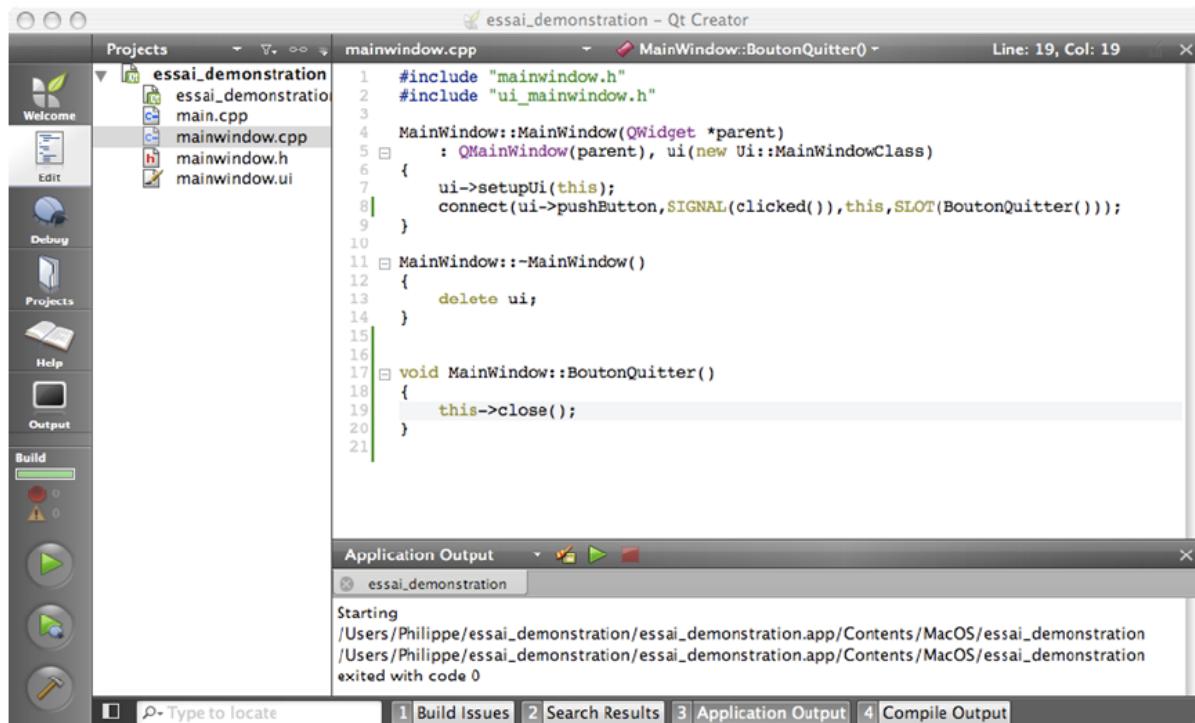
#endif // MAINWINDOW_H
```

Editer ensuite le fichier `mainwindow.cpp`.

CQT Exercices

Il faut inclure dans ce fichier :

- Le corps de la méthode BoutonQuitter
- Une connexion entre le signal ou événement « click » et la méthode BoutonQuitter ;



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()));
```

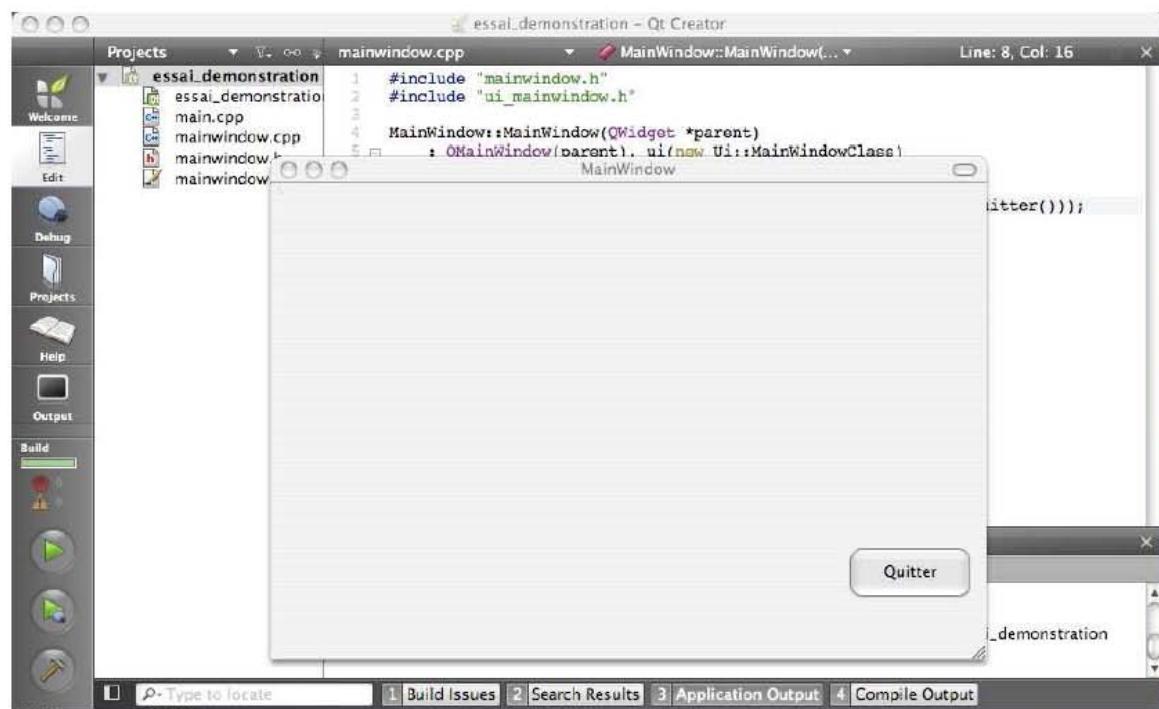
Celle-ci ce compose de 3 parties principales :

- Le premier paramètre (ui->pushButton) fait référence à l'object graphique ;
- Le deuxième SIGNAL(clicked()) fait référence à l'événement concerné (ici « click ») ;
- Le quatrième crée la connexion entre l'objet graphique et la procédure (en Qt le lien s'appelle un slot).

Etape 3. Vérification.

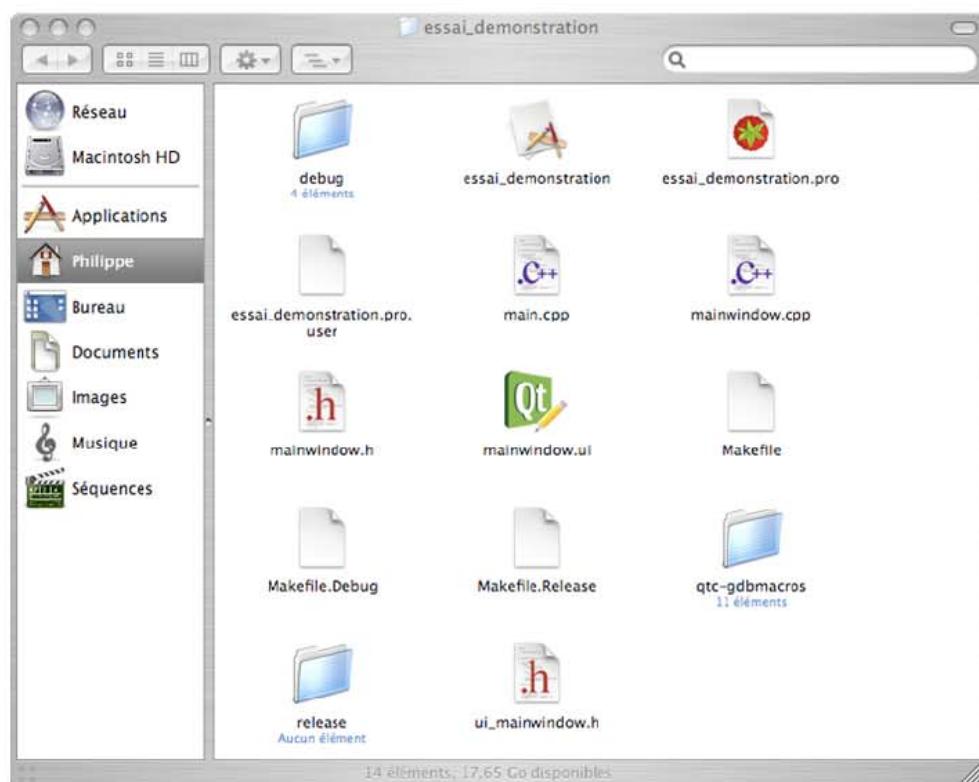
Après compilation et en lançant directement le code à partir de QtCreator on obtient le résultat suivant avec la fermeture de la fenêtre sur l'événement click du bouton.

CQT Exercices



En examinant le dossier de travail, on retrouve les différents fichier du projet ainsi qu'un fichier nommé « essai_demonstration.app » sous MacIntosh ou « essai_demonstration.exe » sous Windows.

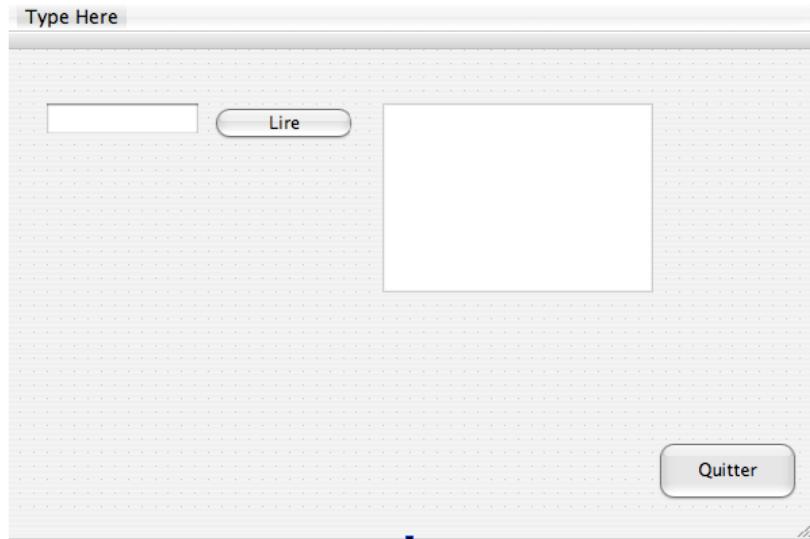
Le double click sur le fichier lance automatique l'application. Attention sous windows, il est nécessaire que votre variable d'environnement PATH contienne un lien vers les fichiers .dll de Qt. Si ce n'est pas le cas, penser à cliquer sur le poste de travail et à modifier vos variables d'environnement.



CQT Exercices

3.2. Afficher des messages sur la fenêtre.

Créez une fenêtre avec un « Lineedit » et un « textEdit » séparé par un bouton intitulé « Lire » comme indiqué sur la copie d'écran ci-dessous.



Ces deux éléments se trouvent dans la partie :



Comme précédemment, déclarer une nouvelle procédure dans le fichier mainwindow.h :

```
private slots:  
    void BoutonQuitter();  
    void BoutonLire();
```

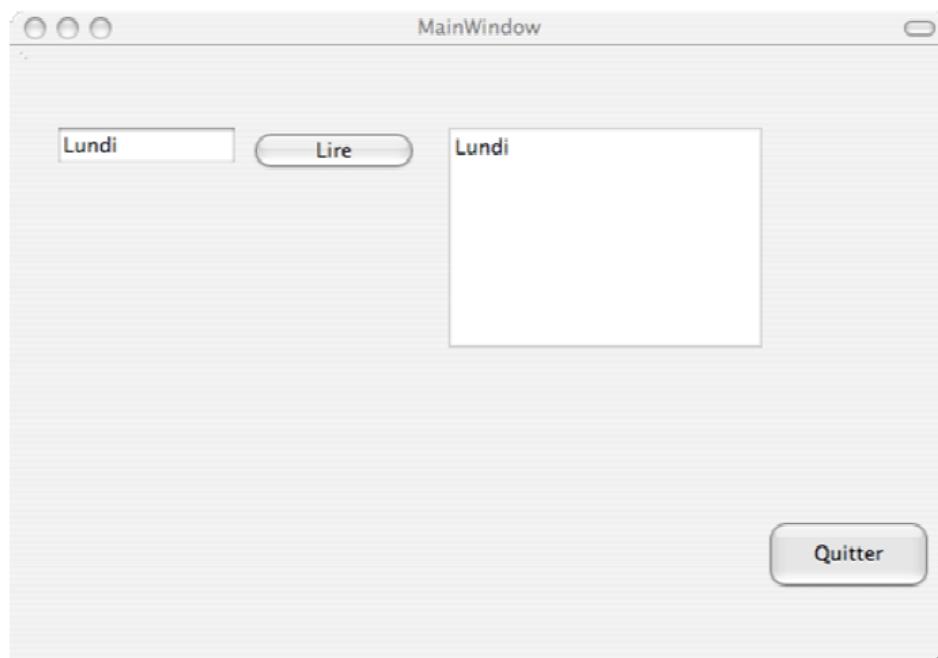
Le fichier mainwindow.cpp doit être modifié comme suit :

CQT Exercices

```
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
```

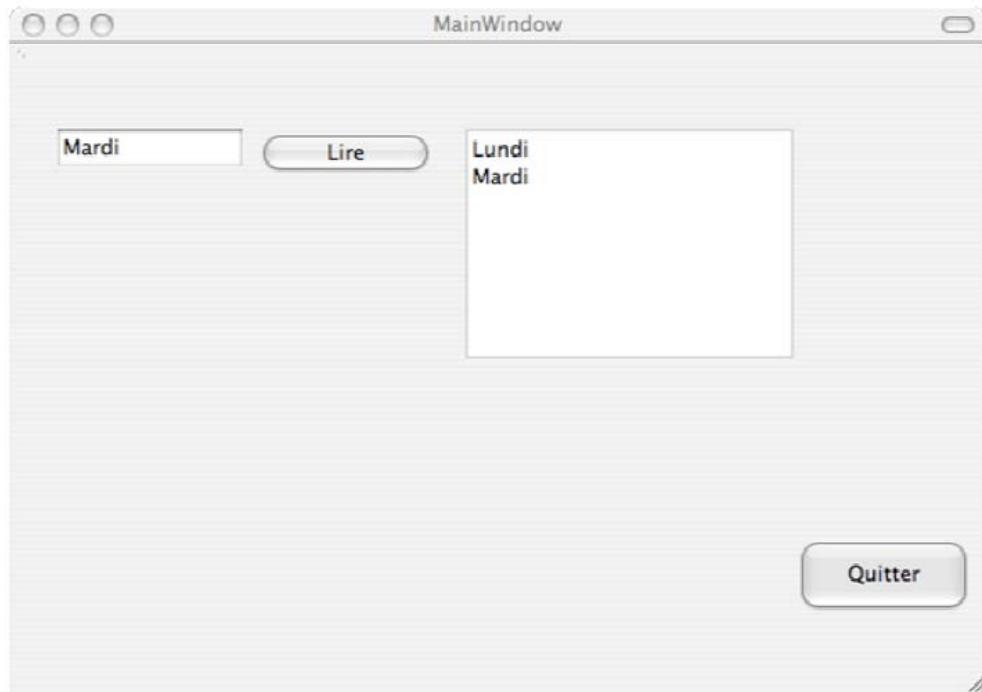
En venant du C++ standard, il faut prendre soin d'utiliser les types spéciaux Qt dont en particulier le type `QString` et non pas le type `string` classique.

Le résultat à l'exécution est le suivant :



On peut constater que le texte du LineEdit est bien ajouté au fur et à mesure dans la zone de type QTextEdit. Nous avons sur cet exemple mis en évidence la différence entre une zone de type LineEdit et une zone de type QTextEdit.

CQT Exercices



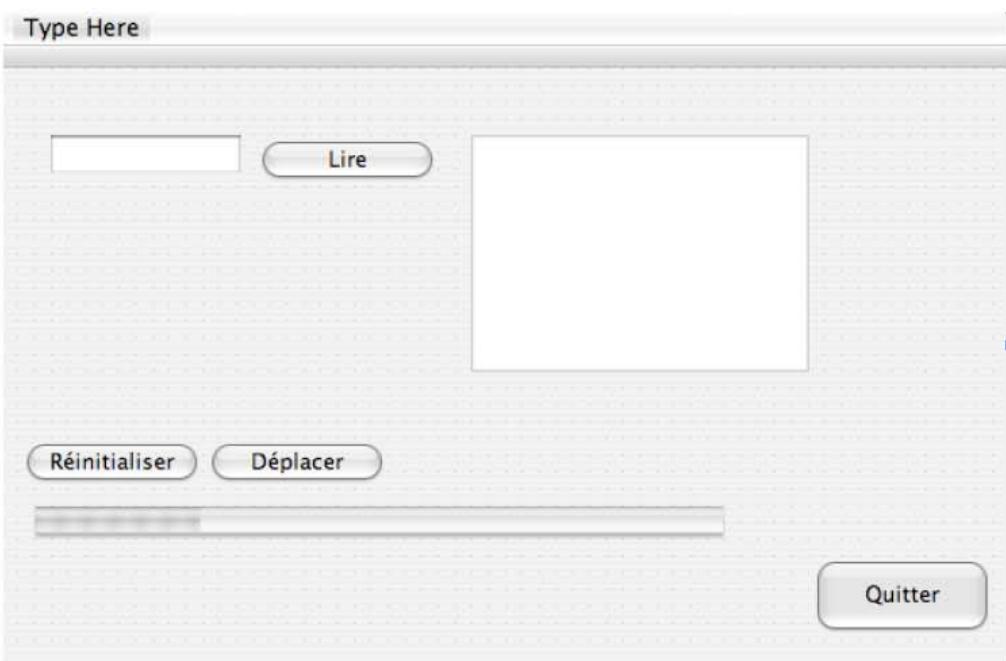
3.3. Utilisation des barres de progression.

Ajoutez sur la fenêtre une barre de progression qui se trouve dans la section « Display Widgets ».



Ajouter 3 boutons afin d'obtenir une fenêtre comme celle-ci :

CQT Exercices



Comme précédemment, modifiez le fichier mainwindow.h :

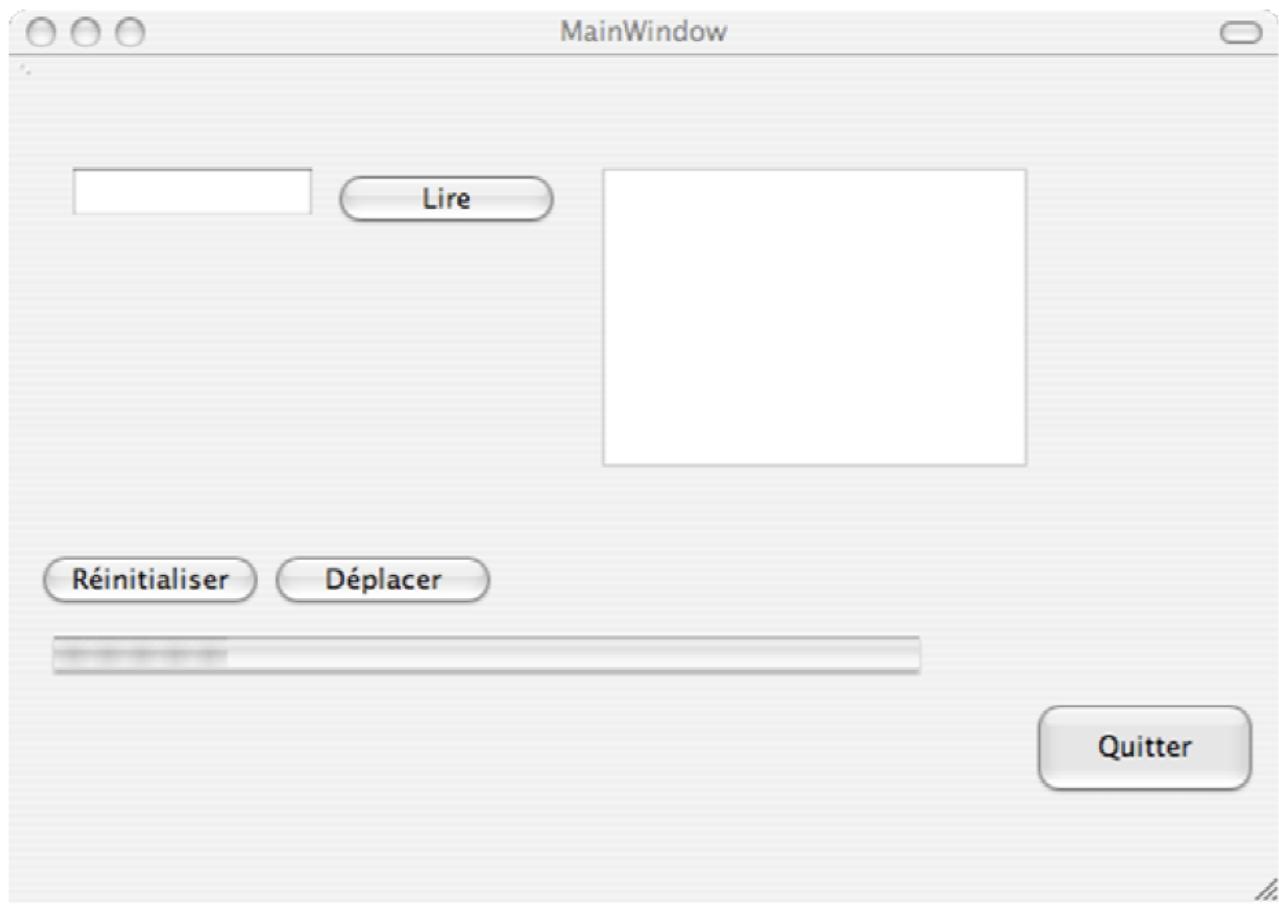
```
void BoutonQuitter();
void BoutonLire();
void BoutonReinitialiser();
void BoutonAugmenter();
```

Le code du fichier mainwindow.cpp est modifié comme suit :

```
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    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
```

CQT Exercices

Ceci qui donne finallement :



4) Conclusion

Voila un premier passage en revue des moyens mis à disposition du développeurs Qt.

CQT Exercices

1 - Crédation d'un projet

La fenêtre d'accueil de QtCreator (cf. ci-contre) se caractérise par la présence d'un certain nombres d'éléments qui resteront en permanence à notre disposition lors de l'utilisation de ce logiciel :

- Une barre de menu tout à fait classique ;
- Un bandeau gauche et un bandeau inférieur proposant divers contrôles qui permettent un accès rapide aux commandes les plus fréquemment utilisées ;
- Une zone centrale dont le contenu dépend du type d'activité en cours.

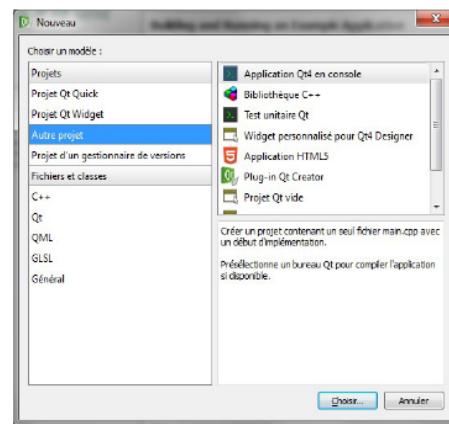
Dans le menu <Fichier>, choisissez la commande <Nouveau fichier ou projet...> .



Le dialogue qui apparaît alors propose différentes options. Nous ne souhaitons pas créer un simple fichier, mais un **projet complet**, c'est à dire un ensemble organisé de fichiers à partir desquels QtCreator sera capable de générer un programme.

Parmi les types de projets C++ disponibles, choisissez <Autre projet> <Application Qt4 en console> et cliquez sur [Choisir] .

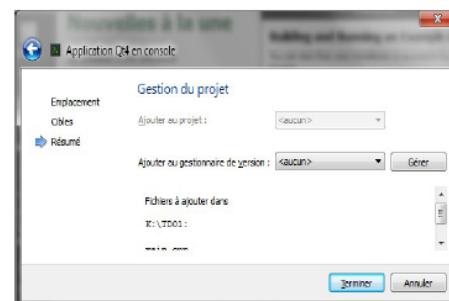
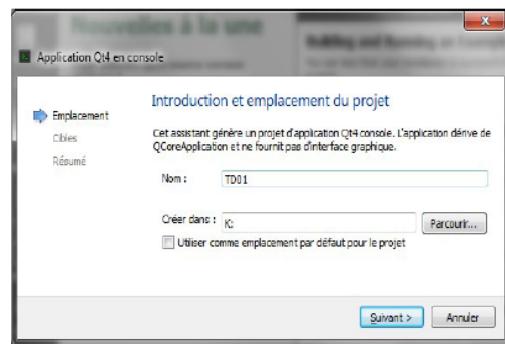
Dès que nous aurons un peu avancé dans notre maîtrise du logiciel et du langage, nous passerons à des projets générant des programmes pourvus d'une interface graphique (`<QWidget>`<Application graphique Qt>).



Le dialogue suivant vous permet de choisir un nom et un emplacement où sera créé le dossier à l'intérieur duquel seront rangés les fichiers constituant le projet.

Une fois ces informations fournies, cliquez sur le bouton [Suivant >] .

Les deux fenêtres suivantes (représentées ci-dessous) ne demandent aucune intervention de votre part, cliquez simplement sur le bouton [Suivant] de la première  puis sur le bouton [Terminer] de la seconde .

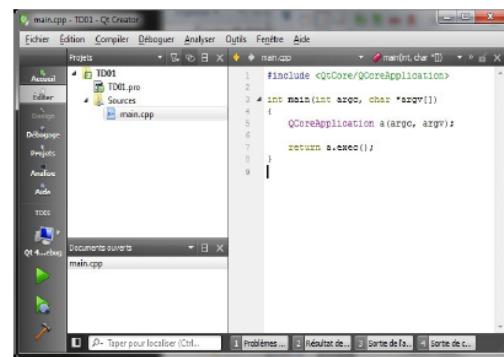


2 - Le contenu initial du fichier main.cpp

Une fois le projet créé, QtCreator passe spontanément en mode Edition (le mode auquel permet d'accéder le bouton [Edit] qui figure dans le bandeau gauche). Dans ce mode, le logiciel propose par défaut une zone de navigation qui permet de choisir le fichier sur lequel nous souhaitons intervenir.

Deux choix nous sont proposés : un fichier .pro qui décrit la façon dont notre programme va être construit et dont nous n'allons pas nous occuper pour l'instant, et le fichier main.cpp.

Double-cliquez sur ce dernier pour en faire apparaître le contenu □.



```

#include <QtCore/QCoreApplication>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    return a.exec();
}

```

En dépit de sa brièveté, le texte contenu dans le fichier main.cpp appelle de nombreux commentaires et il est fort possible que tout ne vous semble pas clair dès la première séance : tout est nouveau, et il faut un peu de temps et de pratique pour s'habituer et découvrir qu'il n'y a là rien de réellement compliqué.

De quoi s'agit-il ?

D'un texte écrit en C++ (d'où le choix de l'extension .cpp pour le nom du fichier) qui décrit les opérations que notre programme devra exécuter lorsque nous le mettrons en route.

Attention à ne pas confondre un texte en C++ avec le programme qu'il permet de générer. Des traitements complexes sont nécessaires pour créer effectivement un programme exécutable à partir d'un texte source rédigé dans un langage de programmation tel que C++. C'est QtCreator qui va se charger d'appliquer ces traitements, mais leur succès n'est pas garanti : si le texte source ne respecte pas les règles du langage, il peut s'avérer impossible de l'utiliser pour créer un programme.

Dans son état actuel, le texte source comporte deux parties : une directive d'inclusion (ligne 1) et la définition d'une fonction (lignes 2 à 6).

Notion de fonction

La description des opérations devant être effectuées par un programme peut s'avérer très longue (plusieurs milliers de pages, dans certains cas). Pour faciliter la rédaction et la correction de ce texte, on préfère le scinder en unités d'une taille plus maîtrisable, un peu comme on divise le texte d'un roman en chapitres. En C++, ces unités s'appellent des fonctions et leur définition obéit à des règles très précises :

- La définition d'une fonction commence par un en-tête (ligne 2) qui mentionne le nom de la fonction (l'équivalent du titre du chapitre, main dans le cas présent) et différentes informations auxquelles nous n'allons pas nous intéresser au cours de cette première étape.
- L'en-tête est suivi d'un couple d'accolades (lignes 3 et 6) qui délimitent ce qu'on appelle le corps de la fonction. A l'intérieur de ce corps (lignes 4 et 5) figurent les instructions qui spécifient les traitements qui auront lieu lorsque la fonction sera exécutée.

Le corps d'une fonction constitue un cas particulier de bloc d'instructions : une liste d'instructions délimitée par un couple d'accolades.

Les fonctions s'appellent les unes les autres, sauf main()

L'intérêt d'une fonction réside dans le traitement que décrivent les instructions contenues dans son corps. Pour que ce traitement soit effectué, il faut que la fonction soit appelée, c'est à dire qu'une instruction mentionnant le nom de la fonction soit exécutée.

Où peut donc bien figurer une telle instruction ?

Aucun suspens : dans le corps d'une autre fonction (c'est le seul endroit où des instructions exécutables ont le droit de figurer).

Si l'exécution d'une fonction ne peut être déclenchée que par une autre fonction elle-même en cours d'exécution, qui commence ?

CQT Exercices

Par définition, le lancement d'un programme se traduit par l'exécution des instructions contenues dans la fonction `main()`, qui jouit donc d'un statut exceptionnel : elle n'a pas besoin d'être appelée par une de ses collègues pour se mettre au travail.

Le statut de `main()` est même encore plus exceptionnel que ça : son appel explicite est *interdit*.

La présence d'une fonction `main()` est donc indispensable pour qu'un texte source puisse donner naissance à un programme. C'est la raison pour laquelle la création d'un projet par QtCreator s'accompagne de la rédaction automatique d'une fonction `main()` minimale.

Types et variables

Les traitements effectués par un programme s'appliquent à des données représentées dans la mémoire de l'ordinateur. En C++, les notions de variable et de type permettent de définir les conventions de codage utilisées pour représenter ces données, ainsi que les opérations élémentaires qui peuvent leur être appliquées.

On peut voir les variables comme des sortes de boîtes : elles portent un **nom** (l'équivalent d'une étiquette collée sur la boîte et permettant de la désigner sans ambiguïté), ont un **contenu** (ce qu'on appellera la **valeur** de la variable) mais ne peuvent pas contenir n'importe quoi (la nature de ce qu'une variable peut contenir dépend de son **type**).

Si une boîte est de type "étau à violon", il sera impossible d'y ranger un saxophone ou une contrebasse, sans parler d'un porte-avions.

Notre exploration du langage va commencer par l'usage de variables de différents types, et nous devons d'ores et déjà nous faire à l'idée que ces types se répartissent en trois familles :

- Les **types standard** sont définis une fois pour toutes par la norme internationale régissant le langage C++ (ISO/IEC 14882:1998). Ces types sont disponibles quel que soit le contexte et permettent de créer des variables dans le corps de n'importe quelle fonction. Ils sont simples et peu nombreux (trois, fondamentalement, un peu plus si on considère toutes les variantes offertes).

- D'autres types ont été créés par des programmeurs utilisant le langage C++ et sont mis à votre disposition par l'intermédiaire de **librairies**. On peut se représenter une librairie comme une collection de fragments de programmes pré-fabriqués, qui vont nous permettre de monter nos programmes beaucoup plus rapidement et facilement que s'il nous fallait utiliser uniquement les briques élémentaires proposées par le langage lui-même. Pour utiliser un type défini dans une librairie, il faut réunir deux conditions : la librairie en question doit être disponible (sous la forme d'un fichier .lib) au moment où le programme est généré **ET** le fichier dans lequel le type va être utilisé doit déclarer cette intention (en général au moyen de directives d'inclusion analogues à celle figurant sur la ligne 1). Ces types peuvent être très élaborés et se compter par milliers. L'usage d'une librairie implique donc une consultation fréquente de sa documentation.

- La troisième famille regroupe les **types que vous définirez vous-même** au cours de l'écriture de votre programme. Leur nombre et leur niveau d'élaboration ne dépend donc que de vous, mais leur utilisation exige évidemment qu'ils soient parfaitement définis au moment où vous demanderez à QtCreator de créer un programme à partir de votre texte source.

Le corps d'une fonction peut contenir des définitions de variables

La ligne 5 illustre la façon dont une variable est créée : le corps d'une fonction contient une ligne indiquant un **type**, un **nom** et une **valeur d'initialisation**.

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

Il s'agit ici d'un type défini dans la librairie Qt (ce qui exige la directive d'inclusion de la ligne 1) et la variable est baptisée `a`. La complexité du type `QCoreApplication` ne nous permet pas d'entrer pour l'instant dans les détails de l'initialisation de la variable `a` et de l'usage qui en est fait à la ligne 7.

Avant d'explorer plus avant les joies de la création de variables de types plus simples, voyons un peu à quoi ressemble le programme dans son état initial.

3 - Une console comme interface utilisateur

Pour demander à QtCreator de créer un programme à partir des fichiers de notre projet, cliquez sur le bouton représentant un marteau qui est situé tout en bas du bandeau gauche .

Comme nous n'avons pas encore touché au contenu du fichier `main.cpp`, il devrait se prêter de bonne grâce aux traitements qui l'attendent et dont le succès va se traduire par le passage du témoin de construction à l'état "construction réussie" (cf. ci-contre).

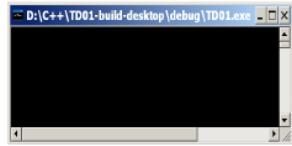


Selon la puissance de l'ordinateur que vous utilisez et la complexité du projet concerné, les traitements permettant la création du programme peuvent durer plus ou moins longtemps. Dans le cas présent, une durée de 4 à 5 secondes peut être considérée comme normale.

CQT Exercices

Une fois la construction réussie, vous pouvez lancer l'exécution du programme en cliquant sur la flèche verte située immédiatement au dessous du témoin de construction .

L'interface utilisateur de notre programme est une fenêtre classique (avec sa barre de titre, sa case de fermeture ou tous les dispositifs normaux proposés par votre système) qui simule un environnement de type "console texte", c'est à dire un dispositif où la communication entre programme et utilisateur se résume à un échange de messages écrits.



Dans ce type d'interface, le programme affiche un message (typiquement, une question) et attend que l'utilisateur saisisse un texte au clavier et achève sa saisie en pressant la touche [Entrée]. Le programme cherche alors à exploiter le texte saisi par l'utilisateur, puis pose éventuellement une nouvelle question, et ainsi de suite.

Ce type d'interaction était le seul connu avant l'apparition des interfaces dites "graphiques", qui se caractérisent par l'utilisation de la souris et de dispositifs logiciels tels que les icônes, les menus et les différents types de boutons que vous connaissez. Du point de vue de l'écriture des programmes, ces deux contextes diffèrent radicalement. Dans un univers "console", c'est le programme qui décide du déroulement des opérations : il pose ses questions les unes après les autres, dans un ordre choisi par le programmeur, et sera incapable d'exploiter une saisie qui ne correspondrait pas à ses attentes. Dans une interface graphique, au contraire, c'est l'utilisateur qui a l'initiative et le programme doit être conçu pour réagir correctement à des sollicitations qui lui sont adressées dans un ordre imprévisible.

Dans son état actuel, le programme n'affiche rien et ne se prête à aucune saisie de la part de son utilisateur. La seule chose que vous puissiez en faire, c'est cliquer sur la case de fermeture de la fenêtre, ce qui met fin à l'exécution du programme .

4 - Utilisation de la console

L'affichage et la saisie de texte dans une console ne sont pas des opérations supportées directement par le langage C++.

Ce langage n'est pas spécialement destiné à créer des programmes évoluant dans ce contexte. La plupart des programmes que vous allez écrire utiliseront une interface graphique, et il faut aussi signaler que de nombreux programmes écrits en C++ sont destinés à des machines dépourvues du couple écran/clavier qui équipe les micro-ordinateurs actuels.

Affichage de texte

Pour utiliser la console, nous allons donc avoir recours à la librairie standard, que QtCreator met à la disposition de nos programmes sans que nous ayons autre chose à faire que d'annoncer notre intention d'utiliser certaines de ses fonctionnalités :

```
#include <QtCore/QCoreApplication>
#include <iostream>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    std::cout << "Bonjour !";
    return a.exec();
}
```

Cette précaution oratoire étant prise (ligne 2), nous sommes en mesure de demander (ligne 6) à notre fonction d'afficher un message dans sa console. Trois points sont à souligner :

- Le texte que nous voulons voir apparaître est placé entre guillemets.
- La demande d'affichage est symbolisée par deux chevrons qui forment une sorte de flèche.
- Cette flèche envoie le texte en direction de la console, désignée par une appellation qui peut sembler un peu barbare mais qui obéit à une certaine logique : `std::` rappelle qu'il s'agit d'un dispositif relevant de la librairie standard, le `c` est simplement l'initiale du mot "console" et le `out` indique que la console est ici utilisée comme une sortie (le message va du programme vers l'utilisateur, c'est à dire de l'intérieur de la machine vers l'extérieur).

Ajoutez les lignes 2 et 6 à votre fichier main.cpp .

En C++, les instructions exécutables se concluent par un point virgule.

Attention aux fautes de frappe. Les deux chevrons constituent un mot du langage C++, il faut donc éviter de briser ce mot en laissant un espace entre eux...

Reconstruisez , puis exécutez  votre programme .

Bravo ! Comme des milliers d'apprentis programmeurs avant vous, vous venez de faire vos premiers pas en C++ sous la forme d'un "Hello, world!" absolument réglementaire (bien que francophone).

CQT Exercices

Saisie de texte

Récupérer un texte saisi par l'utilisateur est une opération un peu plus compliquée. On peut deviner que la console va être désignée par `std::cin` (puisque le message va maintenant de l'extérieur vers l'intérieur de la machine) et qu'une flèche composée de deux chevrons va diriger le texte provenant de la console vers...

Vers quoi, justement ?

Pour que le programme soit en mesure d'utiliser l'information par la suite, il FAUT que celle-ci soit stockée en mémoire. Il nous faut donc créer une variable, c'est à dire une boîte susceptible de contenir le nom que l'utilisateur va nous donner. La librairie standard offre justement, sous le nom de `std::string`, un type qui convient parfaitement pour stocker des chaînes de caractères. Comme la variable doit exister avant que la saisie ne s'effectue, nous écrirons donc :

```
int main(int argc, char *argv[])
{
QCoreApplication a(argc, argv);
std::cout << "Comment vous appelez-vous ? ";
std::string nomUtilisateur;
std::cin >> nomUtilisateur;
```

L'exécution de la ligne 5 crée une variable nommée `nomUtilisateur`, de type `std::string`.

L'exécution de la ligne 6 range dans cette boîte la séquence de lettres que l'utilisateur a tapée. Ceci signifie que l'exécution de la ligne 6 peut durer très longtemps : tant que l'utilisateur n'a pas pressé la touche [Entrée], la saisie est en cours et le programme ne peut pas continuer.

Une fois la saisie terminée, le contenu de la boîte peut être utilisé, même si nous n'avons évidemment aucune idée du nom de l'utilisateur au moment où nous écrivons le programme :

```
std::cout << "Bonjour, ";
std::cout << nomUtilisateur;
std::cout << ". Je suis absolument ravi de faire votre connaissance !";
return a.exec();
}
```

Le phénomène intéressant est illustré sur la ligne 8 : ce n'est pas le nom de la variable qui est affiché sur l'écran, mais son contenu.

Lorsque le contexte s'y prête, le nom d'une variable désigne le contenu de celle-ci.

Donnez à votre fonction `main()` le contenu décrit ci-dessus □ et faites construire le programme □.

Vérifiez que l'exécution de celui-ci est conforme à vos attentes □.

Les caractères apparaissent à l'écran à mesure que vous les tapez pour indiquer votre nom. Il s'agit là d'un simple écho proposé par la console elle-même, pour vous permettre de savoir où vous en êtes et de corriger une éventuelle erreur. Le programme, lui, n'aura accès à votre saisie que lorsque vous finirez par presser la touche [Entrée].

5 - Affectation

Il arrive évidemment que le contenu d'une variable ne dépende pas (ou pas seulement) d'une saisie effectuée par l'utilisateur. Pour illustrer cette situation, nous allons introduire dans notre programme des variables d'un type standard : `int`. Ces boîtes peuvent contenir des valeurs numériques entières. Notre programme va simplement demander son `age` à l'utilisateur, puis utiliser cette information pour calculer en quelle année celui-ci aura 100 ans.

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <qdate>

int main(int argc, char *argv[])
{
QCoreApplication a(argc, argv);
std::cout << "Comment vous appelez-vous ? ";
std::string nomUtilisateur;
std::cin >> nomUtilisateur;
std::cout << "Bonjour, ";
std::cout << nomUtilisateur;
std::cout << ". Je suis absolument ravi de faire votre connaissance !";
std::cout << "\nQuel age avez-vous ? ";
int ageActuel(0);
std::cin >> ageActuel;
```

CQT Exercices

Les lignes 13 à 16 n'ont rien de bien nouveau. Remarquez simplement que la variable `ageActuel` est initialisée avec la valeur 0 (ligne 14), une précaution judicieuse dans le cas des types standard.

Les types fournis par les librairies sont généralement assez sophistiqués pour qu'une variable qui vient d'être créée n'ait pas un contenu fantaisiste. Dans le cas des types standard, mieux vaut prendre soin de les initialiser explicitement avec une valeur significative. Ici, un age nul signifie clairement que l'utilisateur n'a pas (encore ?) répondu.

```
int anneeActuelle(0);
anneeActuelle = QDate::currentDate().year();
```

Comme son nom l'indique, la variable `anneeActuelle` est destinée à contenir l'année courante. Plutôt que d'indiquer littéralement cette valeur dans le texte source (ce qui conduirait à un programme ayant une date limite d'utilisation), nous obtenons la valeur correcte grâce à la librairie Qt (en espérant, évidemment, que l'ordinateur sur lequel le programme sera exécuté aura bien une date courante exacte). Cette utilisation de la librairie Qt exige la présence de la directive d'inclusion de la ligne 3.

En plus du recours à la librairie Qt, la ligne 17 fait intervenir une opération d'**affectation** : le signe = exige que la valeur fournie par Qt soit rangée dans la variable `anneeActuelle`.

Lorsqu'il figure à gauche de l'opérateur d'affectation (le signe =), le nom d'une variable désigne la variable elle-même, où il s'agit de ranger une nouvelle valeur (en oubliant l'ancienne).

La fin du programme ne devrait comporter aucune surprise pour vous, si ce n'est un petit raffinement à la ligne 20, où deux choses sont affichées par une même instruction : un texte littéral (placé entre guillemets) puis, grâce à un second double chevrons, la valeur d'une variable.

```
int anneeCentenaire(0);
anneeCentenaire = anneeActuelle + 100 - ageActuel;
std::cout << "Vous serez donc centenaire en " << anneeCentenaire;
return a.exec();
}
```

La ligne 19 procède, elle-aussi, à une affectation. La valeur qui doit être stockée dans la variable `anneeCentenaire` est obtenue par un calcul qui fait intervenir une addition (notée +) et une soustraction (notée -). Ce genre de calcul utilise évidemment les valeurs contenues dans les variables dont le nom est mentionné (`anneeActuelle` et `ageActuel`, dans cet exemple).

Si vous donnez à votre fichier `main.cpp` le contenu décrit ci-dessus, vous pourrez, une fois éliminées toutes les fautes de frappe, faire construire votre programme et jouer avec ☐.

6 - Corriger les fautes de frappe

Comme tous les langages de programmation, C++ est peu enclin à pardonner les approximations. Une simple erreur de ponctuation, un caractère omis ou une majuscule qui devient minuscule suffisent à déclencher une avalanche de protestations de la part de QtDesigner.

Même le plus maniaque des débutants passe son temps à faire des fautes de ce genre, qui sont une source importante de frustration.

Il n'y a pas de solution miracle : c'est seulement avec l'expérience que vient le privilège d'éviter le plus souvent (et de corriger très facilement) ces petites imperfections. Vient alors le plaisir de commettre des erreurs bien plus graves (et plus difficiles à corriger), mais ceci est une autre histoire...

En attendant, le seul remède vraiment efficace si vous n'arrivez pas à trouver ce qui empêche la création de votre programme, c'est de montrer votre texte source à quelqu'un qui a des yeux un peu plus entraînés : le forum est là pour ça !

7 - Exercice

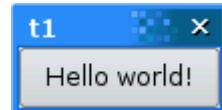
Créez un programme affichant dans une console le message "Nous sommes en 2011".

Cet affichage devra être effectué sans que l'utilisateur ait à effectuer une quelconque saisie.

Il va sans dire que, sans qu'il soit nécessaire de modifier le texte source et de reconstruire votre programme, celui-ci devrait être capable d'afficher "Nous sommes en 2012" s'il est exécuté l'année prochaine, puis "Nous sommes en 2013" s'il est exécuté l'année d'après, et ainsi de suite.

Remarque :

Il est maintenant demandé de transformer cette application en application graphique telle que



Bien sûr ce code est trivial

```
#include <QApplication>
```

CQT Exercices

```
#include <QPushButton>

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

    QPushButton hello("Hello world!");

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

Si vous utilisez la ligne de commande pour développer des applications Qt, vous devez vous assurer que les librairies et les exécutables de Qt sont accessibles à votre environnement en ajoutant le chemin du répertoire bin de Qt à votre variable PATH. Cette opération est décrite dans les instructions d'installation de votre plate-forme. Sous Windows, ceci est automatique si vous utilisez la fenêtre de commande depuis le menu Démarrer > (Tous les) Programmes > Qt. Si vous utilisez la fenêtre de commande depuis Démarrer > Exécuter > command ou cmd, vous devrez paramétrier vous-même la variable PATH ou utiliser le script du Qt SDK qtenv.bat,

```
qmake -project
qmake
make # ou nmake
```

La première commande ordonne à qmake de créer un fichier de projet (.pro). La seconde commande ordonne à qmake d'utiliser le fichier .pro pour créer un makefile adapté à la plateforme et au compilateur. Vous n'avez plus qu'à lancer make (nmake si vous êtes sous Visual Studio) pour compiler le programme et vous pourrez ensuite lancer votre première application Qt

CQT Exercices

Le programme que nous allons mettre au point au cours de cette seconde étape est un grand classique de l'initiation à la programmation : il tire un nombre au hasard et propose à l'utilisateur d'essayer de le deviner. Si ce jeu manque totalement d'intérêt, il va cependant nous fournir l'occasion d'aborder les structures de contrôles, qui permettent de créer des programmes qui ne sont pas de simples listes d'instructions exécutées l'une après l'autre, dans l'ordre où elles figurent dans la liste.

1 - Première version : en un coup

Créez, en suivant la procédure décrite pour l'étape 1, un projet de type <Application Qt 4 en console> □.

Tirer un nombre au hasard

La librairie standard propose une fonction qui fournit une valeur entière imprévisible. Pour illustrer son fonctionnement, donnez à votre fichier main.cpp le contenu suivant :

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <cstdlib>
#include <ctime>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    std::cout << rand() << "\n" << rand() << "\n" << rand();
    return a.exec();
}
```

Le texte "\n" représente une demande de passage à la ligne suivante et permet donc ici d'obtenir l'affichage des trois valeurs sur des lignes différentes. La succession de << permet d'obtenir des affichages successifs, exactement comme si on employait trois instructions d'affichage différentes.

Si vous n'avez pas oublié d'insérer, avant la fonction main(), les directives #include nécessaires, vous pouvez faire construire □, puis exécuter □ votre programme □.

Les nombres affichés donnent effectivement l'impression d'être aléatoires, mais que se passe-t-il si, après avoir refermé la fenêtre, vous demandez une nouvelle exécution du programme □ ?

Un ordinateur n'est pas réellement capable d'avoir un fonctionnement aléatoire, et les nombres fournis par la fonction rand() sont le résultat d'un calcul qui, pour donner des résultats "imprévisibles", a besoin d'un point de départ lui-même "imprévisible". Ce point de départ peut être fixé à l'aide d'une autre fonction de la librairie standard, et il est de coutume d'utiliser l'heure d'exécution (fournie par une troisième fonction) pour "amorcer" le générateur de nombres "imprévisibles".

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    srand(time(0));
    std::cout << rand() << "\n" << rand() << "\n" << rand();
    return a.exec();
}
```

Avant d'utiliser rand(), un programme devrait toujours appeler une fois (et une seule) srand().

Modifiez votre fonction main() comme suggéré ci-dessus, et exéutez plusieurs fois votre programme pour constater que les valeurs ont cessé de se répéter □.

Bien qu'on puisse maintenant les considérer "imprévisibles", les nombres fournis par rand() ont encore un aspect qui les rend impropres à l'usage que nous souhaitons en faire : ils peuvent être très grands, et il n'est pas réaliste d'attendre de l'utilisateur qu'il devine un nombre tiré d'un ensemble aussi vaste.

Pour ramener les valeurs tirées dans un intervalle plus raisonnable, nous allons utiliser une propriété mathématique élémentaire : dans une division entière, le reste est toujours strictement inférieur au diviseur. L'utilisation de cette propriété est rendue très facile par le fait que le langage C++ dispose d'un opérateur "reste de la division entière" (aussi appelé "modulo"), noté %. Si nous souhaitons obtenir une valeur imprévisible tirée, par exemple, de l'ensemble {0, 1, 2, 3, 4, 5}, nous pouvons écrire :

```
rand() % 6;
```

Si A et B sont deux nombres, A % B est le reste de la division de A par B.

Si nous choisissons de tirer un nombre inférieur à 32, le début de notre programme pourrait donc être :

CQT Exercices

```
int main(int argc, char *argv[])
{
    QCOREAPPLICATION a(argc, argv);
    srand(time(0));
    int tirage = rand() % 32;
    std::cout << "J'ai choisi un nombre entier positif plus petit que 32";
    std::cout << "\nEssayez de le deviner !\n\n";
    int proposition (-1);
    std::cout << "Proposez un nombre : ";
    std::cin >> proposition;
```

Le problème qui se pose alors est celui du verdict : l'utilisateur a-t-il proposé la bonne valeur ?

Afficher le verdict : l'exécution conditionnelle

Pour savoir s'il convient de consoler ou de féliciter le joueur, il faut comparer sa proposition à notre tirage. Le langage C++ propose six opérateurs permettant de comparer deux valeurs (cf. tableau ci-contre).

Ici, la victoire correspond au cas où la comparaison

```
proposition == tirage
```

se solde par une réponse affirmative.

| Opérateur | Signification |
|-----------|--------------------------------|
| == | égal à |
| != | non égal à (i.e. différent de) |
| < | inférieur à |
| <= | inférieur ou égal à |
| > | supérieur à |
| >= | supérieur ou égal à |

Le résultat d'une telle comparaison peut être utilisé pour décider quelles lignes du programme doivent être exécutées. On parle alors d'*exécution conditionnelle*, et cet effet peut être obtenu à l'aide des mots `if` et `else` (*si* et *sinon*). La fin de notre fonction `main()` prend donc la forme suivante :

```
if (proposition == tirage) {
    std::cout << "Bravo !";
} else {
    std::cout << "Dommage... Il fallait dire " << tirage;
}
return a.exec();
}
```

Le langage n'impose aucune règle de "mise en page" (passages à la ligne et indentations) du texte source. Vos chances de succès à moyen terme augmenteront cependant beaucoup si vous adoptez des règles facilitant la relecture et que vous les appliquez SYSTEMATIQUEMENT. Le style adopté dans les documents du cours est celui utilisé par les fonctions de mise en page automatique de QtCreator.

Remarquez que, à la ligne 5, la valeur tirée (puis ramenée dans l'intervalle souhaité) est stockée dans une variable. Le tirage n'a lieu qu'une seule fois, au moment où la ligne 5 est exécutée. Toutes les mentions ultérieures de `tirage` désignent donc l'unique valeur tirée. Ceci n'aurait pas été le cas si notre programme répétait l'expression `rand() % 32` aux lignes 11 et 14 : chaque évaluation de cette expression aurait donné un résultat différent, et le programme aurait été incohérent.

Donnez à votre fonction `main()` le contenu suggéré ci-dessus (lignes 1 à 17) et vérifiez que votre programme fonctionne ☐.

L'exécution conditionnelle peut être obtenue à l'aide du mot `if` suivi d'une paire de parenthèses entourant une expression dont on peut déterminer si elle est vraie ou fausse. Si (et seulement si) cette expression est vraie, le bloc d'instructions suivant (délimité par un couple d'accolades) est exécuté.

En cas de besoin, ce bloc de code peut être suivi du mot `else` et d'un second bloc d'instructions qui sera exécuté si et seulement si le premier bloc ne l'a pas été.

La présence du `else` et du second bloc de code est donc optionnelle. En leur absence, et dans le cas où l'expression de contrôle du `if` est fausse, le programme se poursuit normalement, comme si le `if` et son bloc d'instructions n'étaient pas là.

Une expression dont on peut dire si elle est vraie ou fausse est appelée une expression booléenne.

2 - Seconde version : en plusieurs coups

Cette première version du jeu est vraiment sans intérêt : l'utilisateur ne peut compter que sur un pur hasard et n'a qu'une chance sur 32 de gagner. Pour améliorer un peu les choses, notre seconde version va laisser l'utilisateur procéder à plusieurs tentatives et lui donner matière à réflexion.

CQT Exercices

Répéter la question

Une approche envisageable est de continuer à demander une proposition tant que l'utilisateur ne gagne pas. Ceci peut être obtenu à l'aide de l'instruction `do ... while()`, que l'on peut effectivement traduire par *répète ... tant que()*. Notre programme ressemblerait alors à ceci :

```
int main(int argc, char *argv[])
{
    QCOREApplication a(argc, argv);
    srand(time(0));
    int tirage = rand() % 32;
    std::cout << "J'ai choisi un nombre entier positif plus petit que 32";
    std::cout << "\nEssayez de le deviner !\n\n";
    int proposition = -1;
    do {
        std::cout << "Proposez un nombre : ";
        std::cin >> proposition;
    } while (proposition != tirage);
    std::cout << "Bravo !";
    return a.exec();
}
```

Remarquez que l'exécution conditionnelle réalisée précédemment à l'aide d'un `if else` n'est plus nécessaire : par définition, l'exécution du bloc de lignes [10-11](#) va être répétée jusqu'à ce que l'utilisateur finisse par trouver (ou par se lasser et refermer la fenêtre, mettant ainsi brutalement fin à cette mauvaise plaisanterie). La ligne [13](#) ne sera donc, quoi qu'il en soit, exécutée que si l'utilisateur a gagné. Elle peut donc le féliciter sans se poser de question.

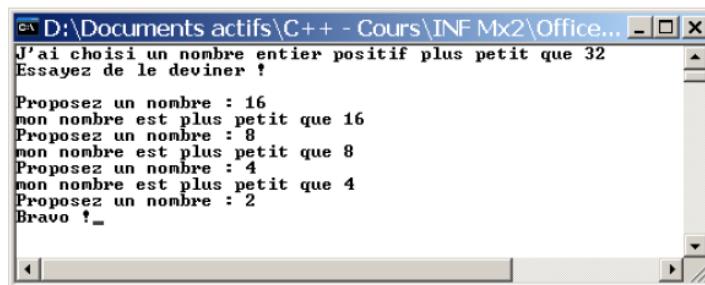
Donnez à votre fonction `main()` le contenu suggéré ci-dessus et vérifiez que votre programme fonctionne

La répétition de l'exécution d'un bloc d'instructions peut être obtenue le faisant précéder du mot `do` et suivre du mot `while` accompagné d'un couple de parenthèses entourant une expression booléenne.

Le bloc est exécuté une première fois, puis l'expression booléenne est évaluée et, si elle est vraie, l'exécution du programme "remonte" au `do`. Lorsque l'expression booléenne finit par être fausse, l'exécution du programme se poursuit normalement.

Fournir des indices

Pour commencer à ressembler vaguement à un jeu, notre programme doit permettre à l'utilisateur de faire autre chose que taper des nombres au hasard. On peut, par exemple, lui indiquer si le nombre recherché est plus grand ou plus petit que sa proposition :



Modifiez le programme pour qu'il se comporte ainsi

Vous savez dès à présent tout ce qu'il y a à savoir pour effectuer les modifications requises. Mais la logique des opérations est peut-être un peu plus complexe qu'elle en a l'air, et quelques tests pourraient bien révéler que votre premier programme syntaxiquement correct reste incorrect du point de vue de son comportement (au moment de la victoire ?)...

Remarque : modifier ce programme afin de prendre en compte une interface graphique de saisie et une interface graphique pour l'affichage des données.

CQT Exercices

3 - Exercices

1) Modifiez le programme pour que l'utilisateur puisse, au début du jeu, indiquer la valeur maximale autorisée pour le tirage.

```
Valeur maximale permise : 8  
J'ai choisi un nombre entier positif plus petit que 9  
Essayez de le deviner !  
Proposez un nombre : 4  
mon nombre est plus grand que 4  
Proposez un nombre : 6  
mon nombre est plus petit que 6  
Proposez un nombre : 5  
Bravo !
```

2) Modifiez le programme de façon à compter les propositions faites par l'utilisateur.

```
J'ai choisi un nombre entier positif plus petit que 32  
Essayez de le deviner !  
Proposez un nombre : 16  
mon nombre est plus petit que 16  
Proposez un nombre : 8  
mon nombre est plus grand que 8  
Proposez un nombre : 12  
mon nombre est plus petit que 12  
Proposez un nombre : 10  
mon nombre est plus grand que 10  
Proposez un nombre : 11  
Bravo : victoire en 5 coups !
```

3) Modifiez le programme de façon à pouvoir faire plusieurs parties sans avoir à le relancer.

```
J'ai choisi un nombre entier positif plus petit que 32  
Essayez de le deviner !  
Proposez un nombre : 16  
mon nombre est plus petit que 16  
Proposez un nombre : 8  
mon nombre est plus grand que 8  
Proposez un nombre : 10  
mon nombre est plus petit que 10  
Proposez un nombre : 9  
Bravo : victoire en 4 coups !  
  
Tapez 1 pour une autre partie, 0 pour stoper 1  
J'ai choisi un nombre entier positif plus petit que 32  
Essayez de le deviner !  
Proposez un nombre : 16  
mon nombre est plus grand que 16  
Proposez un nombre : 24  
mon nombre est plus petit que 24  
Proposez un nombre : 20  
mon nombre est plus petit que 20  
Proposez un nombre : 18  
mon nombre est plus grand que 18  
Proposez un nombre : 19  
Bravo : victoire en 5 coups !  
  
Tapez 1 pour une autre partie, 0 pour stoper
```

4) Modifiez le programme de façon à ce qu'il affiche le nombre moyen de propositions nécessaires à l'utilisateur pour trouver la solution.

```
Bravo : victoire en 4 coups !  
Sur 6 parties, votre score moyen est de 3.16667 coups  
  
Tapez 1 pour une autre partie, 0 pour stoper 1  
J'ai choisi un nombre entier positif plus petit que 32  
Essayez de le deviner !  
  
Proposez un nombre : 16  
mon nombre est plus grand que 16  
Proposez un nombre : 24  
mon nombre est plus petit que 24  
Proposez un nombre : 20  
mon nombre est plus petit que 20  
Proposez un nombre : 18  
Bravo : victoire en 4 coups !  
Sur 7 parties, votre score moyen est de 3.28571 coups
```

Remarque 1 : Nous allons maintenant continuer afin de terminer proprement l'application sur requête de l'utilisateur.



Il est demandé d'ajouter un bouton quitter pour stopper l'application correctement.

```
#include <QApplication>
```

CQT Exercices

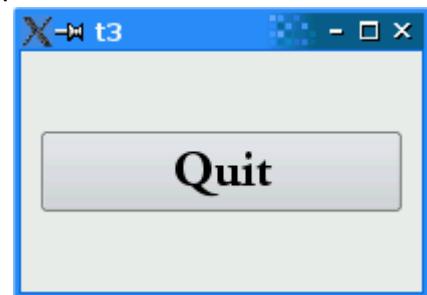
```
#include <QFont>
#include <QPushButton>

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

    QPushButton quit("Quit");
    quit.resize(75, 30);
    quit.setFont(QFont("Times", 18, QFont::Bold));

    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
    quit.show();
    return app.exec();
}
```

Remarque 2 : Cet exemple montre comment créer des widgets parents et enfants.



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

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

    QWidget window;
    window.resize(200, 120);

    QPushButton quit("Quit", &window);
    quit.setFont(QFont("Times", 18, QFont::Bold));
    quit.setGeometry(10, 40, 180, 40);
    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));

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

CQT Exercices

Le but de cette étape n'est pas de rendre programme "Devine un nombre" plus amusant, mais d'utiliser dans son code des techniques qui seront indispensables pour rédiger des programmes plus ambitieux.

1 - Le code obtenu à l'issue de l'étape 2

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <ctime>
#include <cstdlib>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    srand(time(0));
    int encore(1);
    double nbCoups(0);
    int nbParties(0);

    do {//boucle permettant de faire plusieurs parties
        int solution = (rand() % 32);
        nbParties = nbParties + 1;
        std::cout << "J'ai choisi un nombre entier positif plus petit que 32";
        std::cout << "\nEssayez de le deviner !\n\n";
        int proposition(-1);
        int nbCoupsPartie(0);

        do {//boucle de saisie des propositions
            std::cout << "Proposez un nombre : ";
            std::cin >> proposition;
            nbCoupsPartie = nbCoupsPartie + 1;
            if(proposition < solution)
                std::cout << "mon nombre est plus grand que " << proposition << "\n";
            if(proposition > solution)
                std::cout << "mon nombre est plus petit que " << proposition << "\n";
            } while(proposition != solution);

            std::cout << "Victoire en " << nbCoupsPartie << " coups\n";
            nbCoups = nbCoups + nbCoupsPartie;
            std::cout << "Votre moyenne est de ";
            std::cout << nbCoups/nbParties << " coups par partie\nEncore ?";
            std::cin >> encore;
        } while (encore == 1);

    return a.exec();
}
```

Si vous ne vous sentez pas capable d'écrire vous-même (ie. sans recopier un modèle) la fonction ci-dessus, vous devriez **travailler encore l'étape 2** plutôt que de chercher à passer à la suite.

Si vous rencontrez des difficultés, n'hésitez pas à vous manifester (forum, mail...).

Ne croyez surtout pas que ces difficultés vont disparaître par magie si vous les traitez par le mépris. Elles vont au contraire s'aggraver, parce que la suite de notre étude du langage suppose en permanence la maîtrise des mécanismes de base illustrés par cette version de "Devine un nombre" : création et utilisation de variables, boucles et tests.

Une dernière fois :

NE TOURNEZ PAS LA PAGE SI LE CODE CI-DESSUS VOUS SEMBLE MYSTERIEUX !

2 - Crédation d'une fonction

Si l'on examine le code de "Devine un nombre", on remarque qu'il s'agit fondamentalement d'une boucle enchaînée dans une autre : une partie est une répétition de séquences saisie-verdict, et cette répétition est elle-même répétée tant que l'utilisateur le souhaite.

Sur cette structure fondamentale se greffent des détails techniques tels que la création et l'initialisation des variables nécessaires et les tests qui permettent l'affichage du bon verdict. Une grande partie de la difficulté de mise au point du programme est justement d'arriver à gérer ces détails correctement à l'intérieur des deux boucles.

Lorsque nous allons chercher à écrire des programmes un peu plus complexes, cette façon de rédiger le code source va se traduire par des boucles à l'intérieur de boucles à l'intérieur de boucles à l'intérieur de boucles à l'intérieur....

De plus, le nombre des "détails techniques" qui devront être gérés correctement dans cet enchaînement vertigineux de boucles ne va cesser de croître.

Conclusion : ce n'est pas comme ça qu'il faut s'y prendre.

Un des outils que propose C++ pour dompter la complexité croissante des programmes est le découpage de ceux-ci en fonctions. Dans le cas présent, il est tentant de faire de la gestion d'une partie une fonction autonome. Il devient alors possible de cacher l'enchaînement des boucles : il suffit d'appeler la fonction en question depuis l'intérieur d'une boucle unique :

```
int main(int argc, char *argv[])
{
QCoreApplication a(argc, argv);
srand(time(0));
int encore(1);
double nbCoups(0);
int nbParties(0);
do {
    nbParties = nbParties + 1;
    int reponseFonction = jouePartieEtRenvoieNbCoups();
    nbCoups = nbCoups + reponseFonction;
    std::cout << "Victoire en " << reponseFonction << " coups\n";
    std::cout << "Votre moyenne est de " << nbCoups/nbParties;
    std::cout << " coups par partie\nEncore ?\n";
    std::cin >> encore;
} while (encore == 1);
return a.exec();
}
```

Cette façon de procéder exige évidemment que la fonction responsable du déroulement d'une partie soit elle aussi définie :

```
int jouePartieEtRenvoieNbCoups()
{
int solution = rand() % 32;
std::cout << "J'ai choisi un nombre entier positif plus petit que 32\n";
std::cout << "Essayez de le deviner !\n\n";
int proposition(-1);
int nbCoupsPartie(0);
do {
    std::cin >> proposition;
    nbCoupsPartie = nbCoupsPartie + 1;
    if(proposition < solution)
        std::cout << "mon nombre est plus grand que " << proposition << "\n";
    if(proposition > solution)
        std::cout << "mon nombre est plus petit que " << proposition << "\n";
} while(proposition != solution);
return nbCoupsPartie;
}
```

Une fonction peut renvoyer une réponse à l'aide de l'instruction return

Remarquez que l'en-tête de la fonction ([ligne 19](#)) mentionne le fait que l'exécution celle-ci s'achève en produisant une valeur de type `int`. Cette valeur est effectivement produite par la dernière instruction de la fonction ([34](#)). L'instruction qui appelle cette fonction ([10](#)) prend soin de recueillir cette valeur dans une variable pour pouvoir s'en servir ensuite ([11 et 12](#)).

CQT Exercices

3 - En pratique

Créez, en suivant la procédure décrite pour l'étape 1, un projet de type <Application Qt4 en console> □.

Donnez à votre fonction main() le contenu suggéré ci-dessus (*lignes 1-18*) □ et faites-la suivre de la définition de la fonction jouePartieEtRenvoieNbCoups() (*lignes 19-35*) □.

N'oubliez pas qu'une directive d'inclusion de iostream doit figurer en tête de votre fichier.

Déclaration et définition

Essayez de compiler votre programme □.

Le message d'erreur que vous obtenez indique que la fonction jouePartieEtRenvoieNbCoups() n'a pas été reconnue par le compilateur :

```
error: 'jouePartieEtRenvoieNbCoups' was not declared in this scope
```

En effet, lorsqu'il tente de traduire la ligne (10) qui appelle cette fonction, le compilateur n'en a encore jamais entendu parler.

Il serait ici possible de résoudre le problème en inversant, à l'aide d'un simple copier/coller, l'ordre des définitions de nos deux fonctions dans le fichier source.

En effet, si la définition de jouePartieEtRenvoieNbCoups() figure dans le texte source avant celle de main(), l'appel qui figure dans main() ne sera plus rencontré avant que la fonction concernée ne soit définie.

Cette approche n'est toutefois pas assez générale pour être satisfaisante, et nous allons donc plutôt adopter la méthode communément utilisée, qui consiste à placer en tête du fichier source une **déclaration** des fonctions qui y seront ensuite définies (à l'exception de main(), qui ne doit jamais être déclarée explicitement). Cette déclaration prend exactement la forme de l'en-tête de la fonction concernée et doit être suivie d'un point virgule. Notre fichier source adopte donc le plan suivant :

```
#include <QtCore/QCoreApplication>
#include <iostream>

int jouePartieEtRenvoieNbCoups(); //déclaration de cette fonction
//*****
int main()
{
//corps de la fonction main()
}
//*****
int jouePartieEtRenvoieNbCoups()
{
//corps de la fonction jouePartieEtRenvoieNbCoups()
}
```

Séparer clairement les définitions des fonctions les unes des autres (en faisant, par exemple, précéder chacune d'entre-elles d'une ligne d'étoiles) facilite grandement la lecture du fichier source, et donc la mise au point du programme.

Rendez le programme compilable en adoptant le plan suggéré ci-dessus □.

Utilisation d'un fichier d'en-tête

Dans la plupart des cas, les déclarations nécessaires ne figurent pas *directement* en tête du fichier source, mais y sont "injectées" à l'aide d'une directive d'inclusion. Bien qu'un peu plus lourde à mettre en place, cette façon de procéder s'avère infiniment plus pratique lorsque certaines fonctions en appellent d'autres *qui ne sont pas définies dans le même fichier source*.

L'éclatement du programme en plusieurs fichiers source devient inéluctable lorsque le volume de code s'accroît un tant soit peu. Un cas voisin est celui des fonctions qui ne sont pas définies dans un fichier source (ie. un texte en C++) mais dans une librairie (ie. sous la forme de code déjà compilé). C'est pourquoi nous avons déjà rencontré à plusieurs reprises des directives d'inclusion.

Dans le Menu <Fichier>, sélectionnez la commande <Nouveau fichier ou projet...> □.

Dans la boîte de dialogue qui s'ouvre alors, sélectionnez <C++> et <Fichier Header C++>, puis cliquez sur [Ok] □.

Le dialogue suivant permet de baptiser le fichier. Donnez lui pour nom "declarationsDeMesFonctions" et cliquez sur [Suivant], puis sur [Terminer] dans la fenêtre suivante □.

CQT Exercices

Ne donnez jamais à vos projets ou fichiers des noms comportant des espaces ou des minuscules accentuées, même si votre système d'exploitation l'autorise.

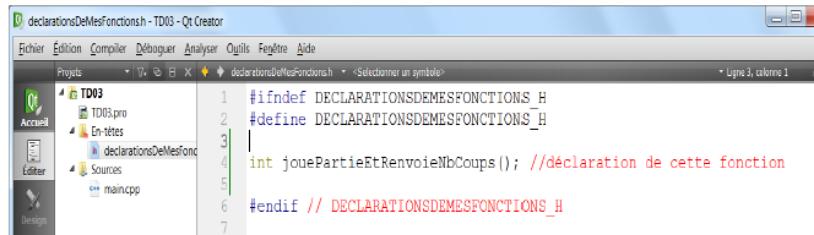
QtCreator fait appel à des logiciels vénérables, développés à l'origine sous des systèmes qui n'envisageaient pas l'usage d'une langue autre que l'anglais. Ces contraintes sont le prix qu'il faut actuellement payer pour utiliser ces outils gratuits.

Dans le dernier dialogue, vérifiez que l'option <Ajouter au projet> est bien cochée et cliquez sur [Terminer].

L'éditeur ouvre alors le nouveau fichier, dans lequel QtCreator a placé pour vous quelques lignes destinées à vous défendre contre des cas particuliers difficiles à repérer et à gérer. Remerciez-le mentalement et souvenez vous que :

Dans un fichier d'en-tête, il faut toujours placer les déclaration entre les directives `#define` et `#endif`.

Donnez donc à votre fichier le contenu suivant :



The screenshot shows the Qt Creator interface with the project 'declarationsDeMesFonctions - TD03 - Qt Creator'. The 'declarationDeMesFonctions.h' file is open in the editor. The code contains a preprocessor directive '#ifndef DECLARATIONSDEMESFONCTIONS_H' followed by '#define DECLARATIONSDEMESFONCTIONS_H', an empty line, and then the declaration of a function 'int jouePartieEtRenvoieNbCoup(); //déclaration de cette fonction'. Finally, there is an '#endif // DECLARATIONSDEMESFONCTIONS_H' directive. The code is color-coded with syntax highlighting.

Remarquez que QtCreator a attribué à votre fichier l'extension ".h", ce qui correspond à la tradition concernant les fichiers d'en-tête en C++ (en-tête = header, en anglais).

Dans votre fichier main.cpp, remplacez la déclaration de la fonction par une directive d'inclusion :

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include "declarationsDeMesFonctions.h"
//*****
int main()
{
//corps de la fonction main()
}
//*****
int jouePartieEtRenvoieNbCoup()
{
//corps de la fonction jouePartieEtRenvoieNbCoup()
}
```

Vérifiez que votre programme reste compilable en dépit des faits que main.cpp ne comporte plus de déclaration directe de la fonction `jouePartieEtRenvoieNbCoup()` et que celle-ci y est appelée (par `main()`) avant d'y être définie.

4 - Exercice

Dans sa version actuelle, le programme n'effectue aucune vérification sur la saisie effectuée par l'utilisateur, qui peut très bien être négative, supérieure à 32, ou même ne pas être un nombre. En cas de faute de frappe, un essai échoué est donc comptabilisé dans la partie, au détriment du joueur.

Créez une fonction `saisieProposition()` à laquelle la fonction `jouePartieEtRenvoieNbCoup()` fera appel (à la ligne 27, en lieu et place de l'instruction `std::cin >> proposition;`).

Cette fonction devra s'assurer que la saisie est bien un nombre positif inférieur à 32 et, dans le cas contraire, devra exiger une nouvelle saisie (en affichant de préférence un message pour indiquer à l'utilisateur ce qui ne va pas). Cette fonction ne renverra donc que des propositions vraisemblables.

Cet exercice porte sur la création (déclaration et définition) d'une fonction et sur son utilisation (appel). Ne cherchez pas à sophistiquer outre mesure les tests de validité et les messages utilisés. Renoncez, en particulier, à l'idée de gérer élégamment les saisies non numériques (c'est un problème plus complexe qu'il en a l'air).

CQT Exercices

Nous pouvons désormais envisager la création de programmes dont l'interface utilisateur présente l'aspect attendu par nos contemporains : menus, boutons, icônes etc. L'essentiel du travail de programmation nécessaire a déjà été fait (par les programmeurs de Trolltech) et est mis à notre disposition sous la forme d'une vaste collection de classes (la librairie Qt).

Notre premier programme utilisant une interface graphique sera une nouvelle version de "Devine un nombre". Notre expérience antérieure concernant l'écriture de ce programme va nous permettre de nous concentrer sur les spécificités imposées par l'interface graphique.

1 - Structure d'un projet graphique

Plutôt que de greffer une interface graphique sur un projet console (comme nous l'avons fait lors de l'étape précédente), nous allons créer directement un projet de programme à interface graphique.

Création du projet

Dans le menu <Fichier>, choisissez la commande <Nouveau fichier ou projet...> □.

Dans la liste des types de projets proposés, choisissez <Application graphique Qt4> (cf. ci-contre) □.

Choisissez, comme d'habitude, un nom et un emplacement pour votre projet □.

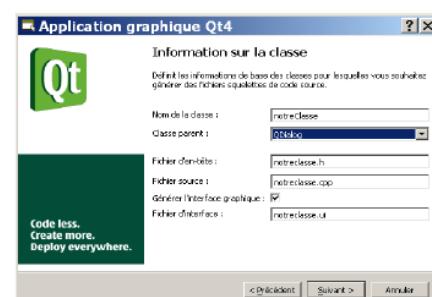
Remarquez que, dans la "Liste des modules requis", le module "QtGui" que nous avions du cocher nous-mêmes lors de l'étape 7 est maintenant obligatoire □.

La création d'un projet graphique implique une étape supplémentaire, car l'interface de notre programme va être décrite par une classe dont il nous faut maintenant choisir le nom et la nature.

Dans le champ "Nom de la classe :", tapez notreClasse □.

Dans la liste "Classe parent :", choisissez QDialog □.

Nous allons, certes, indiquer certaines des propriétés que nous souhaitons voir adopter par notreClasse. Mais nous n'avons pas l'ambition de créer cette classe ex nihilo. Nous allons nous appuyer sur une classe fournie par Qt, à laquelle nous allons simplement ajouter les spécificités qui nous intéressent. Plusieurs points de départ sont envisageables, et le choix proposé par défaut (QMainWindow) conduit à un projet un peu trop complexe pour ce que nous entendons faire. C'est la raison pour laquelle nous nous contenterons de créer un QDialog.



La fonction main()

Par rapport aux projets "console" auxquels nous sommes habitués, un projet graphique se singularise par le fait que la fonction main() instancie notreClasse (4), puis exécute une fonction membre au titre de l'instance créée (5). La fonction show() ordonne au dialogue de dessiner sa fenêtre sur l'écran, de façon à ce que l'utilisateur puisse manipuler les éléments d'interface qu'elle propose.

La fonction exec(), pour sa part, attend que l'utilisateur mette fin au programme (en cliquant sur la case de fermeture de la fenêtre, par exemple), tout en veillant à ce que ses actions soient prises en compte.

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    notreClasse w;
    w.show();
    return a.exec();
}
```

Dans un projet graphique, il n'est souvent pas nécessaire de modifier la fonction main(). Nos efforts vont plutôt viser à modifier notreClasse, de façon à ce que l'appel de la fonction exec() se traduise par le comportement que nous attendons de notre programme.

Les lignes 4 et 5 de la fonction main() sont deux les seules lignes du programme qui utilisent notreClasse. Tout le code que nous allons écrire ne fera que définir notreClasse.

2 - Dessin du dialogue

QtCreator nous propose deux façons de modifier notreClasse.

La première est très classique : nous pouvons simplement modifier les fichiers notreClasse.h et notreClasse.cpp pour ajouter à notre guise des variables et des fonctions membre, exactement comme nous l'avons fait pour la classe CEtudiant lors de l'étape 7.

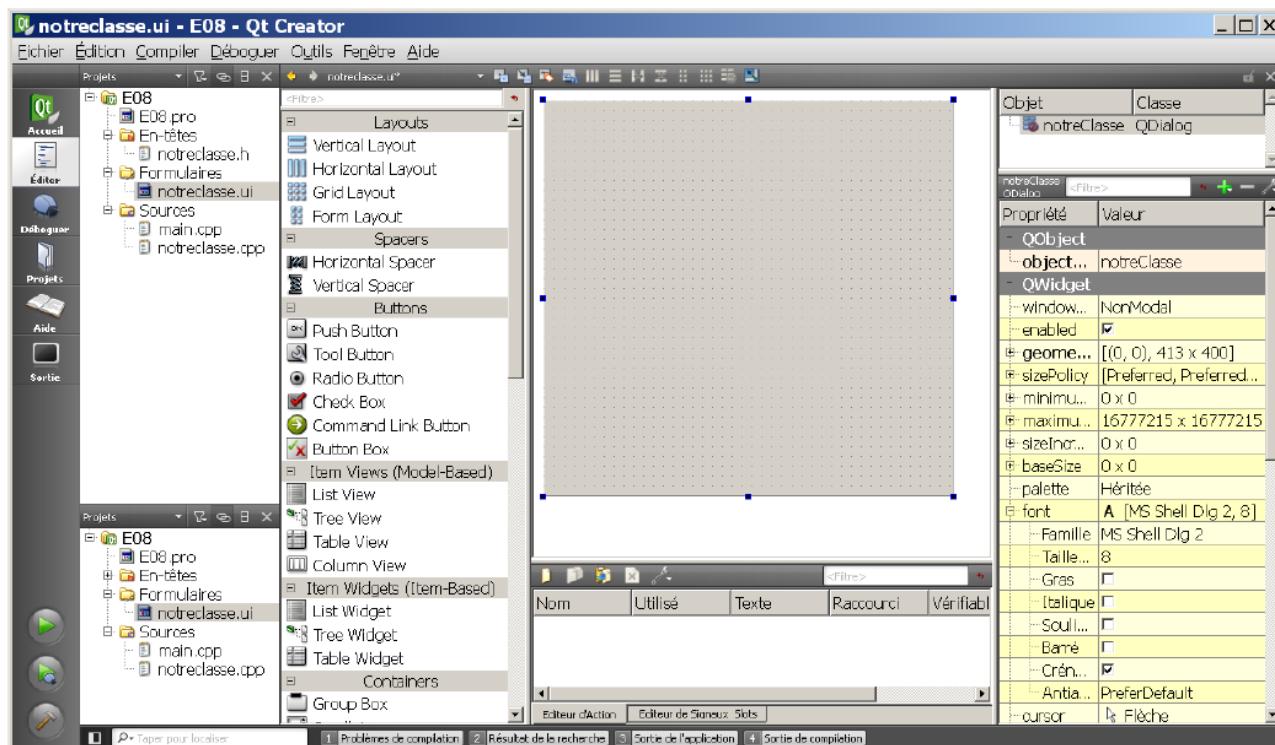
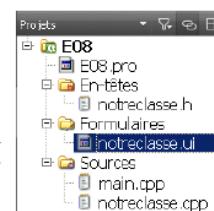
La seconde est beaucoup plus originale : nous disposons d'un éditeur graphique qui nous permet de spécifier l'apparence que nous souhaitons donner à notre dialogue.

Le dessin que nous produirons à l'aide de l'éditeur graphique donnera automatiquement naissance à un texte source en C++. La compilation de ce texte générera du code dont l'exécution se traduira par l'apparition d'une fenêtre exactement conforme à notre dessin.

L'éditeur graphique

Les projets graphiques comportent un fichier qui n'est ni un .h ni un .cpp, mais un .ui (acronyme de **u**ser **i**nterface). Ces fichiers sont rangés dans la catégorie "Formulaires" (cf. ci-contre). Ouvrez le fichier notreclasse.ui en double cliquant sur son nom ☐.

L'éditeur graphique apparaît alors à l'écran. Il s'agit d'un environnement de travail fortement multi-fenêtré et très configurable, manifestement conçu et optimisé pour des stations de travail disposant d'un écran de grande taille. Il se présente typiquement sous la forme suivante :



La surface grise occupant ici la place centrale est le "fond" de la fenêtre sur lequel nous allons disposer les différents éléments qui vont composer l'interface de notre programme.

La liste qui figure ici à gauche propose des éléments d'interface (widgets). L'idée générale est de venir piocher dans cette liste et de faire glisser sur le fond les widgets dont nous avons besoin.

Les autres fenêtres proposent différentes informations dont la nature dépend du widget sélectionné.

Si votre écran s'avère trop petit pour afficher confortablement toutes les fenêtres représentées ci-dessus, vous pouvez refermer certaines d'entre elles. La "barre latérale", en particulier (ie. la fenêtre qui affiche la liste des fichiers) peut être ouverte ou fermée à l'aide du menu <Fenêtre> (commande <Afficher la barre latérale>).

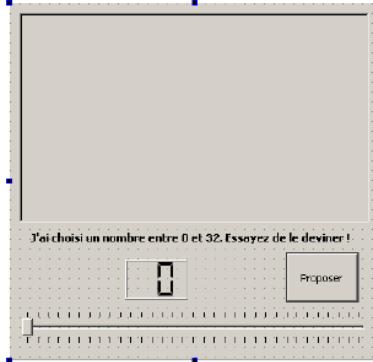
CQT Exercices

Mise en place des widgets

L'interface de notre programme comporte cinq widgets :

- un **TextEdit** qui servira à afficher les messages ("trop grand", "trop petit", "bravo !", etc.)
- un **Label** qui rappelle les règles du jeu
- un **LCDNumber** qui affichera la valeur envisagée par l'utilisateur
- un **PushButton** qui permettra de soumettre la valeur envisagée au verdict du programme
- un **Horizontal Slider** qui permettra de modifier la valeur envisagée.

Positionnez ces cinq widgets et ajustez leur taille de façon à reproduire approximativement le dialogue représenté ci-contre ☐.



Veuillez à bien faire glisser le bon type de widget.

Un **TextEdit**, par exemple, **n'est pas** un **PlainTextEdit** ou un **LineEdit**...

Pour modifier l'aspect d'un widget, sélectionnez-le et modifiez la propriété concernée dans la fenêtre qui présente des couples propriété/valeur sur des lignes alternativement plus ou moins teintées.

Le code que nous allons écrire suppose que l'**Horizontal Slider** a pour **objectName** **proposition** et pour **maximum** **32**. Modifiez ces deux propriétés pour leur conférer ces valeurs ☐.

Pour modifier une propriété d'un widget, **il faut commencer par le sélectionner**. Veuillez à ne **JAMAIS** changer l'objectName du dialogue lui-même (sa valeur doit rester **notreClasse**).

Vous pouvez aussi modifier les propriétés suivantes (ces propriétés ne sont pas nécessaires au fonctionnement du programme, mais améliorent son apparence) :

PushButton : la propriété **text** vaut "Proposer"

TextEdit : la propriété **enabled** est **décochée** (pour empêcher l'utilisateur d'y insérer du texte).

Label : la propriété **font/Gras** est **cochée** (et le **text** est évidemment modifié !)

LCDNumber : **numDigits** vaut **2** et **segmentStyle** est **flat**

Horizontal Slider : **tickPosition** vaut **TicksBothSides** (pour faire apparaître les graduations).

Connexions

Maintenant que l'aspect visuel de l'interface est spécifié, il convient de commencer à s'intéresser au comportement du programme. Deux phénomènes nous concernent :

- lorsque l'utilisateur déplace le curseur, la valeur affichée par le **LCDNumber** doit changer.
- lorsque l'utilisateur clique sur le bouton, une fonction que nous allons écrire doit être exécutée.

C'est cette fonction qui devra procéder aux comparaisons et affichages qui composaient l'essentiel de notre version "console" du programme "Devine un nombre".

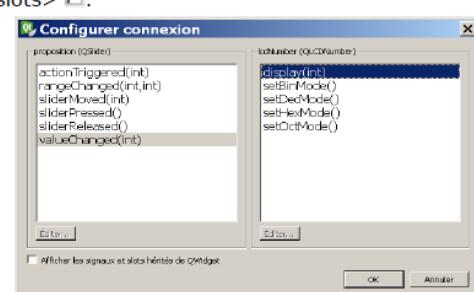
La liaison entre la position du curseur et la valeur affichée par le **LCDNumber** ne nécessite aucune écriture de code : il s'agit simplement d'indiquer qu'un événement (le déplacement du curseur) doit déclencher un autre (la modification de la valeur affichée). Les widgets détectent certains événements les concernant, et l'éditeur graphique permet d'établir directement ce type de connexion.

Dans le menu <Edition>, choisissez la commande <Editer signaux/slots> ☐.

Pour Qt, les **signaux** d'un widget sont des fonctions qui sont appelées automatiquement lorsque survient un événement qui concerne ce widget.

Cliquez sur le **Horizontal Slider** et, tout en tenant le bouton de la souris enfoncé, faites glisser son pointeur vers le **LCDNumber**. Lorsque ce dernier devient rouge, relâchez le bouton de la souris ☐.

Le dialogue représenté ci-contre apparaît alors.



CQT Exercices

Les deux événements qui nous intéressent sont le changement de la valeur correspondant à la position du curseur et le changement de l'affichage proposé par le LCDNumber.

Le premier de ces événements se traduit par l'émission du signal `valueChanged()` par le curseur, alors que le second sera causé par l'exécution de la fonction `display()` au titre du LCDNumber.

Sélectionnez ces deux fonctions et cliquez sur [OK] .

Ce que nous venons de dire, c'est que lorsque la fonction `valueChanged()` est exécutée au titre de proposition, elle doit appeler la fonction `display()` au titre du LCDNumber.

Pour Qt, les fonctions qui peuvent être liées à des signaux s'appellent des **slots**.

Nous venons donc de connecter le signal `valueChanged()` d'un `QHorizontalSlider` au slot `display()` d'un `QLCDNumber`. Remarquez que ces deux fonctions disposent d'un paramètre, qui permet à `valueChanged()` de recevoir la valeur correspondant à la position du curseur et, le moment venu, de la transmettre à `display()`.

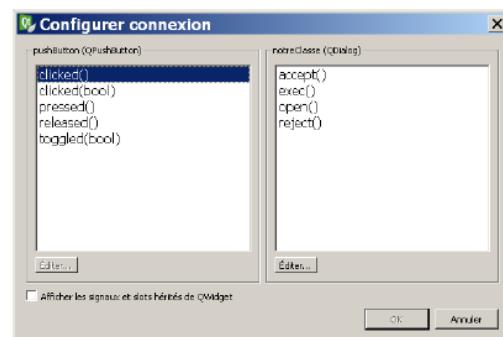
Nous devons maintenant connecter le bouton [Proposer] à une fonction... qui n'existe pas encore puisque nous devrons l'écrire. Nous allons donc simplement indiquer comment elle s'appellera, et QtCreator s'arrangera pour la trouver lorsqu'il en aura besoin.

Cliquez sur le pushButton [Proposer] et, tout en maintenant le bouton de la souris enfoncé, faites glisser son pointeur en dehors du pushButton, sur le fond du dialogue. Relâchez alors le bouton de la souris .

Le signal qui nous intéresse est évidemment celui qui est émis lorsque l'utilisateur clique sur le bouton (la fonction `clicked()`), mais aucun des slots proposés ne correspond au traitement que nous attendons (Qt ne comporte aucune fonction spécifiquement prévue pour jouer à "Devine un nombre").

Il faut donc que nous annonçons que notre programme va comporter une fonction nommée `f_proposser()`.

Cliquez sur le bouton [Editer] .



Dans le dialogue suivant (cf. ci-contre), cliquez sur le symbole "+" de la zone "Slots" .

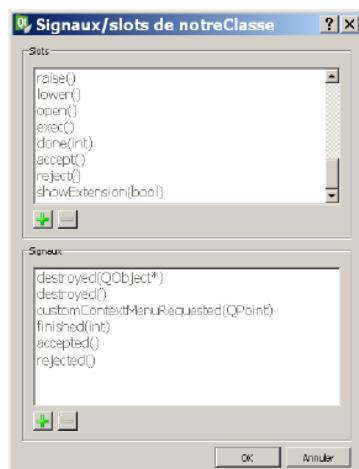
Un nouveau slot est créé. Donnez-lui pour nom "`f_proposser()`" .

La liste de slots proposée par le dialogue de connexion comporte maintenant notre nouvelle fonction. Sélectionner-la (ainsi que l'événement `clicked()`, si ce n'est déjà fait) et cliquez sur le bouton [OK] .

Nous venons de connecter le signal `clicked()` d'un `QPushButton` au slot `f_proposser()` de `notreClasse`.

L'éditeur graphique nous propose, dans l'onglet "Editeur de signaux/slots", un résumé des connexions établies :

| Émetteur | Signal | Recepteur | Slot |
|-------------|--------------------------------|--------------------------|----------------------------|
| proposition | <code>valueChanged(int)</code> | <code>LcdNumber</code> | <code>display(int)</code> |
| pushButton | <code>clicked()</code> | <code>notreClasse</code> | <code>f_proposser()</code> |



Cette connexion était la dernière opération nécessitant l'éditeur graphique. Revenez à l'éditeur de texte en double-cliquant sur "notreClasse.h" dans la barre latérale .

Si vous avez masqué celle-ci pour gagner de la place, il faut la faire réapparaître (menu <Fenêtre>).

3 - Écriture du programme

Il reste maintenant à écrire les fonctions membre qui effectueront les traitements nécessaires au fonctionnement du programme. La principale d'entre-elles est évidemment celle qui sera appelée à chaque clic sur le bouton [Proposer], le slot `f_proposser()`.

Déclaration de la fonction `f_proposser()`

Il s'agit d'une fonction membre de `notreClasse`, et il faut donc que sa déclaration soit présente dans la définition de cette classe. Il s'agit également d'un slot, et ceci doit être précisé en faisant figurer la déclaration dans une section spécialement prévue à cet effet :

```
class notreClasse : public QDialog {
    Q_OBJECT
public:
    notreClasse(QWidget *parent = 0);
    ~notreClasse();
public slots:
    void f_proposser();
protected:
    void changeEvent(QEvent *e);
private:
    Ui::notreClasse *ui;
};
```

Ajoutez les lignes 6 et 7 à la définition de `notreClasse` □.

Les titres `public:`, `public slots:` et `protected:` délimitent des sections dans la classe. Si vous devez ajouter d'autres slots, il est inutile de répéter le titre `public slots:`, il suffit de placer la déclaration de ces fonctions dans la bonne section (ie. entre `public slots:` et le titre suivant).

Définition de la fonction `f_proposser()`

La première chose que doit faire cette fonction est de récupérer la valeur sélectionnée par l'utilisateur au moyen du curseur. Cette opération est réalisée en appelant une fonction membre de la classe `QHorizontalSlider` au titre de l'objet qui représente ce curseur.

Pour éviter de mêler les variables qui représentent des widgets issus du dessin produit avec l'éditeur graphique aux autres variables, QtCreator crée une classe, l'instancie et ajoute à `notreClasse` une variable membre nommée `ui` qui permet d'accéder à nos widgets (cf. ligne 11 ci-dessus).

Si un widget a été nommé `truc` dans l'éditeur graphique, on le désigne par `ui->truc` dans le code.

La fonction membre de `QHorizontalSlider` qui nous intéresse ici s'appelle `value()`. Elle renvoie tout simplement la valeur qui correspond à la position du curseur au titre de laquelle elle est appelée :

```
void notreClasse::f_proposser()
{
    //on récupère la proposition faite par le joueur
    int valeurProposee = ui->proposition->value();
```

Une fois cette valeur récupérée dans `valeurProposee`, nous allons devoir la comparer à notre tirage et afficher nos conclusions. Cet affichage va utiliser le `QTextEdit` que nous avons placé dans notre dialogue et baptisé `verdict`. Il est très facile d'afficher du texte dans un `QTextEdit`, à condition que ce texte soit stocké dans une `QString`.

Tout ce que vous savez des `std::string` s'applique directement au `QString`

Il aurait sans doute été préférable de ne jamais parler de `std::string` et d'utiliser directement des `QString`. Malheureusement, les flux standard qui permettent d'utiliser la console (`std::cin` et `std::cout`) ignorent tout de Qt et, donc, de la classe `QString`.

Après avoir créé une `QString` nommée `aAfficher` (4), le programme utilise une autre classe de la librairie Qt. Cette classe, nommée `QTextStream`, ressemble beaucoup aux flux `std::cin` et `std::cout`, mais offre une possibilité tout à fait originale : on peut choisir la destination réelle (ou la provenance) du texte inséré dans un (ou extrait d'un) `QTextStream` à l'aide de l'opérateur `<<` (ou de l'opérateur `>>`).

CQT Exercices

Cette destination peut, par exemple, être une imprimante, un fichier ou, ce qui est plus pertinent ici, une variable de type `QString`.

```
//on prépare une QString déguisée en std::cout
QString aAfficher;
QTextStream out(&aAfficher);
```

Une fois le `QTextStream` "branché" sur notre `QString` (5), on peut l'utiliser d'une façon qui devrait vous paraître familière (6-11) :

```
//calcul du message
if(valeurProposee < m_tirage)
    out << valeurProposee << " est trop petit";
if(valeurProposee > m_tirage)
    out << valeurProposee << " est trop grand";
if(valeurProposee == m_tirage) {
    out << "Bravo ! Je tire un nouveau nombre...";
    prepareNouvellePartie(); //tirage d'une nouvelle valeur et autres préparatifs
}
```

Lorsque la partie s'achève, le programme procède immédiatement à un nouveau tirage. Cette tâche est déléguée à une fonction `prepareNouvellePartie()`, qui devra être aussi appelée en début de programme (pour préparer la première partie) et pourra éventuellement remettre à zéro des compteurs de coups ou de parties utilisés dans une version future du programme.

Il ne reste plus à la fonction `f_proposer()` qu'à afficher le texte contenu dans la `QString`. Les `QTextEdit` proposent une fonction `append()` qui ajoute la chaîne qui lui est passée à la fin du texte précédemment affiché par le widget :

```
//affichage du message dans le QTextEdit
ui->verdict->append(aAfficher);
}
```

Ajoutez à votre fichier `notreClasse.cpp` la définition de la fonction `f_proposer()` décrite ci-dessus □.

L'usage d'un `QTextEdit` exige l'ajout d'une directive `#include <qtextedit>` en tête de fichier.

La variable `m_tirage`

Cette variable devant contenir la solution de la devinette, elle est évidemment de type `int`.

Nous savons que sa valeur doit être fixée par `prepareNouvellePartie()` et utilisée par `f_proposer()`.

Celle-ci ne dispose d'aucun paramètre et n'est pas appelée par `prepareNouvellePartie()` : `m_tirage` ne peut donc être une variable locale à `prepareNouvellePartie()` dont la valeur serait transmise à `f_proposer()` lors de l'appel de cette dernière.

Nos deux fonctions sont toutefois membre de `notreClasse`. Elles peuvent donc toutes deux accéder aux variables membre de l'instance au titre de laquelle elles sont exécutées.

Dans notre cas, il n'existe qu'une instance de `notreClasse` : la variable `w` définie dans `main()`. C'est au titre de cette variable que `main()` appelle `show()`, c'est donc à cette variable qu'appartiennent les widgets qui apparaissent à l'écran. Lorsque l'utilisateur clique sur le bouton [Proposer], il est donc logique que le slot `f_proposer()` soit appelé au titre de `w`. Lorsque `f_proposer()` appelle à son tour `prepareNouvellePartie()`, c'est donc (implicitement) également au titre de `w` (12).

La variable `m_tirage` doit donc être membre de `notreClasse`. Ajoutez sa déclaration dans le fichier `notreClasse.h` □.

Une variable membre n'est pas un slot (un slot est une fonction !) et la déclaration de `m_tirage` ne doit donc pas être placée dans la section `public slots:`. Placez-la plutôt dans la section `protected:` (la section `public:` n'a normalement pas vocation à contenir des déclarations de variables).

Définition et appel de la fonction `prepareNouvellePartie()`

Le rôle de cette fonction est très simple et son code se passe de commentaire :

CQT Exercices

```
void notreClasse::prepareNouvellePartie()
{
    m_tirage = rand() % 32;
}
```

Ajoutez cette fonction à notreClasse □.

Cette opération exige d'intervenir dans notreClasse.cpp **et** dans notreClasse.h

Un dernier détail doit cependant être réglé. Cette fonction doit en effet être exécutée une première fois lors du lancement du programme.

Une première façon de procéder serait d'ajouter un appel à cette fonction dans main() :

```
w.prepareNouvellePartie();
```

La fonction main() ne faisant pas partie de notreClasse, elle n'est pas exécutée au titre d'une instance de celle-ci. Elle ne peut donc accéder à un membre de notreClasse (ici, la fonction nouvellePartie()) qu'en indiquant explicitement au titre de quelle instance cet accès doit être fait (ici, la variable w).

Il n'est toutefois pas conseillé de confier la responsabilité de la "mise en état de marche" d'une instance au programmeur qui crée cette instance (et qui ne connaît pas forcément tous les détails de fonctionnement interne de la classe). On préfère donc effectuer ce genre d'opérations dans une fonction membre spéciale, qui est exécutée automatiquement lors de l'instanciation. Cette fonction s'appelle un **constructeur**, et QtCreator l'a déjà préparée pour nous dans le fichier notreClasse.cpp :

```
notreClasse::notreClasse(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::notreClasse)
{
    ui->setupUi(this);
    prepareNouvellePartie();
}
```

Ajoutez la ligne 6 au constructeur de notreClasse □.

Initialisation du générateur de nombres pseudo-aléatoires

Si nous ne voulons pas retrouver la même séquence de nombres à chaque lancement du programme, nous devons faire en sorte que srand() soit exécutée **une fois et une seule** avant le début de la première partie.

Placer l'appel à srand() dans le constructeur de notreClasse ne garantirait pas absolument que cette instruction ne sera exécutée qu'une seule fois. En effet, le constructeur va être exécuté lors de chaque instanciation de notreClasse. Notre programme n'en comporte pour l'instant qu'une seule (dans main(), nous l'avons vu), mais rien ne prouve qu'il en sera toujours ainsi.

Il est donc préférable d'insérer l'appel à srand() dans la fonction main() □ :

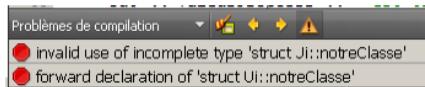
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    srand(time(0));
    notreClasse w;
    w.show();
    return a.exec();
}
```

L'appel à time() peut nécessiter une directive #include <ctime>

L'appel à srand() peut nécessiter une directive #include <cstdlib>

Le programme est terminé, vous pouvez le compiler et jouer avec □.

Si le compilateur émet des protestations telles que celles représentées ci-contre, ouvrez le fichier notreClasse.ui et vérifiez que vous n'avez pas changé par inadvertance l'objectName du dialogue (qui doit être "notreClasse").



4 - Exercice

Sauriez-vous doter cette version de "Devine un nombre" des fonctionnalités "avancées" que nous avions prévues pour la version console (compter le nombre de parties, faire la moyenne du nombre de coups par partie, etc) ?

Vous pouvez commencer par afficher ces informations dans le textEdit. Une fois réglés les problèmes de création de variables et de calcul, vous pouvez essayer de rajouter des widgets dans le dialogue pour obtenir un affichage plus conforme aux normes visuelles actuelles (des LCDNumber, par exemple).