Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator
Etape 8 : Un dialogue comme interface utilisateur

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.

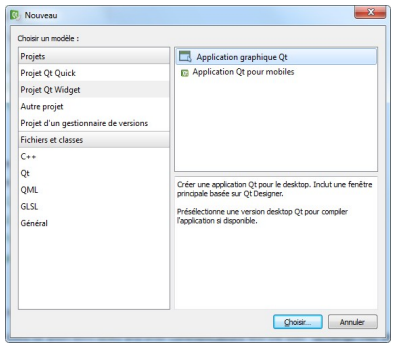
1 - Structure d'un projet graphique

La création d'un projet graphique s'accompagne de celle d'une classe destinée à représenter l'interface utilisateur. Cette interface sera dessinée à l'aide d'un outil spécifique (QtDesigner), qui génère automatiquement une description du dessin réalisé. Lors de son instanciation, la classe représentant l'interface utilise cette description pour disposer les différents éléments (widgets) composant l'interface.

Création du projet

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

Dans la liste des types de projets proposés, choisissez <Projet Qt Widget> et <Application graphique Qt> (cf. ci-contre) ☐.

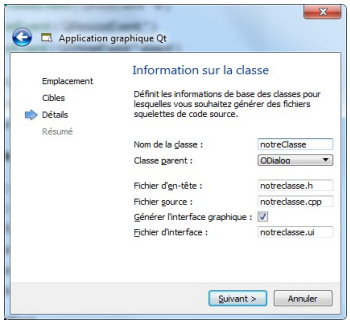


Choisissez, comme d'habitude, un nom et un emplacement pour votre projet ☐.

La création d'un projet graphique implique une étape supplémentaire, car il nous faut maintenant choisir le nom et la nature de la classe qui sera créée pour représenter l'interface.

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

Dans la liste "Classe parent :", choisissiez `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.

```
1 int main(int argc, char *argv[])
2 {
3 QApplication a(argc, argv);
4 notreClasse w;
5 w.show();
6 return a.exec();
7 }
```

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

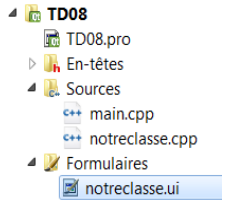
QtCreator nous propose deux façons de modifier `notreClasse`.

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.

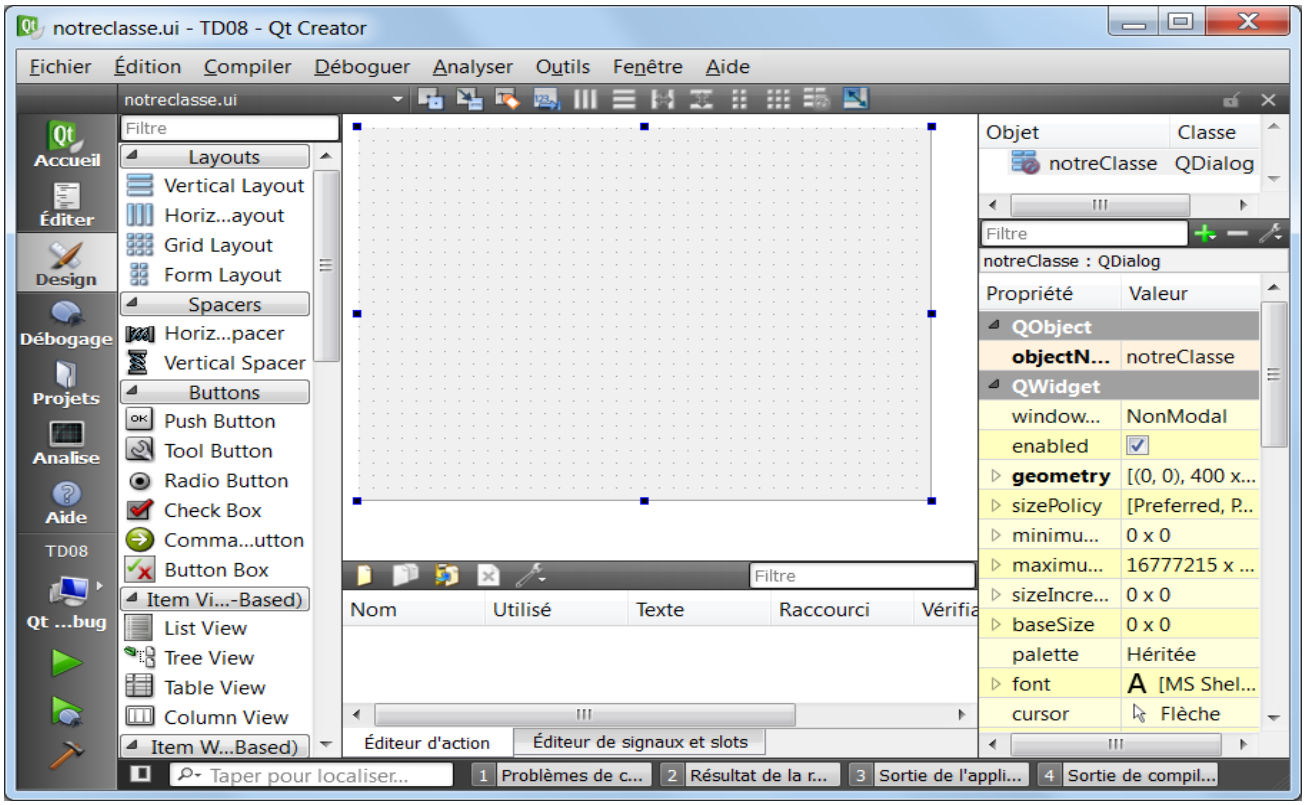
La seconde est plus originale : nous pouvons utiliser l'éditeur graphique QtDesigner pour spécifier l'apparence visuelle que nous souhaitons donner à notre dialogue.

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 :



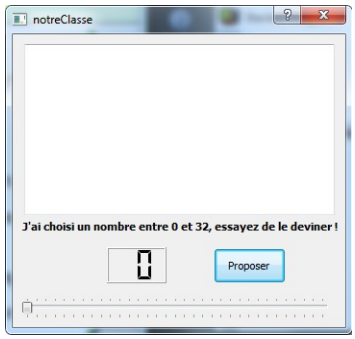
La surface grisée occupant ici la place centrale est le "fond" de la fenêtre sur lequel nous allons disposer les différents éléments qui vont composer l'interface de notre programme.

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.

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 ☐.

Veillez 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**. Veillez à ne **JAMAIS** changer l'objectName du dialogue lui-même (sa valeur doit rester `notreClasse`).

Il faut également que le TextEdit ait pour objectName `verdict` ☐.

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 read only est cochée (pour empêcher l'utilisateur d'insérer du texte dans le widget).

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".

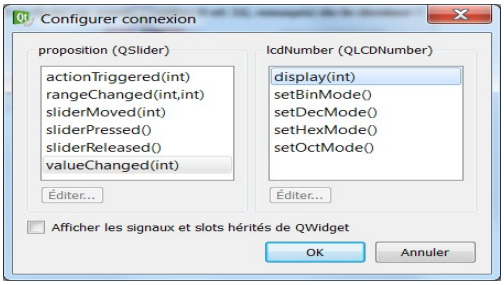
La liaison entre la position du curseur et la valeur affichée par le LCDNumber ne nécessite aucune écriture de code : il s'agit simplement d'indiquer qu'un événement (le déplacement du curseur) doit en déclencher un autre (la modification de la valeur affichée). Les widgets détectent certains événements 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.



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()`.

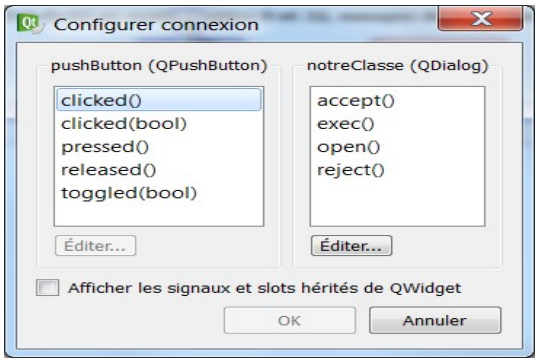
Nous devons maintenant connecter le bouton [Proposer] à une fonction... qui n'existe pas encore puisque nous devrons l'écrire. Nous allons donc simplement indiquer comment elle s'appellera, et QtCreator s'arrangera pour la trouver lorsqu'il en aura besoin.

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 annoncions que notre programme va comporter une fonction nommée `f_proposer()`.

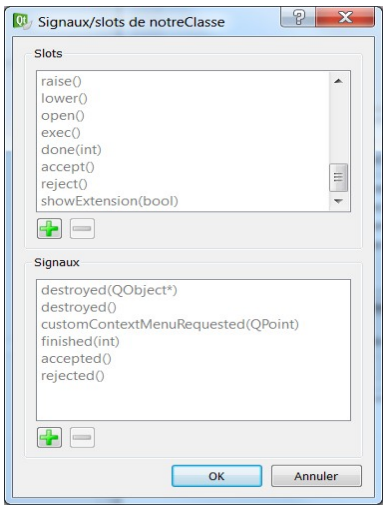
Cliquez sur le bouton [Editer] ☐.



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

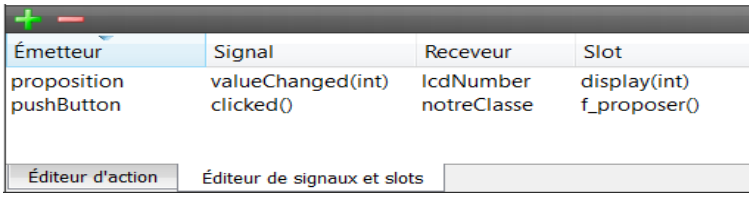
Un nouveau slot est créé. Donnez-lui pour nom "`f_proposer()`" ☐.

La liste de slots proposée par le dialogue de connexion comporte maintenant notre nouvelle fonction. Sélectionner-la (ainsi que l'évènement `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_proposer()` de `notreClasse`.

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



Cette connexion était la dernière opération nécessitant l'éditeur graphique. Revenez à l'éditeur de texte en fermant la fenêtre (confirmez évidemment la sauvegarde si elle vous est proposée) ☐.

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_proposer()`.

Déclaration de la fonction `f_proposer()`

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 :

```
1  class notreClasse : public QDialog
2  {
3      Q_OBJECT
4  public:
5      explicit notreClasse(QWidget *parent = 0);
6      ~notreClasse();
7  public slots:
8      void f_proposer();
9  private:
10     Ui::notreClasse *ui;
11 };
```

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_proposer()`

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 :

```
1  void notreClasse::f_proposer()
2  {
   //on récupère la proposition faite par le joueur
3  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 >>). 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 :

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

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

> L'usage d'un `QTextStream` exige l'ajout d'une directive `#include <QTextStream>` 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 tout `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 `private:` (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 :

```
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 connait 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 :

```
1 notreClasse::notreClasse(QWidget *parent) :
2     QDialog(parent),
3     ui(new Ui::notreClasse)
4 {
5     ui->setupUi(this);
6     prepareNouvellePartie();
7 }
```

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.

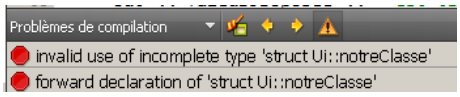
Il est donc préférable d'insérer l'appel à `srand()` dans la fonction `main()` ☐ :

```
1 int main(int argc, char *argv[])
2 {
3 QApplication a(argc, argv);
4 srand(time(0));
5 notreClasse w;
6 w.show();
7 return a.exec();
8 }
```

> 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 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).