Centre Informatique pour les Lettres et les Sciences Humaines

TD 12 : La course d'escargots

Le programme que nous allons réaliser propose une animation réalisée au moyen d'un `QTimer`. Les `QTimer` sont des widgets très utiles, qui présentent la particularité de ne pas être proposés par Qt Designer.

> Qt Designer est un programme de dessin, qui permet de mettre en place les éléments visibles d'une interface utilisateur. Un `QTimer` est invisible et ne fait pas à proprement parler partie de l'interface utilisateur du programme...

L'interface utilisateur du programme réalisé au cours du TD 12

La création du `QTimer` sera donc confiée au code que nous allons placer dans le corps du constructeur de notre classe de dialogue.

1 - Ce que nous cherchons à obtenir

Les trois escargots participant à une course sont représentés par trois `QSlider`. L'utilisateur parie sur l'un des concurrents (en cliquant sur le bouton radio correspondant), puis donne le départ en cliquant sur le bouton. Le programme fait ensuite avancer les curseurs à des vitesses variables et imprévisibles, jusqu'à ce que l'un d'entre eux parvienne à l'extrémité de sa piste. L'utilisateur reçoit alors un message indiquant l'issue de son pari. Il doit ensuite cliquer à nouveau sur le bouton pour replacer les escargots sur la ligne de départ et pouvoir organiser une nouvelle course.

2 - Création du projet et dessin de l'interface

Créez, en suivant la procédure habituelle, un projet nommé TD12 ☐.

Placez sur ce dialogue trois slider (menu Tools, catégorie Input) respectivement baptisés `escargot_0`, `escargot_1` et `escargot_2` ☐.

Augmentez les propriétés "maxValue" de ces slider (donnez-leur la valeur 800, par exemple) ☐ et veillez à que leurs propriétés "enabled" soient `false` ☐.

> Si vous le souhaitez, vous pouvez aussi changer les propriétés paletteBackgroundColor, de façon à obtenir un programme qui ne se contente pas d'être inutile mais soit aussi de mauvais goût.

Placez un pushButton (menu Tools, catégorie Button) nommé `b_bouton` sous les sliders ☐.

Placez un buttonGroup ((menu Tools, catégorie Containers) à droite des sliders ☐.

A l'intérieur de ce buttonGroup, placez trois radioButton (menu Tools, catégorie Button) respectivement nommés `pari_0`, `pari_1` et `pari_3` ☐.

Modifiez les propriétés de ces radioButton de façon à ce qu'ils soient dépourvus de "text" et que leurs "buttonGroupID" soient respectivement 0, 1 et 2 ☐.

Votre dialogue devrait maintenant ressembler à l'image ci-contre.

Effacez le titre et mettez à 0 la propriété "lineWidth" du buttonGroup, ce qui le rendra invisible ☐.

Créez un slot `f_pari(int)` et connectez-le à l'événement `clicked(int)` du buttonGroup ☐.

Créez enfin trois slots nommés `f_avance()` , `f_marques()` et `f_partez()` ☐.

> Ces slots, tout comme b_bouton, ne font pour l'instant l'objet d'aucune connexion. En effet, `f_avance()` est la fonction qui sera exécutée en réponse à des signaux émis par le `QTimer`, et n'il n'est donc pas possible d'établir la connexion avec Qt Designer. Le bouton, pour sa part, ne déclenchera pas toujours l'exécution de la même fonction, et c'est donc le programme qui se chargera d'établir les connections requises aux moments nécessaires.

N'oubliez pas d'enregistrer votre travail avant de revenir à Visual C++ ☐.

3 - Ecriture du code

Ce programme est très simple et ses différentes fonctions peuvent être écrites directement, dans l'ordre où elles seront exécutées lors de la première course.

Le constructeur

La première tâche incombant au constructeur est la création du `QTimer`. Il s'agit d'un widget dont le rôle est d'émettre un signal `timeout()` à intervalles réguliers. En connectant ce signal à une fonction (un slot), nous obtiendrons donc une succession d'appels à cette fonction, un peu comme si un utilisateur invisible cliquait à intervalles réguliers sur un bouton (tout aussi invisible) connecté à la fonction.

La création d'un widget Qt doit se faire par allocation dynamique de mémoire.

> Il arrive parfois qu'un programme comportant un widget qui n'a pas été créé par allocation dynamique donne l'impression de fonctionner normalement. Ce n'est qu'une illusion transitoire.

Comme diverses fonctions devront accéder au timer pour le mettre en route et l'arrêter, nous devons leur fournir un moyen d'accéder à ce widget. Le plus simple est d'ajouter à la classe TD12DialogImpl une variable membre de type "pointeur sur `QTimer`" nommée `m_leTimer` ☐.

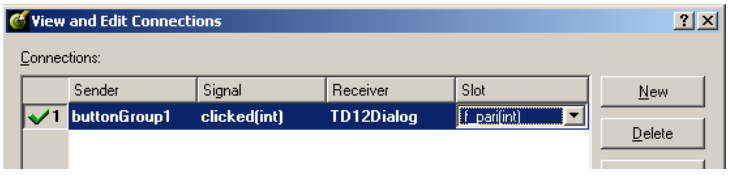
> Si, au lieu d'un "pointeur sur `QTimer`", vous ajoutez à la classe un membre de type `QTimer`, vous créez un widget sans utiliser l'allocation dynamique (cf. ci-dessus...)

L'adresse du `QTimer` sera donc stockée dans ce pointeur *(3)*. Remarquez que l'appel à `new` s'accompagne de la transmission d'une valeur destinée à initialiser un paramètre du constructeur de `QTimer`. Cette valeur (l'adresse de l'instance de TD12DialogImpl pour le compte de laquelle le `QTimer` est créé) permet de transférer au dialogue la responsabilité de la destruction du `QTimer`.

> En d'autres termes, c'est le destructeur du dialogue qui se chargera d'appliquer un `delete` à tous les widgets du dialogue. C'est notamment pour cette raison que les widgets en question doivent avoir été créés par `new`.

```
1  TD12DialogImpl::TD12DialogImpl(QWidget* parent, const char* name, bool modal, WFlags f )
                                                    : TD12Dialog( parent, name, modal, f )
2  {
3  m_leTimer = new QTimer(this);
4  connect(m_leTimer, SIGNAL(timeout()), this, SLOT(f_avance()));
5  srand(time(0));
6  f_marques();
7  }
```

Une fois le timer créé, il faut le connecter au slot dont il devra déclencher l'appel. La fonction `connect()` attend quatre arguments : l'émetteur, le signal, le récepteur et le slot. L'analogie avec le fonctionnement de Qt Designer est évidente :



Le constructeur s'achève par *(5)* l'initialisation du générateur de nombres pseudo-aléatoires (cf. Annexe 2) et l'appel *(6)* de la fonction qui procède aux préparatifs nécessaires avant une course.

> Le générateur de nombre pseudo-aléatoire va, bien entendu, être nécessaire pour obtenir "*des vitesses variables et imprévisibles*".

Donnez au corps de votre constructeur le contenu suggéré ci-dessus ☐.

La fonction `f_marques()`

Dans un premier temps, cette fonction s'assure *(3-5)* que tous les escargots sont bien alignés sur la ligne de départ. Elle veille ensuite à ce que les trois boutons radios soient dans le même état : non "cochés" *(6-8)* et actifs *(9-11)*.

```
1  void TD12DialogImpl::f_marques()
2  {
3  escargot_0->setValue(0);
4  escargot_1->setValue(0);
5  escargot_2->setValue(0);
6  pari_0->setChecked(false);
7  pari_1->setChecked(false);
8  pari_2->setChecked(false);
9  pari_0->setEnabled(true);
10 pari_1->setEnabled(true);
11 pari_2->setEnabled(true);
12 b_bouton->setText("< Choisissez votre favori ! >");
13 disconnect (b_bouton, SIGNAL(clicked()), this, SLOT(f_marques()));
14 }
```

La fonction `f_marques()` place ensuite sur `b_bouton` le texte indiquant au joueur qu'il doit cliquer sur l'un des boutons radio avant de pouvoir lancer la course *(12)* et en s'assurant qu'elle ne sera pas appelée lors du prochain clic sur le bouton *(13)*.

> Si la fonction `disconnect()` n'appelle guère de commentaires, il est possible que vous vous demandiez pourquoi `f_marques()` a besoin d'être déconnectée du bouton. Pour le comprendre, il faut considérer ce qui va se passer lors des courses suivantes. Pour la première course, c'est le constructeur qui appelle `f_marques()` explicitement, mais il ne peut évidemment en aller de même par la suite. Pour que `f_marques()` prépare la deuxième course, il faudra qu'elle soit appelée en cliquant sur le bouton, ce qui implique qu'elle sera alors connecté à celui-ci...

Ajoutez une fonction `f_marques()` à votre dialogue et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `f_pari()`

Lorsqu'un utilisateur choisit l'un des concurrents, il faut mémoriser ce choix sous une forme qui permettra, une fois de course achevée, de vérifier facilement s'il était judicieux. Différentes méthodes sont envisageables et le code proposé ci-dessous *(3-8)* stocke dans une variable membre l'adresse du `QSlider` représentant l'escargot choisi.

```
1  void TD12DialogImpl::f_pari(int quoi)
2  {
3  switch(quoi)
4     {
5     case 0 : m_pari = escargot_0; break;
6     case 1 : m_pari = escargot_1; break;
7     case 2 : m_pari = escargot_2; break;
8     }
9  b_bouton->setText("Prêts ? Partez !");
10 connect (b_bouton, SIGNAL(clicked()), this, SLOT(f_partez()));
11 }
```

Une fois qu'un choix a été exprimé, l'utilisateur est autorisé à lancer la course (mais il peut également modifier son choix). Cette autorisation est signifiée par le changement du texte affiché sur le bouton *(9)* et matérialisée par la connexion du bouton à la fonction `f_partez()` *(10)*.

Ajoutez à votre dialogue une variable membre de type "pointeur sur `QSlider`" nommée `m_pari` ☐.

Ajoutez une fonction `f_pari()` à votre dialogue et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `f_partez()`

La première précaution *(3-5)* que doit prendre la fonction `f_partez()` est, bien entendu, de bloquer les paris...

L'animation que nous allons mettre en place fait avancer tous les escargots à la même fréquence, les différences de vitesses provenant de la longueur des pas effectués.

> L'option inverse (taille des pas identique, mais fréquences différentes) pourrait aussi convenir, mais elle est nettement plus difficile à programmer.

Au départ de la course, tous les escargots adoptent un pas identique *(6-8)* qui, s'il ne variait pas par la suite, les conduirait à atteindre la ligne d'arrivée en 800 / 4 = 200 pas (en supposant que vous avez adopté 800 comme valeur maximale pour les `QSlider`).

```
1  void TD12DialogImpl::f_partez()
2  {
3  pari_0->setEnabled(false);
4  pari_1->setEnabled(false);
5  pari_2->setEnabled(false);
6  escargot_0->setPageStep(4);
7  escargot_1->setPageStep(4);
8  escargot_2->setPageStep(4);
9  b_bouton->setText("< C'est parti... >");
10 disconnect (b_bouton, SIGNAL(clicked()), this, SLOT(f_partez()));
11 m_leTimer->start(50, false);
12 }
```

La fonction `f_partez()` signale *(9)* que la course est en cours en changeant à nouveau le texte affiché par le bouton et se déconnecte du celui-ci *(10)* pour éviter que le joueur puisse faire recommencer une course qui lui semblerait mal engagée pour lui.

Il reste enfin à mettre le timer en route *(11)* pour que les pas soient effectués. La fonction `start()` attend deux arguments : le premier indique la périodicité avec laquelle le signal `timeout()` doit être émis, alors que le second indique si ce signal doit être émis plusieurs fois.

> En clair, après exécution de la ligne 11, le signal `timeout()` sera émis toutes les 50 millisecondes, jusqu'à ce que le timer soit arrêté. Si le second argument avait été `true`, le signal n'aurait été émis qu'une seule fois, 50 millisecondes après la mise en route du timer, qui se serait ensuite arrêté tout seul.

Comme le constructeur de la classe de dialogue a connecté le signal `timeOut()` du timer à la fonction `f_avance()`, c'est cette fonction qui va être appelée toutes les 50 millisecondes (ie. 20 fois par seconde).

> Si l'on suppose que la fonction `f_avance()` fait faire un seul pas à chaque escargot et que la taille des pas reste constante, la course durera donc 200 / 20 = 10 secondes, comme un 100 mètres olympique hommes...

Ajoutez une fonction `f_partez()` à votre dialogue et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `f_avance()`

Pour chaque escargot, la fonction `f_avance()` doit modifier aléatoirement la taille du pas, le faire effectuer et vérifier si la ligne d'arrivée est atteinte. Etant donné qu'il s'agit là d'une répétition d'une tâche non élémentaire, il serait tentant d'utiliser une boucle. Malheureusement, nous n'avons pas de moyen direct pour faire en sorte que la boucle travaille sur `escargot_0` lors du premier passage, sur `escargot_1` lors du deuxième et sur `escargot_2` lors du troisième. Plutôt qu'avec une boucle, nous allons donc éviter d'écrire trois fois la même séquence en appelant explicitement trois fois une fonction sous-traitante.

Cette fonction, que nous nommerons `avance()`, doit donc avoir un paramètre de type "pointeur sur un `QSlider`" (ce qui permettra de lui dire quel escargot elle doit faire avance) et renvoyer un booléen (indiquant si l'escargot en question a franchi la ligne d'arrivée).

La fonction `f_avance()` commencera donc ainsi :

```
1  void TD12DialogImpl::f_avance()
2  {
3  bool courseFinie = avance(escargot_0);
4  courseFinie = courseFinie || avance(escargot_1);
5  courseFinie = courseFinie || avance(escargot_2);
6  if(!courseFinie) //ie. si aucun des appels à avance() n'a renvoyé true
7     return;
```

La seconde partie de la fonction `f_avance()` doit donc prendre en charge les traitements qui sont nécessaires lorsque la course vient de s'achever. Après avoir *(8)* arrêté le timer, il convient d'annoncer le verdict au joueur *(9-12)*. Remarquez que la détection du franchissement de la ligne

d'arrivée opérée par `avance()` n'est pas utilisée ici : on se borne à vérifier si l'escargot sur lequel portait le pari est arrivé en bout de piste.

> Utiliser `avance()` pour déterminer si le pari est gagné complique considérablement les problèmes de communication entre `f_avance()` et `avance()`, surtout si l'on souhaite prendre en compte la possibilité d'ex æquo. La méthode proposée ici est plus simple et juge le pari gagné si l'escargot désigné arrive premier, même s'il est ex æquo avec un autre.

```
8  m_leTimer->stop();
9  if(m_pari->value() == m_pari->maxValue())
10   QMessageBox::information(this, "Escargots", "Vous avez gagné votre pari !" );
11 else
12   QMessageBox::critical(this, "Escargots", "Vous avez perdu votre pari !" );
13 b_bouton->setText("A vos marques !");
14 connect(b_bouton, SIGNAL(clicked()), this, SLOT(f_marques()));
15 }
```

La fonction `f_avance()` se conclue en changeant à nouveau l'intitulé du bouton *(13)* et en le connectant à la fonction qui permet de faire une nouvelle course *(14)*.

> Nous établissons ici la connexion dont la suppression par la fonction `f_marques()` a pu vous surprendre page 4.

Ajoutez une fonction `f_avance()` à votre dialogue et donnez-lui le contenu suggéré ci-dessus ☐.

La fonction `avance()`

Cette fonction exploite directement le fonctionnement des `QSlider` : lorsque la fonction `addStep()` est appelée au titre de l'un d'entre eux *(5)*, celui-ci avance son curseur d'une quantité déterminée par sa propriété pageStep.

La fonction `f_partez()` a initialement fixé cette quantité à 4, mais nous souhaitons introduire un peu de suspens en faisant varier aléatoirement la taille des pas. La fonction `avance()` tire donc au hasard un nombre compris entre –2 et 2, qu'elle ajoute ensuite à la propriété pageStep du `QSlider` concerné *(4)*.

```
1  bool TD12DialogImpl::avance(QSlider * unEscargot)
2  {
3  int variationDePas = (rand() % 5) - 2; // un nombre entre -2 et 2
4  unEscargot->setPageStep(unEscargot->pageStep() + variationDePas);
5  unEscargot->addStep();
6  return unEscargot->value() == unEscargot->maxValue();
7  }
```

La fonction s'achève en renvoyant le résultat de la comparaison entre la position actuelle de l'escargot et la valeur correspondant à la fin de la piste.

> Deux précisions importantes : essayer de donner à pageStep une valeur négative revient à lui donner une valeur nulle (c'est pour ça que les escargots ne reculent jamais) et essayer de donner à un `QSlider` une valeur supérieure à sa `maxValue` revient à lui donner cette valeur (c'est pour ça que le test de la ligne *6* peut être `==` et non `>=`).

Ajoutez une fonction `avance()` à votre dialogue et donnez-lui le contenu suggéré ci-dessus ☐.

Complétez votre fichier td12dialogimpl.cpp avec les directives d'inclusion nécessaires ☐ :

```
1  #include "time.h"
2  #include "qslider.h"
3  #include "qradiobutton.h"
4  #include "qpushbutton.h"
5  #include "qmessagebox.h"
```

Vous pouvez maintenant compiler ☐ et exécuter ☐ le programme. Bonne chance !