Centre Informatique pour les Lettres et les Sciences Humaines

TD 16 : Taquin

1 - Réflexions préalables 2

2 - Création du projet et dessin de l'interface 2

3 - La classe `TD16DialogImpl` 3

Le constructeur 3
La fonction `f_clic()` 4
La fonction `deplace()` 5
La fonction `f_melange()` 6
La fonction `f_inverse()` 6
La fonction `miseEnRoute()` 7
La fonction `f_tic()` 7
La fonction `f_annales()` 7

3 - La classe `CAnnales` 8

Choix techniques 8
Le constructeur 9
La fonction `annonceVerdict()` 9
La fonction `insere()` 10
La fonction `saisieNomJoueur()` 11
Les fonctions `afficheRapides()` et `afficheEconomes()` 11
La fonction `afficheScores()` 12

4 - La classe `CScore` 13

Le fichier cscore.h 13
Le constructeur 13
L'opérateur d'insertion dans un `QTextStream` 13
L'opérateur d'extraction à partir d'un `QTextStream` 14

5 - Exercice 14

Le programme réalisé au cours du TD 15 propose une version informatisée d'un casse-tête bien connu : il s'agit de remettre dans l'ordre 15 pions disposés dans une grille de quatre lignes et quatre colonnes, en utilisant la case vide pour faire glisser les pions. Le programme mesure le temps mis pour parvenir à l'objectif et le nombre de déplacements effectués, et conserve une trace des meilleures performances réalisées.

Pour effectuer un déplacement, il suffit de cliquer sur le pion concerné : il se déplace alors en direction de la case vide, en entraînant éventuellement les pions qui l'en séparent. Dans la position illustrée ci-contre, par exemple, cliquer sur le pion XIII aurait pour effet de faire descendre les pions VI, XI et XIII pour laisse libre la case supérieure droite.

L'interface utilisateur du programme réalisé au cours du TD 16

1 - Réflexions préalables

La première décision à prendre concerne la représentation de l'état du problème : notre programme a-t-il besoin d'une représentation interne différente de la présentation visuelle proposée à l'utilisateur ? Les traitements effectués ne visent qu'a déterminer si le mouvement proposé est possible et à vérifier si l'état souhaité est atteint. Il ne parait donc pas nécessaire de disposer d'une représentation interne spécifiquement destinée à favoriser ces traitements.

Il en irait tout autrement si nous envisagions un programme tel que celui décrit ici, par exemple.

Si les pions sont représentés par des boutons (ce qui parait logique, puisque l'utilisateur va cliquer dessus...), deux approches sont envisageables : soit les boutons se déplacent réellement, soit ils se contentent de changer d'intitulé (ce qui suffit à donner l'impression d'un mouvement). La première approche permettrait de représenter les étapes intermédiaires du mouvement (les moments au cours desquels le pion qui se déplace n'est plus dans sa case d'origine mais n'est pas encore arrivé dans sa case de destination). La seconde promet une simplicité de mise en œuvre qui semble bien plus attractive que l'éventuel avantage esthétique d'une animation sophistiquée, et c'est donc celle-ci que nous allons adopter.

Le programme doit également offrir des commandes permettant de placer les pièces dans une position de départ intéressante et d'obtenir l'affichage des meilleurs scores enregistrés. Ces commandes seront proposées dans un menu et seront également accessibles grâce à des "raccourcis clavier". Deux positions de départ seront proposés : un mélange aléatoire des pièces et une inversion parfaite de leurs positions.

2 - Création du projet et dessin de l'interface

Utilisez la procédure habituelle (cf. aide mémoire) pour créer un projet nommé TD16 ☐ et effacez tous les widgets présents dans le dialogue ☐.

Modifiez la propriété "font" du dialogue pour adopter la police Arial en taille 22 ☐.

Insérez un buttonGroup (Menu Tools, catégorie Containers) auquel vous donnerez les propriétés suggérées ci-contre ☐.

name		groupeBoutons
geometry	x	3
	y	22
	width	248
	height	248
frameShadow		Raised
lineWidth		2
title		

Ajoutez, en bas de votre dialogue, deux textLabel (Menu Tools, catégorie Display) respectivement nommés `m_afficheDuree` et `m_afficheCoups` ☐.

Encadrez ces deux textLabel à l'aide de frames (Menu Tools, catégorie Containers) ☐.

Votre dialogue devrait maintenant ressembler à celui représenté ci-contre.

Slot	Rôle de la fonction correspondante
`f_clic(int)`	Déplacement des pièces.
`f_melange()`	Création d'une position de départ aléatoire.
`f_inverse()`	Création d'une position de départ "inversée".
`f_annales()`	Affichage des meilleurs scores.
`f_tic()`	Chronométrage de la partie.

Créez les slots décrits ci-dessus ☐ et connectez `f_clic(int)` à l'événement `clicked(int)` du buttonGroup ☐ :

	Sender	Signal	Receiver	Slot
✓1	groupeBoutons	clicked(int)	TD16Dialog	f_clic(int)

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

3 - La classe `TD16DialogImpl`

L'interface que nous venons de dessiner est incomplète, puisque le menu et les boutons représentant les pions n'y figurent encore pas.

Qt Designer ne permet pas de créer des menus, mais les boutons auraient pu être mis en place avec cet outil. Il sera toutefois plus facile d'accéder à ces boutons si leurs adresses sont stockées dans un tableau que si elles sont stockées chacune dans une variable différente (ce qui serait le cas si les boutons étaient positionnés avec Qt Designer).

Une des informations nécessaires à la mise en place des pions est la liste des étiquettes que doivent porter les boutons. Pour rendre cette information facilement accessible, on peut utiliser un tableau de constantes défini dans le fichier TD16DialogImpl.cpp :

```
const char * etiquettes[] = { "I", "II", "III", "IV", "V", "VI", "VII", "VIII",
                              "IX", "X", "XI", "XII","XIII", "XIV", "XV", ""};
```

Introduisez cette définition avant le début du constructeur ébauché par Visual C++ ☐.

Le constructeur

La première partie du constructeur procède donc à la mise en place des boutons, qui exige évidemment de tenir compte de la position de ceux-ci dans la grille. Deux fonctions sont donc chargées, à partir du numéro d'un bouton (son index dans le tableau `m_pions`), de calculer les numéros de la ligne et de la colonne où celui-ci figure. Un second calcul transforme ces numéros (de 0 à 2) en position physique à l'intérieur du `groupeBoutons`, cette position étant évidemment fonction de la taille d'un bouton et de l'espace laissé inoccupé entre les boutons et le cadre :

```
1   TD16DialogImpl::TD16DialogImpl(QWidget* parent, const char* name, bool modal, WFlags f )
                                           : TD16Dialog( parent, name, modal, f )
2   {
    //mise en place des boutons
3   const int MARGE = 4;
4   const int PION = 60;
5   assert(sizeof(etiquettes) / sizeof(etiquettes[0]) == 16);
6   int n;
7   for (n=0 ; n < 16 ; ++n)
8      {
9      m_pions[n] = new QPushButton(etiquettes[n], groupeBoutons);
10     m_pions[n]->setGeometry(MARGE + (colonne(n) * PION),   //position horizontale
                               MARGE + (ligne(n) * PION),     //position verticale
                               PION,                          //largeur
                               PION);                         //hauteur
11     }
12  QPushButton * bidon = new QPushButton(this);
13  bidon->setFocus();        //c'est lui qui a le focus…
14  bidon->hide();            //mais on ne le voit pas
```

Le bouton créé par les lignes *12-14* ne sert qu'à éviter que la marque du focus privilégie arbitrairement l'un des pions.

Le constructeur met ensuite en place le système de gestion des commandes. Pour gérer facilement les raccourcis claviers, quatre `QAction` sont créées *(15-22)*, puis "greffées" sur les menus *(23-32)* :

```
15  QAction * melange = new QAction("Mélangée", CTRL+Key_M, this);
16  connect(melange, SIGNAL(activated()), this, SLOT(f_melange()));
17  QAction * inverse = new QAction("Inversée", CTRL+Key_I, this);
18  connect(inverse, SIGNAL(activated()), this, SLOT(f_inverse()));
19  QAction * scores = new QAction("Meilleurs scores...", CTRL+Key_S, this);
20  connect(scores, SIGNAL(activated()), this, SLOT(f_annales()));
21  QAction * aPropos = new QAction("Quiter", Key_Escape, this);
22  connect(aPropos, SIGNAL(activated()), this, SLOT(accept()));
    //mise en place des menus
23  QPopupMenu * sousMenu = new QPopupMenu(this);
24  melange->addTo(sousMenu);
25  inverse->addTo(sousMenu);

26  QPopupMenu * leMenu = new QPopupMenu(this);
27  leMenu->insertItem("&Nouvelle partie", sousMenu);
28  scores->addTo(leMenu);
29  leMenu->insertSeparator();
30  aPropos->addTo(leMenu);

31  QMenuBar *laBarre = new QMenuBar(this);
32  laBarre->insertItem("&Jeu", leMenu );
```

En plus de simplifier la mise en place des menus, les objets de type `QAction` prennent en charge les équivalents clavier des commandes auxquelles ils correspondent.

Il ne reste plus au constructeur qu'à mettre en place le timer et à procéder à quelques réglages :
- Initialisation *(35)* du générateur de nombre aléatoires (pour mélanger les pions...)
- Initialisation *(36)* d'une variable membre booléenne qui permet à la fonction `f_clic()` de vérifier qu'une partie est effectivement en cours
- Initialisation *(37)* d'une variable membre qui indique quelle case est vide. Le bouton qui figure cette case ne comporte aucun intitulé, mais l'illusion de son absence peut être améliorée en rendant vraie sa propriété "flat" *(38)*.

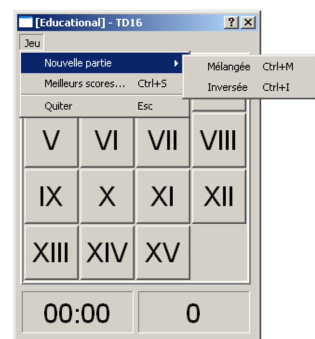
```
    //mise en place du chronomètre
33  m_timer = new QTimer(this);
34  connect(m_timer, SIGNAL(timeout()), this, SLOT(f_tic()) );
    //conditions initiales
35  srand(time(0));
36  m_enCours = false;
37  m_trou = 15;
38  m_pions[m_trou]->setFlat(true);
39  }
```

Donnez au constructeur le contenu suggéré ci-dessus ☐.

Ajoutez à la classe TD16DialogImpl les variables membre nécessaires :
- un tableau de 16 pointeurs sur `QPushButton` nommé `m_pions` ☐.
- un pointeur sur `QTimer` nommé `m_timer` ☐.
- un `int` nommé `m_trou` ☐.
- un booléen nommé `m_enCours` ☐.

Ajoutez aussi à cette classe les deux fonctions membre suivantes ☐ :

```
int ligne(int n) {return n/4;}
int colonne(int n) {return n%4;}
```



Après avoir ajouté les directives d'inclusions ☐ nécessaires en tête du fichier td16dialogimpl.cpp, vous pourrez compiler et exécuter le programme. Bien qu'il soit encore loin d'être complet, il permet déjà de vérifier que tout se présente normalement et que les menus fonctionnent (cf. ci-dessus).

La fonction `f_clic()`

La fonction qui est appelée lorsque le joueur clique sur l'un des pions ne fait rien si la partie n'est pas en cours ou si le mouvement demandé est impossible. L'examen de la légalité du coup et les opérations permettant de l'effectuer réellement sont déportés dans une fonction nommée `deplace()`, qui reçoit comme paramètre le numéro du pion à déplacer.

```
1  void TD16DialogImpl::f_clic(int candidat)
2  {
3  if(!m_enCours || ! deplace(candidat))
4    return;
```

Si le mouvement demandé a été jugé acceptable par la fonction `deplace()`, la fonction `f_clic()` incrémente le compteur de coups et en affiche la valeur *(5-7)*, puis vérifie s'il reste des pions n'ayant pas encore rejoint leur place d'origine *(8-11)* :

```
5  char leTexte[5];
6  sprintf(leTexte, "%i", ++m_nbCoups);
7  m_afficheCoups->setText(leTexte);
   //gagné ?
8  int n;
9  for(n=0 ; n < 16 ; ++n)
10   if(m_pions[n]->text() != etiquettes[n])
11     return; //...pas encore !
```

La fonction `sprintf()` est l'une des fonctions de la librairie standard C qui facilitent (un peu) la manipulation de textes stockés dans des tableaux de `char`. Son rôle est de créer une chaîne qui peut comporter la représentation textuelle de valeurs numériques (`sprintf()` joue donc, pour les tableaux de `char`, un rôle analogue à celui joué par `number()` pour les `QString`). Le premier argument donne l'adresse où la chaîne produite doit être stockée. Remarquez que, ici, l'adresse passée est celle du tableau défini à la ligne *5*. La fonction `f_clic()` suppose donc que la représentation textuelle du nombre de coups occupera moins de 5 caractères. Même si cette supposition semble raisonnable, il serait préférable de ne pas avoir à la faire. C'est là l'une des bonnes raisons justifiant l'utilisation de classes du genre de `QString`...

Le deuxième argument de `sprintf()` est une chaîne à l'intérieur de laquelle peuvent figurer un ou plusieurs "spécificateurs de format". Les paramètres suivants donnent les valeurs qui vont, dans la chaîne produite, prendre la place des spécificateurs de format. Cette logique de fonctionnement a été reprise pour la fonction `qDebug()`, qui est décrite dans l'aide-mémoire "Outils et technique de débugage").

Si tous les pions sont à leur place, la partie est finie *(12-13)* et il faut examiner la performance réalisée *(14)* pour, éventuellement, la mémoriser et afficher un message de félicitation et un récapitulatif du palmarès *(15)*. La gestion de ce palmarès est confiée à une classe `CAnnales` dont la classe `TD16DialogImpl` possède une instance comme variable membre.

```
12 m_enCours = false;
13 m_timer->stop();
14 if(m_annales.annonceVerdict(m_nbCoups, m_duree))  (1)
15   f_annales(); //affiche le nouvel état du annales
16 }
```

1 : Il existe une classe `CAnnales` dotée d'une fonction membre `annonceVerdict()` qui reçoit comme paramètres le nombre de coups et le temps mis pour résoudre le casse tête. Cette fonction compare la performance ainsi décrite aux meilleures performances déjà réalisées et affiche éventuellement un message de félicitation. Elle renvoie un booléen qui indique si la performance qu'elle vient d'examiner est ou non mémorable.

Ajoutez à votre classe de dialogue une variable de type `CAnnales` nommée `m_annales` ☐, deux variables de type `int` respectivement nommées `m_nbCoups` et `m_duree` ☐ et une fonction `f_clic()` définie comme suggéré ci-dessus ☐.

La fonction deplace()

Pour que le mouvement demandé soit acceptable, il faut que le pion à déplacer soit sur la même ligne ou sur la même colonne que la case vide. Le fragment de code ci-dessous rejette donc les tentatives de déplacement qui concernent des pions qui ne sont situés ni sur la bonne ligne ni sur la bonne colonne (ainsi que les tentatives de déplacement de la case vide elle-même) :

```
1  bool TD16DialogImpl::deplace(const int p)
2  {
3  int delta = 0;
4  if(ligne(p) == ligne(m_trou))                //on va se déplacer horizontalement
5    delta = (colonne(p) < colonne(m_trou)) ? -1 : 1;   //gauche ou droite ?
6  if(colonne(p) == colonne(m_trou))            //on va se déplacer verticalement
7    delta = (ligne(p) < ligne(m_trou)) ? -4 : 4;       //bas ou haut ?
8  if (p == m_trou || delta == 0)
9    return false; //le déplacement demandé est impossible
```

Ce fragment de code donne au passage à la variable `delta` une valeur qui correspond à la différence entre le numéro de la case vide et celui du pion adjacent devant être déplacé : -1 si le(s) pion(s) se déplace(nt) de la gauche vers la droite, 1 si le mouvement est de droite à gauche, -4 si le déplacement est du haut vers le bas, et 4 s'il est du bas vers le haut.

Si le coup est acceptable, la valeur contenue dans `delta` permet de "déplacer" tous les pions nécessaires, en changeant simplement l'intitulé des boutons correspondants :

```
   //ok, on bouge...
10 for (m_pions[m_trou]->setFlat(false) ; m_trou != p ; m_trou += delta)
11   m_pions[m_trou]->setText(m_pions[m_trou + delta]->text());
```

Une fois ces déplacements effectués, il ne reste qu'à ajuster l'apparence de la nouvelle case vide *(12-13)* et à signaler que le coup demandé a été accepté *(14)* :

```
12 m_pions[m_trou]->setText("");
13 m_pions[m_trou]->setFlat(true);
14 return true;
15 }
```

Ajoutez à votre classe de dialogue une fonction `deplace()` définie comme suggéré ci-dessus ☐.

La fonction `f_melange()`

Pour que la position obtenue permette effectivement de revenir à la position initiale, la fonction `f_melange()` procède simplement à un certain nombre de déplacements respectant les règles.

> Une position parfaitement aléatoire ne garantirait pas que la disposition initiale puisse être retrouvée. La fameuse version de ce casse-tête publiée par Sam Loyd dans les années 1870 proposait d'ailleurs un point de départ rendant le problème insoluble.

```
1  void TD16DialogImpl::f_melange()
2  {
3  int n;
4  for(n=0 ; n < 1000 ; ++n)
5    deplace(rand() % 16);
6  miseEnRoute();
7  }
```

> Remarquez qu'un grand nombre (près de 80%) des mouvements proposés par `f_melange()` sont rejetés par `deplace()`. Il serait possible de ne proposer que des mouvements légaux, ou même de calculer une position "mélangée" acceptable, sans effectuer des déplacements aléatoires. Pour intéressantes qu'elles soient, ces approches nous éloigneraient par trop du sujet de ce TD.

Une fois la disposition des pions choisie, la fonction `f_melange()` appelle *(6)* une fonction chargée des opérations correspondant à un début de partie.

Ajoutez à votre classe de dialogue une fonction `f_melange()` définie comme suggéré ci-dessus ☐.

La fonction `f_inverse()`

La disposition des étiquettes réalisée par cette fonction n'a évidemment rien d'aléatoire : la case 0 doit recevoir l'étiquette 15 (une chaîne vide), la case 1 l'étiquette 14 (la chaîne "XV"), et ainsi de suite. La fonction s'achève évidemment par la `miseEnRoute` de la partie.

```
1  void TD16DialogImpl::f_inverse()
2  {
3  m_pions[m_trou]->setFlat(false);
4  int n;
```

```
5  for(n=0 ; n < 16 ; ++n)
6     m_pions[n]->setText(etiquettes[15-n]);
7  m_trou = 0;
8  m_pions[m_trou]->setFlat(true);
9  miseEnRoute();
10 }
```

Ajoutez à votre classe de dialogue une fonction `f_inverse()` définie comme suggéré ci-dessus ☐.

La fonction `miseEnRoute()`

Le démarrage d'une partie doit s'accompagner de la remise à zéro des indicateurs de performance *(3-6)*, du positionnement de la variable qui indique à `f_clic()` que la partie est commencée *(7)* et de la mise en route du dit chronomètre *(8)* :

```
1  void TD16DialogImpl::miseEnRoute()
2  {
3  m_nbCoups = 0;
4  m_afficheCoups->setText("0");
5  m_duree = 0;
6  m_afficheDuree->setText("00:00");
7  m_enCours = true;
8  m_timer->start(1000);
9  }
```

Ajoutez à votre dialogue une fonction `miseEnRoute()` définie comme suggéré ci-dessus ☐.

La fonction `f_tic()`

La mise en route en route du timer va se traduire par un appel automatique de `f_tic()` toutes les secondes. Cette fonction se contente donc d'incrémenter le compteur de secondes *(3)* et d'en afficher la valeur, exprimée en minutes et secondes *(4-6)*, dans le widget prévu à cet effet *(7)* :

```
1  void TD16DialogImpl::f_tic()
2  {
3  ++m_duree;
4  char texte[6];
5  sprintf(texte, "%02i:%02i", m_duree/60, m_duree%60);
6  m_afficheDuree->setText(texte);
7  }
```

Ajoutez à votre classe de dialogue une fonction `f_tic()` définie comme suggéré ci-dessus ☐.

La fonction `f_annales()`

Cette fonction est appelée d'une part lorsque l'utilisateur sélectionne la commande correspondante dans le menu et, d'autre part, lorsqu'un record vient d'être battu. Son rôle est simplement d'afficher deux tableaux contenant la description des meilleures performances enregistrées. Elle crée un dialogue *(3)* comportant deux `QTable` dont le contenu est fourni par la variable `m_annales` (*13* et *15*), et en provoque l'affichage *(21)* :

```
1  void TD16DialogImpl::f_annales()
2  {
3  QDialog * dlg = new QDialog(this);
4  dlg->setCaption("Qt - Les meilleurs taquins");
5  QFont f(font());
6  f.setPointSize(12);
7  dlg->setFont(f);

8  QLabel * titreUn = new QLabel(dlg);
9  titreUn->setText("Les plus rapides :");
10 QLabel * titreDeux = new QLabel(dlg);
11 titreDeux->setText("Les plus économes :");

12 QTable * tableauRapides = new QTable(dlg);
13 m_annales.afficheRapides(tableauRapides);      (2)
14 QTable * tableauEconomes = new QTable(dlg);
15 m_annales.afficheEconomes(tableauEconomes);    (3)
```

```
16 QVBoxLayout * general = new QVBoxLayout(dlg);
17 general->addWidget(titreUn);
18 general->addWidget(tableauRapides);
19 general->addWidget(titreDeux);
20 general->addWidget(tableauEconomes);
21 dlg->exec();
22 }
```

Les dialogues simples (par exemple ceux qui, comme c'est le cas ici, n'ont pas besoin d'autres fonctions membre que celles héritées de QDialog) peuvent tout à fait être créés "à la volée".

2 : La classe CAnnales dispose d'une fonction afficheRapides() qui place dans la QTable qui lui est désignée la description des victoires ayant pris le moins de temps.

3 : La classe CAnnales dispose d'une fonction afficheEconomes() qui place dans la QTable qui lui est désignée la description des parties gagnées en peu de coups.

Ajoutez à votre classe de dialogue une fonction f_annales() définie comme suggéré ci-dessus ☐.

3 - La classe CAnnales

Les exigences qui ont été formulées à propos de la classe CAnnales sont peu nombreuses, mais leur satisfaction impose la mise en place d'un dispositif relativement complexe. La conservation d'une trace des meilleures performances réalisées nécessite en effet la gestion d'un fichier de données et des comparaisons permettant de situer toute nouvelle performances par rapport à celles déjà connues.

Choix techniques

La conservation des meilleurs scores dans un fichier est une contrainte technique qui n'intéresse pas le joueur. Plutôt que de proposer des commandes "Ouvrir" et "Enregistrer" qui permettraient à celui-ci de gérer explicitement la conservation du palmarès, il semble plus judicieux de procéder à des lectures et écritures automatiques. Pour que les anciens records soient lus lors du lancement du programme, on peut confier cette tâche au constructeur de CAnnales : l'instanciation de cette classe provoquée par la présence de la variable membre m_annales dans la classe TD16DialogImpl suffira alors à déclencher la lecture des données.

Pour gérer efficacement le palmarès, la variable m_annales doit conserver plusieurs informations :
- Le chemin d'accès au fichier contenant les records établis (si un record est battu, il faudra créer une nouvelle version de ce fichier...)
- Le nom du joueur actuellement aux commandes (pour éviter que le joueur ait à ressaisir son nom chaque fois qu'il réalise une performance mémorable...)
- Une structure de données contenant la description du palmarès actuel (pour pouvoir déterminer si une performance qui vient d'être réalisée est ou non mémorable). Le thème de ce TD étant les tableaux, il semble légitime d'utiliser un tableau de six valeurs de type CScore décrivant chacune une performance (les trois plus rapides et les trois plus économes).

4 : Il existe un classe CScore dont les instances décrivent une performance (auteur, date, nombre de coups, duréee...)

Si le thème de ce TD n'était pas les tableaux, la classe CAnnales disposerait sans doute de deux instances d'une classe CPodium, chargées de garder trace l'une des parties les plus rapides, l'autre des parties les plus économes.

Ajoutez à votre projet une classe nommée CAnnales ☐.

Ajoutez à cette classe deux variables membre de type "tableau de 500 char" respectivement nommées m_chemin et m_nomJoueur, ainsi qu'une variable de type "tableau de 6 CScore" nommée m_scores ☐.

Le constructeur

Puisque nous ne souhaitons pas ennuyer l'utilisateur avec des questions relatives au fichier de données, il convient d'adopter une règle de localisation de celui-ci. Une solution classique est de partir du principe que l'utilisateur a installé (ie. copié) le programme là où il avait envie qu'il soit et que cet emplacement est un bon endroit pour stocker les données.

> Cette approche est facile à mettre en œuvre et totalement transparente pour l'utilisateur, mais elle n'est pas au dessus de toute critique : elle échoue si le programme est installé dans un dossier où l'écriture est impossible (un CD, par exemple) et elle ne permet pas la séparation programmes/données nécessaire à la mise en place d'une politique de sauvegarde efficace. Dans le cas de données qui mériteraient réellement d'être sauvegardées, il faudrait prévoir une procédure d'installation demandant à l'utilisateur d'indiquer où les données doivent être placées.

Le chemin d'accès au fichier contenant le programme en cours d'exécution peut être facilement obtenu (cf. cet aide-mémoire) et utilisé pour créer ce qui sera le chemin d'accès au fichiers de données *(3-5)*. Une fois ce chemin obtenu, il ne reste plus qu'à lire les données *(6-13)* :

```
1  CAnnales::CAnnales()
2  {
3  strcpy(m_nomJoueur, "< Anonyme >");
4  QFileInfo info(*qApp->argv());
5  strcpy(m_chemin, info.dirPath() + "/scores.txt");
6  QFile fichier(m_chemin);
7  if(fichier.open(IO_ReadOnly | IO_Translate))
8     {
9     QTextStream records(&fichier);
10    int n;
11    for(n=0 ; n < 6 ; ++n)
12       records >> m_scores[n];
13    }
14 }
```

5 : La classe `CScore` permet d'extraire une valeur d'un `QTextStream` à l'aide de l'opérateur >>.

Donnez au constructeur de votre classe `CAnnales` le contenu suggéré ci-dessus ☐.

La fonction `annonceVerdict()`

La fonction `annonceVerdict()` doit déterminer si la performance qui vient d'être réalisée est ou non mémorable. Elle compare donc les valeurs qui lui ont été passées aux records stockés dans le tableau `m_scores`, de façon à déterminer à quelle place la nouvelle performance peut prétendre sur chacun des podiums. Si les performances déjà connues sont toutes supérieures à celle qui vient d'être réalisée, la fonction s'achève après un simple message de fin de partie :

```
1  bool CAnnales::annonceVerdict(const int nbCoups, const int duree)
2  {
3  int n;
4  int rangVitesse = 0;
5  int rangCoups = 0;
6  for(n = 0 ; n < 3 ; ++n)
7     {
8     if(m_scores[n].duree() > 0 && m_scores[n].duree() < duree)
9        ++rangVitesse;
10    if(m_scores[n+3].nbCoups() > 0 && m_scores[n+3].nbCoups() < nbCoups)
11       ++rangCoups;
12    }
13 if(rangVitesse > 2 && rangCoups > 2)
14    {
15    QMessageBox::warning(0, "Bravo :", "Vous avez gagné !");
16    return false;
17    }
```

6 : La classe `CScore` dispose d'une fonction membre nommée `duree()` qui renvoie la durée de la partie décrite par l'instance au titre de laquelle elle est appelée

7 : La classe `CScore` dispose d'une fonction membre nommée `nbCoups()` qui renvoie le nombre de coups de la partie décrite par l'instance au titre de laquelle elle est appelée

Si la nouvelle performance est mémorable, il faut en connaître l'auteur. La saisie de cette information est confiée à une fonction ad-hoc *(18)*. Une instance de `CScore` décrivant la nouvelle partie est ensuite créée *(19)* et passée à une fonction `insere()` chargée mettre à jour les podiums *(20-21)*. Remarquez que la fonction `insere()` "voit" les podiums comme des tableaux de trois `CScore` qu'elle traite l'un après l'autre, sans jamais avoir connaissance du fait qu'il s'agit des deux moitiés d'un unique tableau de six valeurs...

Cette façon de jouer sur les rapports entre tableaux et pointeurs pour envisager les mêmes données sous des points de vue différents est une pratique très contestable mais tout à fait caractéristique des programmes qui utilisent ce type de structures de données.

```
   //cette partie entre dans les annales !
18 saisieNomJoueur();
19 CScore nouveau(nbCoups, duree, m_nomJoueur);   // 8
20 insere(nouveau, rangVitesse, m_scores);
21 insere(nouveau, rangCoups, m_scores+3);
```

8 : La classe `CScore` dispose d'un constructeur acceptant comme paramètres le nombre de coups, la durée et le nom de l'auteur de la partie décrite par l'instance en cours de création.

Une fois les podiums mis à jour, il faut encore veiller à ce que le fichier de données reflète leur nouvel état :

```
   //sauvegarde du nouvel état des annales
22 QFile fichier(m_chemin);
23 QTextStream records(&fichier);
24 if(fichier.open(IO_WriteOnly | IO_Translate))
25    for(n=0 ; n < 6 ; ++n)
26       records << m_scores[n];   // 9
27 return true;
28 }
```

9 : La classe `CScore` permet d'insérer une valeur dans un `QTextStream` à l'aide de l'opérateur <<.

Ajoutez à `CAnnales` une fonction `annonceVerdict()` définie comme suggéré ci-dessus ☐.

La fonction insere()

L'insertion d'une valeur dans un tableau exige le décalage préalable des valeurs avant laquelle la nouvelle doit être placée :

```
1  void CAnnales::insere(CScore nouveau, int rang, CScore *tableau)
2  {
3  switch(rang)
4     {
5     case 0 :
6        tableau[2] = tableau[1];
7        tableau[1] = tableau[0];
8        tableau[0] = nouveau;
9        break;
10    case 1 :
11       tableau[2] = tableau[1] ;
12       tableau[1] = nouveau;
13       break;
14    case 2 :
15       tableau[2] = nouveau;
16       break;
17    }
18 }
```

Ajoutez à `CAnnales` une fonction `insere()` définie comme suggéré ci-dessus ☐.

La fonction `saisieNomJoueur()`

Tout comme la fonction `f_annales()`, la fonction `saisieNomJoueur()` utilise un dialogue qui ne comporte aucune fonction membre spécifique. Plutôt que d'instancier une classe dérivée de `QDialog`, elle instancie donc directement `QDialog` et ajoute quelques widgets ad hoc :

```
1  void CAnnales::saisieNomJoueur()
2  {
3  QDialog * dlg = new QDialog();
4  dlg->setCaption("Qt - Taquin");
5  QFont f(dlg->font());
6  f.setPointSize(12);
7  dlg->setFont(f);
8  QLabel * message = new QLabel(dlg);
9  message->setAlignment(Qt::AlignHCenter);
10 message->setText("Vous venez d'établir un nouveau record !\n Veuillez saisir "
11      " ci-dessous le nom qui restera\nassocié à cette performance :");
12 QLineEdit * saisie = new QLineEdit(dlg);
13 saisie->setText(m_nomJoueur);
14 QPushButton * bouton = new QPushButton(dlg);
15 bouton->setText("OK");
16 QObject::connect(bouton, SIGNAL(clicked()), dlg, SLOT(accept()));
17 QHBoxLayout * hor = new QHBoxLayout();
18 hor->addStretch();
19 hor->addWidget(bouton);
20 hor->addStretch();
21 QVBoxLayout * general = new QVBoxLayout(dlg);
22 general->setMargin(10);
23 general->setSpacing(10);
24 general->addWidget(message);
25 general->addWidget(saisie);
26 general->addLayout(hor);
```

Une fois ce `QDialog` "customisé" de façon adéquate, on en provoque l'affichage *(28)*. La suite de la fonction `saisieNomJoueur()` n'est exécutée qu'après que l'utilisateur a refermé le dialogue de saisie, et il ne reste donc plus qu'a exploiter le renseignement fourni :

```
27 int fermeture = dlg->exec();
28 if(fermeture == QDialog::Accepted && ! saisie->text().isEmpty())
29   strcpy(m_nomJoueur, saisie->text());
30 }
```

Ajoutez à `CAnnales` une fonction `saisieNomJoueur()` définie comme suggéré ci-dessus ☐.

Les fonctions `afficheRapides()` et `afficheEconomes()`

Ces deux fonctions effectuent quasiment la même tâche, mais sur des données différentes. Elles sous-traitent donc le travail à une même fonction, a qui un paramètre permet d'indiquer quelles données doivent être utilisées :

```
1 void CAnnales::afficheRapides(QTable *tab)
2 {
3 afficheScores(tab, m_scores);
4 }
```

```
1 void CAnnales::afficheEconomes(QTable *tab)
2 {
3 afficheScores(tab, m_scores+3);
4 tab->swapColumns(1, 2, true);
5 }
```

La fonction `afficheScores()` place les durées dans la deuxième colonne et les nombres de coups dans la troisième. Dans le cas du podium basé sur le nombre de coups, il est plus logique de privilégier cette information en intervertissant *(4)* ces deux colonnes.

Ajoutez à `CAnnales` des fonctions `afficheRapides()` et `afficheEconomes()` définies comme suggéré ci-dessus ☐.

La fonction `afficheScores()`

L'utilisation d'une `QTable` pour afficher des données (cf. TD 6) ne pose guère qu'un seul problème : comme les cellules contiennent du texte, il faut créer la représentation textuelle de toute valeur numérique qui doit apparaître dans la table.

```
void CAnnales::afficheScores(QTable *tab, CScore * scores)
{
char * titres[] = {"Nom", "Durée", "Mouvements", "Date", "Heure"};
tab->setNumCols(sizeof(titres)/sizeof(titres[0]));
QHeader * header = tab->horizontalHeader(); //on veut parler au QHeader
int n;
for(n=0 ; n < tab->numCols() ; ++n)
  header->setLabel(n, titres[n]); //on lui indique les titres des colonnes
tab->setNumRows(3);
for(n=0 ; n < 3 ; ++n)
  if(scores[n].duree() > 0)
    {
    char t[100];
    scores[n].copieAuteur(t);
    tab->setText(n, 0, t);
    sprintf(t, "  %i'%02i\"", scores[n].duree() / 60, scores[n].duree() % 60);
    tab->setText(n, 1, t);
    sprintf(t, "%i  ",scores[n].nbCoups());
    tab->setText(n, 2, t);
    scores[n].copieDate(t);
    tab->setText(n, 3,t);
    scores[n].copieHeure(t);
    tab->setText(n, 4, t);
    }
for(n=0 ; n < tab->numCols() ; ++n)
  tab->adjustColumn(n);
}
```

Le deuxième paramètre de `sprintf()` est une chaîne de caractères où se mêlent du texte qui sera affiché "tel quel" et des spécificateurs de format. Remarquez que, pour que des guillemets soient affichés "tels quels" et non interprétés comme le signal de la fin de la chaîne, il faut les faire précéder d'un \. Les guillemets servent ici de symbole des secondes :

Qt - Les meilleurs taquins

Les plus rapides :

	Nom	Durée	Mouvements	Date	Heure
1	Rapide 1	0'10"	100	20/10/05	14:10:33
2	Rapide 2	0'20"	100	20/10/05	13:41:22
3	Rapide 3	0'30"	100	20/10/05	13:40:46

Les plus économes :

	Nom	Mouvements	Durée	Date	Heure
1	Econome 1	10	1'40"	20/10/05	14:10:33
2	Econome 2	20	1'40"	20/10/05	14:07:35
3	Econome 3	30	1'40"	20/10/05	13:53:25

10 : La classe `CScore` dispose de fonctions membre nommées `copieAuteur()`, `copieDate()` et `copieHeure()` qui placent une chaîne de caractère fournissant l'information requise dans la zone de mémoire dont l'adresse leur est passée comme paramètre.

Ajoutez à `CAnnales` une fonction `afficheScores()` définie comme suggéré ci-dessus ☐.

4 - La classe CScore

Bien que les exigences concernant la classe CScore soient relativement nombreuses, elles sont très faciles à satisfaire. Les cinq données devant être mémorisée (nom du joueur, durée et nombre de coups de la partie, date et heure de son achèvement) correspondent à cinq variables membre, et la plupart des fonctions sont triviales au point d'être définies en ligne dans la définition de la classe elle-même. Le code correspondant aux fonctions operator <<() et operator >>() sera donc la seule compagnie qu'aura le constructeur de CScore dans le fichier CScore.cpp.

Bien que les fonctions operator <<() et operator >>() ne puissent pas être membre de la classe CScore, elles sont si intimement liées à celle-ci qu'il semble logique de les déclarer dans le fichier CScore.h et de les définir dans le fichier CScore.cpp.

Le fichier cscore.h

```
1  class CScore
2  {
3  public:
4     CScore(int d =- 1, int n = -1, char * s = "< Anonyme >");
5     int duree()   const {return m_duree;}
6     int nbCoups() const {return m_nbCoups;}
7     void copieAuteur(char * quelquePart) const {strcpy(quelquePart, m_auteur);}
8     void copieDate  (char * quelquePart) const {strcpy(quelquePart, m_date);}
9     void copieHeure (char * quelquePart) const {strcpy(quelquePart, m_heure);}
10 protected:
11    int m_duree;
12    int m_nbCoups;
13    char m_auteur[100];
14    char m_date[10];
15    char m_heure[10];
   //déclarations d'amitié
16 friend QTextStream & operator << (QTextStream &out, CScore leScore);
17 friend QTextStream & operator >> (QTextStream &out, CScore &leScore);
18 };
   //Déclaration des fonctions amies
19 QTextStream & operator << (QTextStream &out, CScore leScore);
20 QTextStream & operator >> (QTextStream &from, CScore &leScore);
```

Ajoutez à votre projet une classe CScore ☐ et donnez au fichier cscore.h le contenu suggéré ci-dessus ☐.

Le constructeur

La seule originalité du constructeur de CScore est qu'il initialise les variables membre m_date et m_heure avec la date et l'heure de création de l'instance sur laquelle il opère :

```
1  CScore::CScore(int n, int d, char * s): m_duree(d), m_nbCoups(n)
2  {
3  strcpy(m_auteur, s);
4  QDateTime maintenant = QDateTime::currentDateTime();
5  strcpy(m_date,  maintenant.date().toString("dd/MM/yy"));
6  strcpy(m_heure, maintenant.time().toString("hh:mm:ss"));
7  }
```

Donnez au constructeur de CScore la définition suggérée ci-dessus ☐.

L'opérateur d'insertion dans un QTextStream

Cette fonction se borne à envoyer dans le QTextStream concerné les valeurs des cinq variables membres, séparées par des passages à la ligne :

```
1  QTextStream & operator << (QTextStream &destination, CScore leScore)
2  {
3  destination << leScore.m_auteur << "\n";
4  destination << leScore.m_duree << "\n";
```

```
5 destination << leScore.m_nbCoups << "\n";
6 destination << leScore.m_date << "\n";
7 destination << leScore.m_heure << "\n";
8 return destination;
9 }
```

Ajoutez à votre fichier cscore.cpp la définition de la fonction suggérée ci-dessus ☐.

L'opérateur d'extraction à partir d'un `QTextStream`

La tâche de l'opérateur d'extraction est un peu plus délicate que celle de son compère. Il faut en effet utiliser `strcpy()` pour transférer le texte lu dans les variables membre qui sont de type "tableau de `char`" et veiller à convertir en valeurs numériques les chaînes de caractères qui décrivent les valeurs des variables de type `int` :

```
1 QTextStream & operator >> (QTextStream &source, CScore &leScore)
2 {
3 strcpy(leScore.m_auteur, source.readLine());
4 leScore.m_duree = source.readLine().toInt();
5 leScore.m_nbCoups = source.readLine().toInt();
6 strcpy(leScore.m_date, source.readLine());
7 strcpy(leScore.m_heure, source.readLine());
8 return source;
9 }
```

Ajoutez à votre fichier cscore.cpp la définition de la fonction suggérée ci-dessus ☐.

Ajoutez les directives d'inclusion nécessaires ☐, puis compilez votre programme et vérifiez qu'il fonctionne normalement ☐.

> Méfiez-vous tout particulièrement de la directive `#include "qbuttongroup.h"`. Son omission dans le fichier TD16DialogImpl.cpp se traduit par le message
>
> ```
> QPushButton::QPushButton(class QWidget *,const char *)' : cannot convert parameter 1
> from 'const char *' to 'class QWidget *'
> ```
>
> concernant la ligne
>
> ```
> m_pions[n] = new QPushButton(etiquettes[n], groupeBoutons);
> ```
>
> Le lien entre ce message et l'erreur qui la cause peut ne pas vous sembler évident...

5 - Exercice

Faites une copie de sauvegarde de votre projet ☐ et modifiez-le de façon à ce qu'il n'utilise plus aucun tableau ☐.

> Les tableaux de `char` seront donc remplacés par des `QString` et les autres types de tableaux par des `QValueList` ou des `QMap`.