Centre Informatique pour les Lettres et les Sciences Humaines

TD 8 : La patience des pharaons

1 - Analyse préalable 2

2 - Création du projet et dessin de l'interface 3

3 - La classe de dialogue 4

Le constructeur 4

Création du jeu de carte 4

Positionnement des boutons et distribution des cartes 5

La fonction `tirerUneCarteDans()` 6

La fonction `calculeGeometrie()` 6

La fonction `descendre()` 7

La fonction `piocher()` 8

4 - La classe `CCarte` 9

Définition de la classe 9

La fonction `devient()` 9

La fonction `associeBouton()` 9

La fonction `valeur()` 10

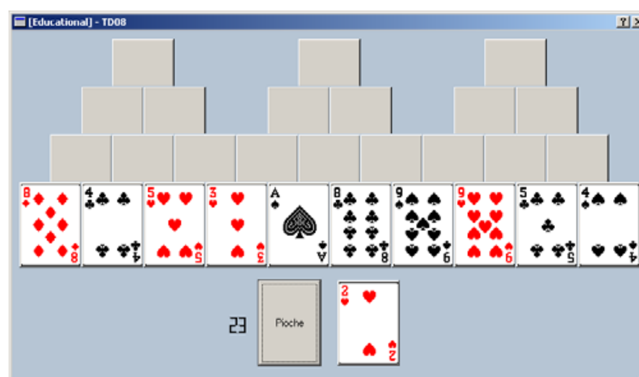
La fonction `recoit()` 10

La fonction `montreToi()` 10

5 - Exercice 10

Le programme réalisé au cours de ce TD est une version informatisée du jeu de cartes connu sous le nom de "réussite des pharaons".

Le but de ce jeu est de faire disparaître toutes les cartes initialement arrangées sur le plateau dans une disposition évoquant trois pyramides. Une carte peut descendre sur le tas placé sous les pyramides si et seulement si sa valeur est immédiatement supérieure ou inférieure à celle de la carte placée au sommet de ce tas. (Les as sont considérés à la fois comme supérieurs aux rois et comme inférieurs aux deux.)



L'interface utilisateur du programme réalisé au cours du TD 08

Dans la configuration représentée ci-dessus, le deux (de cœur) figurant au sommet du tas autorise la descente soit de l'as (de pique), soit du trois (de cœur). Ce second mouvement est préférable, puisqu'il permet la descente ultérieure du quatre (de trèfle), puis du cinq (de cœur), puis du quatre (de pique) et enfin du cinq (de trèfle). Remarquez que, dans ce jeu, seule la valeur des cartes est prise en compte, leur couleur (pique, cœur, carreau ou trèfle) est sans importance.

Au début du jeu, seules les cartes de la rangée inférieure des pyramides sont accessibles. Une carte initialement "bloquée" n'est libérée que lorsque les cartes qui en occultent la partie basse sont toutes deux descendues sur le tas. Si aucun mouvement de descente n'est possible, le joueur doit placer sur le tas la carte occupant le sommet de la pioche. Si cette pioche est épuisée et qu'il reste des cartes dans les pyramides, la partie est perdue.

1 - Analyse préalable

Les règles du jeu exigent que notre programme soit en mesure de manipuler deux ensembles de cartes : celles qui figurent dans les pyramides et celles qui sont disponibles dans la pioche.

L'accès aux cartes placées dans les pyramides doit pouvoir se faire au grée des choix faits par le joueur parmi les possibilités que lui laisse la distribution des cartes. Il semble donc naturel d'organiser ces cartes dans une `QMap`, puisque ce type de conteneur permet d'accéder facilement à n'importe lequel des éléments qu'il contient.

L'accès aux cartes placées dans la pioche va, en revanche, se faire dans un ordre parfaitement prévisible : seule la première d'entre-elles est accessible, et un conteneur de type `QValueList` convient donc parfaitement.

Une autre carte joue un rôle important : celle qui figure au sommet du tas, dont la valeur conditionne les mouvements possibles. Les autres cartes du tas ne sont plus en jeu, et il n'est pas nécessaire d'en conserver une représentation. Le tas ne sera donc pas géré à l'aide d'une collection, mais d'une simple instance de la classe que nous allons créer spécialement pour représenter les cartes, la classe `CCarte`.

Les pyramides, la pioche et le sommet du tas constituent les données essentielles de la représentation de l'état de la partie et l'option la plus raisonnable est certainement d'en faire trois variables membre de la classe représentant notre dialogue.

Pour éviter de toujours proposer la même disposition initiale des cartes (ce qui priverait vite le jeu de tout intérêt), notre programme fera appel aux techniques d'utilisation de nombres pseudo-aléatoires décrites dans l'Annexe 2.

En ce qui concerne la représentation des cartes, nous allons nous appuyer sur des widgets familiers : des `QPushButton`, qui se prêtent facilement aussi bien à l'affichage d'images qu'au déclenchement de l'exécution d'une fonction lorsque l'utilisateur clique dessus. En raison du nombre d'objets concernés, nous allons utiliser un widget de type `buttonGroup` pour regrouper les 28 boutons représentant les pyramides.

Grâce à ce `buttonGroup`, nous allons pouvoir d'une part ajuster précisément la position des cartes à l'aide de quelques ligne de code figurant dans le constructeur de la classe de dialogue et, d'autre part, utiliser un slot unique (qui dispose d'un paramètre lui permettant de recevoir une valeur indiquant lequel des boutons du groupe a été cliqué), ce qui nous épargnera la création de 28 fonctions quasiment identiques.

2 - Création du projet et dessin de l'interface

En vous inspirant de la procédure décrite dans le TD 3, créez un projet nommé TD08 ☐.

Eliminez tous les widgets du dialogue, et remplacez les par un bouton nommé `carte` ☐.

Sélectionnez ce bouton☐, copiez-le ☐ et collez-le 27 fois ☐.

> Le raccourci clavier correspondant à l'action "Coller" est [Ctrl V]. Les copies successives s'empilent les unes sur les autres, de sorte qu'elles n'ont pas d'effet visible. En revanche, Qt Designer numérote judicieusement les boutons ainsi créés : `carte_2`, `carte_3`, etc. Il vous est donc facile de continuer à faire des copies jusqu'à ce que le bouton `carte_28` apparaisse... Ne vous préoccupez pas de la disposition de ces cartes, il faut simplement qu'il y en ait 28.

Créez un buttonGroup (menu "Tools", catégorie "Containers") englobant la "pile" de 28 boutons ☐.

Ouvrez, si nécessaire, la fenêtre Object Explorer (menu Window, catégorie Views) et vérifiez que les 28 cartes sont bien **DANS** le `groupePyramides` (cf ci-contre) ☐.

Name	Class
TD08Dialog	QDialog
groupePyramides	QButtonGroup
carte	QPushButton
carte_2	QPushButton
carte_3	QPushButton
carte_4	QPushButton
carte_5	QPushButton

Rebaptisez ce buttonGroup `groupePyramides` et donnez à ses propriétés geometry/width et geometry/height les valeurs 733 et 273 ☐.

Eliminez le texte correspondant à la propriété "title" du `groupePyramides` ☐.

Ajustez la taille du dialogue de façon à ce que le `groupePyramides` y tienne et qu'il reste un peu de place en bas pour la pioche et le tas ☐.

> Si vous le souhaitez, vous pouvez aussi modifier la propriété PaletteBackgroundColor du dialogue, de façon à ce que les pyramides soient visuellement plus saillantes.

Créez un slot nommé `descendre(int)` et associez-lui l'événement `clicked(int)` du `groupePyramides` ☐.

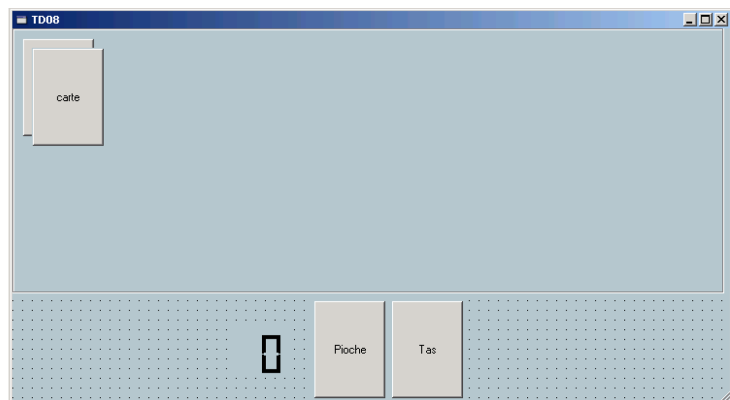
> La fonction correspondant à ce slot ne sera pas exécutée lorsque l'utilisateur cliquera sur le buttonGroup, mais lorsqu'il cliquera sur l'un des boutons qui figurent dans ce groupe.

Placez deux boutons nommés `b_pioche` et `b_tas` dans la partie inférieure du dialogue, et donnez à leurs propriétés geometry/width et geometry/height les valeurs 75 et 102 ☐.

Créez un slot nommé `piocher()` et associez-le à l'événement `clicked()` du bouton `b_pioche` ☐.

Ajoutez finalement un QLCDNumber (menu "Tools", catégorie "Display") nommé `LCDreste` ☐, qui servira à indiquer au joueur combien de cartes sont encore disponibles dans la pioche.

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



Enregistrez votre travail (Menu "File", commande "Save") ☐, puis revenez à Visual C++ ☐ et procédez aux modifications de rigueur sur la fonction `main()`, de façon à ce qu'elle instancie la classe `TD08DialogImpl` ☐.

3 - La classe de dialogue

La première chose à faire est de créer les trois variables membre dont notre analyse préalable a fait apparaître la nécessité. Dans l'onglet "File View" de la fenêtre "Workspace", ouvrez l'onglet "Header Files" et double-cliquez sur l'entrée "td08dialogimpl.h" ☐.

Complétez la définition de la classe (10-12), sans oublier (2-4) les inclusions nécessaires ☐.

```
1  #include "td08dialog.h"
2  #include "qmap.h"
3  #include "qvaluelist.h"
4  #include "carte.h"
5  class TD08DialogImpl : public TD08Dialog
6  {
7     Q_OBJECT
8  public:
9     TD08DialogImpl( QWidget* parent=0, const char* name=0, bool modal=FALSE, WFlags f=0 );
   //variables membre
10    QMap <int, CCarte> pyramides;
11    QValueList <CCarte> pioche;
12    CCarte sommetTas;
13 };
```

La variable `pyramides` est de type `QMap <int, CCarte>`, ce qui signifie qu'il s'agit d'une collection dans laquelle une valeur entière nous permettra d'accéder à la carte correspondante. Cette association `int/CCarte` mérite peut-être quelques explications complémentaires.

Que viennent faire des `int` dans cette histoire ?
Nous avons vu que la fonction `descendre()`, qui correspond au signal émis lorsque l'utilisateur clique sur l'une des cartes des pyramides, dispose d'un paramètre de type `int`. Cette fonction va recevoir une valeur entière, qui est la "signature" du bouton ayant déclenché son exécution. Représenter les pyramides à l'aide d'une `QMap` dont les clés sont des `int` permet donc à la fonction `descendre()` d'utiliser directement cette signature pour accéder à la carte concernée.

Des `CCarte` ? Quésaco ?
Pour l'instant, nous n'en savons rien. A mesure que nous avancerons dans la mise au point du programme, les fonctionnalités dont la classe `CCarte` devra disposer apparaîtront progressivement. La définition de cette classe sera donc la dernière étape de notre projet.

Le constructeur

Plusieurs éléments de la situation initiale de jeu manquent encore, et leur mise en place incombe assez naturellement au constructeur de notre classe de dialogue. Celui-ci va notamment devoir positionner les boutons représentant les pyramides et distribuer les cartes. Pour pratiquer cette distribution, nous allons créer une collection contenant toutes les cartes, ce qui nous permettra de procéder à un tirage au hasard (sans remise) parmi les éléments de cette collection.

Création du jeu de carte

La fonction qui permet d'obtenir un "tirage au hasard" produit des valeurs entières (cf. Annexe 2). L'utilisation de ces valeurs pour extraire des éléments d'une collection sera d'une grande simplicité si la collection en question est une `QMap` dont les clés sont des entiers. La création du jeu relève donc d'une simple boucle ajoutant 52 cartes à la collection, chaque carte se singularisant d'une part par leur valeur (de 0 pour un as à 12 pour un roi, cette valeur étant obtenue en calculant le reste de la division par 13 du numéro de la carte) et, d'autre part, par l'image qu'elle utilise pour se dessiner à l'écran. Ces images sont contenues dans un répertoire baptisé "cartes", sous la forme de fichiers nommés "0.png" (l'as de pique), "1.png" (le deux de pique), …, "51.png" (le roi de trèfle). Remarquez (16) l'utilisation de la fonction `arg()` pour générer les noms de fichiers à partir de la chaîne initiale (11).

```
1  #include "td08dialogimpl.h"
2  #include "time.h"
3  #include "qbuttongroup.h"
4  #include "qmessagebox.h"
5  #include "qlcdnumber.h"
6  #include "qpushbutton.h"
```

```
7  TD08DialogImpl::TD08DialogImpl(QWidget* parent, const char* name, bool modal,
                                  WFlags f)  : TD08Dialog( parent, name, modal, f )
8  {
9  srand(time(0)); //initialisation du générateur de nombre pseudo-aléatoires
   //création du jeu de cartes : une QMap pour tirer facilement une carte au hasard
10 QMap <int, CCarte> leJeu;
11 QString fichier = "cartes/%1.png";
12 CCarte aAjouter;
13 int n = 0;
14 for(n=0; n < 52 ; ++n)
15    {
16    aAjouter.devient(n % 13, fichier.arg(n));   // 1
17    leJeu[n] = aAjouter;
18    }
```

Nous devons prendre note au passage d'une première caractéristique de la classe `CCarte` :

1 : La classe `CCarte` doit disposer d'une fonction membre nommée `devient()` acceptant un argument de type `int` et un argument de type `QString`. Cette fonction fait adopter à l'instance au titre de laquelle elle est appelée la valeur et le fichier image spécifiés par ses paramètres.

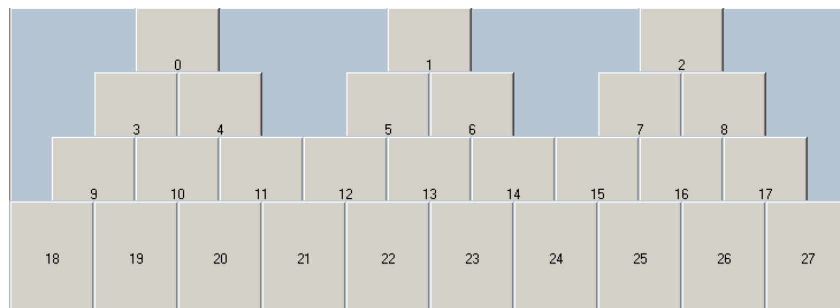
Vous pouvez trouver ici un dossier compressé contenant des images au format .png pouvant être utilisées pour représenter les cartes. La ligne *4* suppose que ce dossier à été décompressé et placé dans celui qui contient le projet TD08. La cohérence entre valeurs des cartes et images affichées repose, bien entendu, sur le respect d'une convention de dénomination des fichiers : 0.png, 13.png, 23.png et 39.png doivent représenter les as, 1.png, 14.png, 24.png et 40.png les deux, et ainsi de suite.

Positionnement des boutons et distribution des cartes

La distribution des cartes se fait en deux temps : 28 cartes sont tout d'abord choisies et placées dans les pyramides *(21)*, et les autres cartes sont ensuite placées dans la pioche *(26-27)*.

```
19 for(n=0 ; n < 28 ; ++n)
20    {//on choisit 28 cartes pour garnir les pyramides
21    pyramides[n] = tirerUneCarteDans(leJeu);
22    QButton * unBouton = groupePyramides->find(n);   // 2
23    unBouton->setGeometry(calculeGeometrie(n));
24    pyramides[n].associeBouton(unBouton, n > 17);//on voit les cartes 18 à 27
25    }
26 while(!leJeu.isEmpty()) //on met le reste dans la pioche
27    pioche.append(tirerUneCarteDans(leJeu));
```

En plus de son transfert dans la `QMap`, le placement d'une carte dans les pyramides s'accompagne de la mise en place du bouton correspondant. L'adresse de ce bouton est obtenue *(22)* à l'aide d'une fonction membre de la classe `QButtonGroup`, et il faut lui donner la taille requise et le positionner au bon endroit sur le dialogue. Cette opération est effectuée *(23)* en appelant la fonction `setGeometry()`, mais quelques calculs sont nécessaires pour garantir un arrangement correct des boutons. Ces calculs sont confiés à une fonction `calculeGeometrie()`, qu'il nous faudra définir de façon à ce que les boutons soient disposés ainsi :



Lorsqu'une des cartes des pyramides descendra sur le tas, il sera nécessaire de modifier l'apparence du bouton correspondant, et le plus simple est sans doute de confier l'adresse de ce bouton à la carte concernée. Cette association est réalisée *(28)* par la fonction `associeBouton()`, qui reçoit aussi un second paramètre indiquant si la carte doit être rendue visible.

Deux opérations sont encore nécessaires pour que la partie puisse commencer : le lien doit être établi entre le `sommetDuTas` et le `QPushButton` qui sert à le rendre visible *(22)* et une première carte doit être tirée de la pioche et placée au sommet du tas *(29)*.

```
   //début de la partie
28 sommetTas.associeBouton(b_tas, true);
29 piocher();
30 }
```

Donnez au corps de la fonction `TD08DialogImpl()` le contenu que nous venons de décrire ☐ et remarquez que la ligne *24* introduit une deuxième contrainte pour la classe `CCarte`:

2 : La classe `CCarte` doit comporter une fonction membre nommée `associeBouton()` acceptant deux paramètres. Le premier est l'adresse d'un `QPushButton` que la fonction doit stocker dans l'instance au titre de laquelle elle est invoquée. Le second paramètre est un booléen qui indique si la carte doit être rendue visible.

La fonction `tirerUneCarteDans()`

Cette fonction dispose d'un paramètre qui est une référence à une collection de cartes, et son rôle est de choisir l'une des cartes, de la supprimer de la collection et d'en renvoyer la valeur.

> On ne peut se contenter d'un paramètre de type `QMap<int, CCarte>`, qui conduirait la fonction à disposer de son propre jeu de cartes. Retirer la carte choisie de ce jeu local resterait sans effet sur le jeu créé par la fonction appelante, et nous n'aurions donc pas un tirage *sans remise*.

Pour tirer une carte dans la collection, nous devons choisir un nombre compris entre 0 et le nombre de cartes présentes dans cette collection.

> S'il y a 52 cartes, elles ont pour clés 0, 1, 2,..., 51. L'intervalle de choix inclut donc sa borne inférieure (zéro), mais exclut sa borne supérieure (le nombre de cartes disponibles).

Nous savons que la fonction `rand()` renvoie un `int` dont la valeur est imprévisible. Pour ramener cette valeur dans notre intervalle de choix *(3)*, nous pouvons à nouveau utiliser l'opérateur `%` (rappel : le reste de la division entière est strictement inférieur au diviseur...).

```
1 CCarte TD08DialogImpl::tirerUneCarteDans (QMap <int, CCarte> & jeu)
2 {
3 int choix = rand() % jeu.count(); //un nombre de l'intervalle [0, jeu.count()[
4 CCarte tiree = jeu[choix];
5 jeu[choix] = jeu[jeu.count()-1];
6 jeu.remove(jeu.count()-1);
7 return tiree;
8 }
```

Une fois ce choix effectué, la carte tirée peut être mise de côté *(4)* et sa valeur éliminée du jeu. Plutôt que de retirer la valeur figurant à la position tirée (ce qui laisserait un "trou" dans le jeu, c'est à dire une clé inférieure au nombre de cartes et pour laquelle il n'existerait plus aucune valeur), il est préférable de remplacer la valeur de la carte choisie par celle de la dernière carte du jeu *(5)* et de supprimer ensuite celle-ci *(6)*.

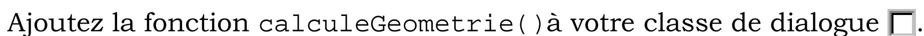
Ajoutez la déclaration de la fonction `tirerUneCarteDans()`dans la définition de la classe `TD08DialogImpl` ☐ et ajoutez la définition de cette fonction dans le fichier td08dialogimpl.cpp ☐.

La fonction `calculeGeometrie()`

Bien qu'indispensable au fonctionnement de notre programme, cette fonction est sans grand intérêt pédagogique : elle ne fait que déterminer la position d'un bouton étant donné son numéro. Un programmeur allergique à l'arithmétique aurait pu en faire un `switch` à 28 `case` adoptant des valeurs choisies empiriquement, et la version qui suit n'est peut être pas beaucoup plus intéressante (un fichier texte est disponible ici pour copier/coller) :

```
1 QRect TD08DialogImpl::calculeGeometrie(const int qui)
2 {
3 const int LARGEUR_CARTE = 75;
4 const int HAUTEUR_CARTE = 102;
5 const int DELTA_V = 56;
6 const int DELTA_H = LARGEUR_CARTE - 2;
```

```
7   QRect reponse;
8   if (qui < 3)
9      {
10     reponse.setX(groupePyramides->x() + (3 * qui + 1.5) * DELTA_H);
11     reponse.setY(groupePyramides->y());
12     }
13  if(qui >= 3 && qui < 9)
14     {
15     reponse.setX(groupePyramides->x() + (qui - 2 + (qui>4) + (qui>6)) * DELTA_H);
16     reponse.setY(groupePyramides->y() + DELTA_V);
17     }
18  if(qui >= 9 && qui < 18)
19     {
20     reponse.setX(groupePyramides->x() + (0.5 + (qui-9)) * DELTA_H);
21     reponse.setY(groupePyramides->y() + 2 * DELTA_V);
22     }
23  if(qui >= 18)
24     {
25     reponse.setX(groupePyramides->x() + (qui-18) * DELTA_H);
26     reponse.setY(groupePyramides->y() + 3 * DELTA_V);
27     }
28  reponse.setWidth(LARGEUR_CARTE);
29  reponse.setHeight(HAUTEUR_CARTE);
30  return reponse;
31  }
```



Ajoutez la fonction `calculeGeometrie()`à votre classe de dialogue ☐.

La fonction `descendre()`

Lorsque le joueur clique sur l'une des cartes qui figurent dans les pyramides, cette carte doit descendre sur le tas (à condition, toutefois, que les règles du jeu autorisent ce mouvement). Le début de la fonction descendre n'est donc pas très difficile à écrire :

```
1   void TD08DialogImpl::descendre(int qui)
2   {
    //vérifie la légalité du coup
3   int difference = abs(pyramides[qui].valeur() - sommetTas.valeur());   (3)
4   if(difference != 1 && difference != 12)
5      return; //les valeurs ne sont ni consécutives ni 0 et 12 : mouvement illégal
```

La fonction `abs()` renvoie la valeur absolue de son argument. Elle permet donc *(3)* de calculer l'écart de valeur entre les deux cartes concernées, sans avoir à s'occuper de qui est la plus petite. Un écart de 12 signifie que l'une des deux cartes est un roi (valeur 12) et l'autre un as (valeur 0), ce qui correspond à un mouvement autorisé.

3 : La classe `CCarte` doit comporter une fonction membre nommée `valeur()` renvoyant la valeur de l'instance au titre de laquelle elle est invoquée.

Si le mouvement est légal, il faut l'effectuer. Le transfert de la carte désignée sur le sommet du tas *(6)* implique diverses manipulations sur les variables membre de la classe `CCarte`, et comme nous ignorons pour l'instant le détail de l'organisation interne de celle-ci, il est préférable de confier cette tâche à une de ses fonctions membre :

4 : La classe `CCarte` doit comporter une fonction membre nommée `recoit()` ayant pour effet de transférer la valeur et l'image de l'instance qui lui est passée comme argument dans l'instance au titre de laquelle elle est invoquée. L'image doit apparaître dans le bouton associé à la carte réceptrice, et le bouton associé à l'autre carte doit disparaître.

```
    //effectue la descente
6   sommetTas.recoit(pyramides[qui]);   (4)
7   pyramides.remove(qui);
8   if(pyramides.count() == 0) //partie finie ?
9      QMessageBox::information(this, "Pharaon", "Bravo, vous avez gagné !");
```

Il faut ensuite mettre à jour l'affichage, car il est possible que certaines cartes aient été libérées par la disparition de la carte descendue. Différentes méthodes sont envisageables, mais toutes sont compliquées par les irrégularités de la disposition des cartes. La méthode décrite ci-dessous ne tient pas compte de la position initiale de la carte qui vient d'être descendue, mais examine toutes les cartes pour vérifier que celles qui ne sont pas bloquées sont bien affichées.

Pour les neuf premières cartes, la solution la plus claire me semble être celle qui s'appuie sur une énumération explicite des obstacles qui les bloquent :

```
   //retourne les cartes éventuellement libérées
10 QMap <int, int> blocage;
11 blocage[0] = 3;          //la carte 0 est bloquée par la 3 et la 4
12 blocage[1] = 5;          //la carte 1 est bloquée par la 5 et la 6
13 blocage[2] = 7;          //la carte 2 est bloquée par la 7 et la 8
14 blocage[3] = 9;          //la carte 3 est bloquée par la 9 et la 10
15 blocage[4] = 10;         //la carte 4 est bloquée par la 10 et la 11
16 blocage[5] = 12;         //la carte 5 est bloquée par la 12 et la 13
17 blocage[6] = 13;         //la carte 6 est bloquée par la 13 et la 14
18 blocage[7] = 15;         //la carte 7 est bloquée par la 15 et la 16
19 blocage[8] = 16;         //la carte 8 est bloquée par la 16 et la 17
```

Il suffit ensuite d'indiquer que si une carte est présente dans les pyramides et que les deux obstacles qui la concernent ont été retirés, alors cette carte doit apparaître :

```
20 int n;
21 for(n=0 ; n < 9 ; ++n)
22    if (pyramides.contains(n))
23       if(!pyramides.contains(blocage[n]) && !pyramides.contains(blocage[n]+1))
24          pyramides[n].montreToi();   5
```

5 : La classe `CCarte` doit comporter une fonction `montreToi()` capable de faire apparaître l'image attribuée à la carte au titre de laquelle elle est appelée dans le bouton associé à cette carte.

Pour les cartes 9 à 17, le problème est bien plus simple, puisqu'on on obtient les numéros des cartes bloquantes en ajoutant respectivement 9 et 10 au numéro de la carte considérée :

```
25 for(n=9 ; n < 18 ; ++n) //pour n>9, la carte n est bloquée par la n+9 et la n+10
26    if (pyramides.contains(n))
27       if(!pyramides.contains(n+9) && !pyramides.contains(n+10))
28          pyramides[n].montreToi();
29 }
```

Ajoutez la fonction `descendre()` à votre classe de dialogue ☐.

La fonction `piocher()`

Le rôle principal de cette fonction est de transférer *(3-4)* la première carte de la pioche au sommet du tas. Cette fonction s'occupe aussi d'afficher *(6)* le nombre de cartes encore disponibles dans la pioche et de masquer le bouton correspondant lorsque celle-ci est épuisée *(7-8)* :

```
1 void TD08DialogImpl::piocher()
2 {
3 sommetTas.recoit(pioche.first()); //le sommet de la pioche passe sur le tas
4 pioche.pop_front(); //elle n'est donc plus dans la pioche
5 int reste = pioche.count();
6 LCDreste->display(reste);
7 if(reste == 0)
8    b_pioche->hide(); //la fonction hide() permet de masquer un widget
9 }
```

Ajoutez la fonction `piocher()` à votre classe de dialogue ☐.

4 - La classe CCarte

Ajoutez une classe CCarte à votre projet ☐. La mise au point de la classe de dialogue nous a conduit à faire cinq "promesses" concernant cette classe :

	Origine	Nature de la promesse
1	TD08DialogImpl()	void devient(int valeur, QString cheminFichier)
2	TD08DialogImpl()	void associeBouton(QButton *b, bool visible)
3	descendre()	int valeur();
4	descendre()	void recoit(CCarte & autre)
5	descendre()	void montreToi()

Définition de la classe

On peut satisfaire ces exigences en munissant la classe CCarte de trois variables membre :
- Un int pour la valeur de la carte représentée (points 1 et 3).
- Une QPixmap pour l'image représentant la carte (points 1 et 5).
- Un "pointeur sur QButton" pour stocker l'adresse du bouton éventuellement associé à la carte (points 2 et 5).

Si l'on ajoute à ces trois variables les cinq fonctions explicitement requises par le cahier des charges, la classe CCarte devra être définie ainsi ☐ :

```
1  #include "qstring.h"
2  #include "qbutton.h"
3  #include "qpixmap.h"
4  class CCarte
5  {
6  public:
7     int val;
8     QPixmap image;
9     QButton * laCase;
   //fonctions membre
10    void devient(int v, QString chemin);
11    void associeBouton(QButton *b, bool visible);
12    int valeur() const { return val; } //cette fonction est définie ici même
13    void recoit(CCarte & autre);
14    void montreToi();
15 };
```

La fonction devient()

Le rôle de cette fonction est de transférer la valeur de ses paramètres dans les variables membre correspondante *(3-4)*, mais une précaution doit être prise : le membre laCase doit recevoir une valeur indiquant qu'il ne pointe pour l'instant sur aucun QButton (c'est la fonction associeBouton() qui est chargée de donner à ce membre une valeur valide, et elle n'est pas appelée pour toutes les cartes : celles de la pioche ne sont jamais associées à aucun bouton). Dans le cas des pointeurs, la valeur nulle (0) sert à manifester qu'ils ne désignent aucun objet, et la tradition veut que, dans ce contexte, zéro s'écrive NULL.

```
1  void CCarte::devient(int v, QString chemin)
2  {
3  val = v;
4  image.load(chemin);
5  laCase = NULL; //donne au pointeur la valeur 0
6  }
```

Ajoutez la définition de la fonction CCarte::devient()à votre fichier carte.cpp ☐.

La fonction associeBouton()

Cette fonction reçoit deux arguments : le premier est simplement l'adresse qui doit être stockée dans le membre laCase *(3)*, et le second indique si la carte doit être rendue visible. Dans ce programme, les boutons associés à des cartes non visibles doivent être désactivés *(4)*, faute de

quoi le joueur pourrait essayer de descendre les cartes correspondantes sans attendre qu'elles soient débloquées. Si la carte doit être visible, il faut, évidemment, qu'elle apparaisse (5-6).

```
1  void CCarte::associeBouton(QButton *b, bool visible)
2  {
3  laCase = b;
4  laCase->setEnabled(visible);
5  if(visible)
6     montreToi();
7  }
```

Ajoutez la définition de la fonction `CCarte::associeBouton()` à votre fichier carte.cpp ☐.

La fonction `valeur()`

Le seul commentaire que mérite cette fonction (définie dans le .h) est une justification énergique de son existence : l'expérience montre que tout va beaucoup mieux si l'on s'interdit absolument d'accéder aux variables membre autrement qu'en passant par une fonction membre.

> Dans la version actuelle de notre programme, seule la fonction `TD08DialogImpl::descendre()` fait appel à la fonction `CCarte::valeur()`, pour déterminer si le mouvement est légal :
>
> ```
> int difference = abs(pyramides[qui].valeur() - sommetTas.valeur());
> ```
>
> Il pourrait sembler plus simple d'accéder directement à la variable membre concernée, ce qui éviterait d'avoir à créer la fonction `valeur()` :
>
> ```
> int difference = abs(pyramides[qui].val - sommetTas.val);
> ```
>
> Cette approche est déconseillée, car elle conduit à placer le nom, le type et la signification d'une variable membre de `CCarte` dans le cahier des charges de cette classe. A l'usage, il s'avère plus souple et plus sûr de n'exprimer les cahiers des charges des classes qu'en termes de fonctions.

La fonction `recoit()`

En plus des transferts de données qui transforment la carte réceptrice en une copie de l'autre carte (3-4), cette fonction garantit l'affichage de la nouvelle valeur de la carte réceptrice (5) et la disparition de l'autre (6-7) :

```
1  void CCarte::recoit(CCarte & autre)
2  {
3  val = autre.val;
4  image = autre.image;
5  montreToi();
6  if(autre.laCase != NULL)
7     autre.laCase->hide();
8  }
```

Ajoutez la définition de la fonction `CCarte::recoit()` à votre fichier carte.cpp ☐.

La fonction `montreToi()`

Une carte ne peut se montrer que si un bouton lui a été associé (3-4) et, dans ce jeu, un bouton sur lequel la carte est visible doit être rendu actif (5). Pour rendre l'image visible, il suffit de la transmettre au bouton à l'aide de la fonction `setPixmap()` (6) :

```
1  void CCarte::montreToi()
2  {
3  if(laCase == NULL)
4     return;
5  laCase->setEnabled(true);
6  laCase->setPixmap(image);
7  }
```

Ajoutez la définition de la fonction `CCarte::montreToi()` à votre fichier carte.cpp ☐.

5 - Exercice

En vous inspirant du code que nous venons de décrire, créez un programme qui permette de faire plusieurs parties de suite sans avoir à le relancer.