***C**entre **I**nformatique pour les **L**ettres et les **S**ciences **H**umaines*

TD 5 : Un jeu de Tic tac toe[1]

1 - Comment pourrait-on programmer ça ? 2

Choix des contrôles 2
Alternance des marques 3
Remise du jeu dans son état initial 3
Détection de la victoire éventuelle d'un des joueurs 3

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

3 - Ecriture du code 4

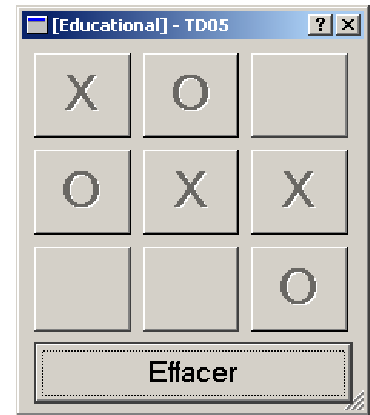
La fonction `f_effacer()` 4
Effacement des marques 4
Rendre toutes les cases utilisables 4
Remettre à zéro les alignements 5
Désignation du joueur qui va commencer 5
Première ébauche d'une fonction associée à un bouton "case" 6
Matérialisation du coup 6
Mise à jour des alignements 6
En cas de victoire 7
La fonction `codeCommunATousLesCoups()` 7
Les fonctions associées aux boutons "case" 8
Création du type `EJoueur` 8
La classe `CAlignement` 8
Création de la classe 8
Création des membres 9
La fonction `reset()` 9
La fonction `ajouteMarque()` 9
Les variables membre de la classe `TD05DialogImpl` 9

4 - Questions et exercices 10

5 - Qu'avons-nous appris ? 10

[1] Selon le Trésor de la Langue Française, le morpion est un "jeu qui consiste pour chacun des deux adversaires à placer à tour de rôle un signe distinctif (croix ou cercle) sur du papier quadrillé pour s'efforcer d'obtenir le plus rapidement une file continue de cinq signes dans n'importe quelle direction". C'est donc improprement que ce nom est parfois utilisé pour désigner le jeu limité à neuf cases dont il est question ici. Bien qu'il sonne fâcheusement anglo-saxon, le nom "tic tac toe" est bien plus usité que "OXO", le seul autre nom que je connaisse pour ce jeu.

L'objectif du TD 05 est d'illustrer l'utilisation de fonctions dotées de paramètres. Le TD 05 offre en outre une occasion d'employer effectivement une classe créée spécialement pour représenter les données de notre problème (et non pour représenter un dialogue).



L'interface utilisateur du programme réalisé au cours du TD 05

Le programme servant de prétexte à cette illustration permet à deux utilisateurs d'employer l'ordinateur pour remplacer le sable habituellement utilisé pour jouer à un jeu de plage bien connu : il s'agit, sur un damier 3 x 3, de parvenir à aligner trois marques identiques sur une ligne, une colonne ou une diagonale (chacun des joueurs utilise une marque différente, et ils choisissent tour à tour une des cases encore libres pour y placer leur marque).

Les joueurs placent simplement leur marque en cliquant sur la case choisie (le programme passe automatiquement d'une marque à l'autre à chaque coup joué). Lorsque l'un des joueurs parvient à aligner trois de ses marques, un message annonçant sa victoire est affiché. Le bouton "Effacer" permet, à tout moment, de commencer une nouvelle partie.

1 - Comment pourrait-on programmer ça ?

Le cahier des charges esquissé ci-dessus révèle trois caractéristiques essentielles du programme :
- il doit être capable de placer alternativement des ronds et des croix dans les cases sur lesquelles les utilisateurs cliquent ;
- il doit être capable de détecter (et d'annoncer) la victoire éventuelle d'un des deux joueurs ;
- il doit enfin être capable de remettre le jeu dans son état initial (les cases doivent être vides).

Choix des contrôles

Une des toutes premières décisions à prendre concerne la façon dont nous allons matérialiser les cases dans lesquelles se déroule le jeu.

Une première façon de faire serait de disposer 9 multiLineEdit, ce qui nous permettrait de modifier le texte affiché en employant l'opérateur d'insertion introduit lors du TD 4. Les cases du jeu ne servent toutefois pas qu'à afficher les marques : elles doivent également réagir lorsqu'un utilisateur clique dessus. Cette caractéristique correspond plus à la vocation d'un bouton qu'à celle d'un multiLineEdit, ce qui suggère une seconde hypothèse.

Si les cases sont représentées par des boutons, nous n'aurons aucune difficulté à les faire réagir en cas de clic, mais nous allons devoir faire en sorte que le programme en modifie les intitulés. C'est cette méthode que nous allons choisir, car elle présente plusieurs avantages :

- Elle est plus conforme aux conventions habituelles des interfaces graphiques, puisque le changement d'intitulé d'un bouton sur lequel on vient de cliquer est un phénomène assez banal, alors que l'action consistant à cliquer sur une zone de texte est normalement destinée à y faire apparaître la barre verticale clignotante associée à l'édition de texte[2].

- Cette "orthodoxie" du changement d'intitulé d'un bouton a deux conséquences positives : d'une part les utilisateurs comprendront plus facilement ce qu'ils sont censés faire, et, d'autre part, l'éditeur de dialogue et la librairie graphique vont nous faciliter la tâche (alors que l'appel d'une fonction en cas de clic sur une zone de texte doit être mis en place de façon "artisanale", ce qui est un peu plus délicat).

- La technique permettant de changer l'intitulé d'un bouton permet également d'en modifier d'autres caractéristiques, ce qui va nous permettre de rendre certains boutons temporairement inactifs (pour empêcher un joueur de placer sa marque dans une case déjà occupée par son adversaire, par exemple).

Le choix de boutons pour figurer les cases du jeu a une conséquence majeure sur l'architecture de notre programme : le clic sur chacune de ces cases va déclencher l'exécution d'une fonction différente (celle associée au bouton concerné).

[2] Ce qu'on appelle "donner le focus" à la zone de texte.

Alternance des marques

Lorsque nous allons modifier l'intitulé d'un bouton, un problème va se poser : quel est le nouvel intitulé qui doit apparaître ? La façon la plus simple de résoudre ce problème est sans doute de créer une variable membre de la classe du dialogue qui permettra d'identifier le joueur dont c'est le tour de jouer. Chacune des fonctions associées aux boutons aura accès à cette variable (puisque ces fonctions sont également membres de la classe du dialogue) et s'en servira pour déterminer la marque qui doit apparaître. Chacune de ces fonctions prendra évidemment soin de modifier ensuite la variable en question, pour que le coup suivant corresponde à l'autre joueur.

Remise du jeu dans son état initial

Cette opération ne présente guère de difficulté : chacun des 9 boutons doit retrouver un intitulé vierge et être rendu disponible pour qu'un joueur puisse y placer sa marque.

Détection de la victoire éventuelle d'un des joueurs

De nombreuses approches sont envisageables pour aborder ce problème, et en examiner plusieurs pour faire ressortir leurs avantages et inconvénients respectifs dépasserait notre propos actuel (qui n'est, rappelons-le, que de réaliser un petit programme illustrant l'usage de paramètres...). Nous allons donc opter pour une méthode qui présente l'avantage de ne pas être trop difficile à expliquer et de ne nécessiter la mise en œuvre d'aucune technique que nous n'ayons déjà utilisée.

> Cette méthode présente, en revanche, l'inconvénient de conduire à une certaine profusion de variables membre, un défaut qui doit normalement être évité. L'intérêt des techniques de structuration des données qui permettent de pallier cet inconvénient ne vous en paraîtra que plus manifeste lorsque nous les introduirons !

Remarquons tout d'abord qu'il n'existe que huit alignements permettant de gagner à ce jeu : trois lignes, trois colonnes et deux diagonales. Il est donc envisageable de tenir à jour l'état de réalisation de chacun de ces huit alignements, en créant à cet effet huit variables membre.

Chacune des cases n'est pertinente que pour certains alignements. La case centrale de la ligne supérieure, par exemple, ne contribue à la réalisation que de deux alignements : la ligne supérieure et la colonne centrale. La case centrale, par contre, contribue à la réalisation de la colonne centrale, de la ligne centrale et des deux diagonales. Comme l'occupation de chacune des cases donne lieu à l'appel d'une fonction spécifique, il est facile de faire en sorte que la fonction en question modifie l'état de réalisation des alignements concernés, et seulement de ceux-ci.

Si l'on envisage les choses du point de vue de l'un des alignements, il s'agit de tenir à jour le nombre de marques placées par chacun des joueurs. Lorsque l'un de ces nombres atteint trois, l'alignement est réalisé et la partie est gagnée pour le joueur correspondant. Comme une simple variable de type numérique ne peut contenir qu'une seule valeur et que nous devons en stocker deux, nous serons amenés à créer un type spécifique, destiné à représenter l'état d'un alignement.

Tous les détails ne sont, certes, pas réglés, mais nous avons maintenant une représentation assez claire de comment nous allons nous y prendre et nous pouvons donc passer aux actes.

2 - Création du projet et dessin de l'interface

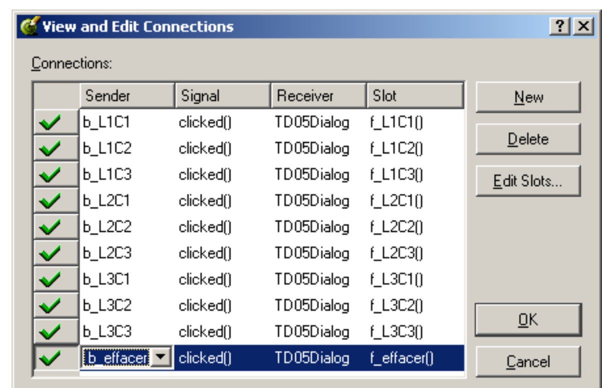
En vous inspirant de la procédure décrite dans le TD 3, créez un projet nommé TD05 ☐ basé sur un dialogue ressemblant à celui représenté page 2 ☐.

> Commencez par créer un bouton et donnez lui les caractéristiques souhaitées (size policy horizontale et verticale "prefered/prefered" et font de grande taille) avant de le dupliquer par copier/coller.

Les boutons figurant les cases doivent être nommés d'une façon rappelant la ligne et la colonne occupée : b_L1C1, b_L1C2, b_L1C3, b_L2C1, b_L2C2, b_L2C3, b_L3C1, b_L3C2 et b_L3C3 ☐.

Le bouton d'effacement est baptisé b_effacer ☐.

Créez les slots et établissez les connexions nécessaires (cf. tableau ci-contre) ☐.



Sender	Signal	Receiver	Slot
b_L1C1	clicked()	TD05Dialog	f_L1C1()
b_L1C2	clicked()	TD05Dialog	f_L1C2()
b_L1C3	clicked()	TD05Dialog	f_L1C3()
b_L2C1	clicked()	TD05Dialog	f_L2C1()
b_L2C2	clicked()	TD05Dialog	f_L2C2()
b_L2C3	clicked()	TD05Dialog	f_L2C3()
b_L3C1	clicked()	TD05Dialog	f_L3C1()
b_L3C2	clicked()	TD05Dialog	f_L3C2()
b_L3C3	clicked()	TD05Dialog	f_L3C3()
b_effacer	clicked()	TD05Dialog	f_effacer()

Les connexions signal/slots du TD 5

3 - Ecriture du code

La fonction la plus facile à écrire est sans doute celle qui efface toutes les cases et remet le programme dans un état qui permet aux utilisateurs de faire une nouvelle partie. Commencer par écrire cette fonction va, par ailleurs, nous permettre de mettre en place les bases de notre gestion de l'alternance des marques et de la détection de la fin d'une partie.

La fonction `f_effacer()`

Cette fonction, qui sera appelée lors du lancement du programme (pour préparer la première partie) et entre chaque partie (lorsque l'utilisateur cliquera sur le bouton "Effacer") doit prendre quatre mesures :
- Eliminer les marques éventuellement présentes dans certaines cases.
- Rendre toutes les cases accessibles au premier joueur.
- Placer chacune des variables représentant une possibilité de victoire dans un état signifiant qu'aucune des cases ne la concernant n'est actuellement occupée.
- Choisir l'identité du joueur qui va commencer.

Effacement des marques

Plutôt que de chercher à déterminer quelles cases ont réellement besoin d'être effacées, il est plus efficace d'ordonner à chacun des boutons d'afficher une chaîne vide :

```
   //Cette fonction est appelée en début de programme et
   //quand l'utilisateur clique sur le bouton [Effacer]
1  void TD05DialogImpl::f_effacer()
2  {
   //vider toutes les cases
3  b_L1C1->setText("");
4  b_L1C2->setText("");
5  b_L1C3->setText("");
6  b_L2C1->setText("");
7  b_L2C2->setText("");
8  b_L2C3->setText("");
9  b_L3C1->setText("");
10 b_L3C2->setText("");
11 b_L3C3->setText("");
```

Rendre toutes les cases utilisables

Ceci revient à rendre disponibles les boutons représentant ces cases, et une fonction de la classe `QPushButton` est prévue pour cela :

```
   //rendre tous les boutons disponibles
12 b_L1C1->setEnabled(true);
13 b_L1C2->setEnabled(true);
14 b_L1C3->setEnabled(true);
15 b_L2C1->setEnabled(true);
16 b_L2C2->setEnabled(true);
17 b_L2C3->setEnabled(true);
18 b_L3C1->setEnabled(true);
19 b_L3C2->setEnabled(true);
20 b_L3C3->setEnabled(true);
```

Les fonctions `setText()` et `setEnabled()` utilisent des paramètres qui permettent de leur indiquer respectivement quel texte doit être affiché dans le bouton et si le bouton doit être rendu actif ou inactif. Ce type de paramètres est tellement "intuitif" que nous avons commencé à l'utiliser avant même d'avoir étudié la Leçon 5 (le TD 3 utilise la fonction `display()`, qui est au `QLCDNumber` ce que `setText()` est au `QPushButton`).

Ces fonctions doivent, ici, être exécutées au titre de chacun des boutons concernés, ce qui exige que chacune d'entre-elles soit appelée neuf fois (nous ne disposons pas encore des outils qui nous permettraient d'utiliser une boucle pour procéder à ces appels répétés, et nous devons donc nous contenter de recopier plusieurs fois des lignes de code quasi identiques.

Remettre à zéro les alignements

Nous avons décidé que les huit alignements possibles allaient être représentés chacun par une instance d'une classe conçue spécialement à cet effet, que nous pouvons décider de baptiser `CAlignement`. Nous sommes donc conduits à décider que cette classe devra offrir une fonction membre permettant de placer une instance dans l'état correspondant à l'absence de marques la concernant. Le nom de cette fonction pourrait être `reset()`, un verbe souvent employé en informatique dans ce genre de contexte.

La classe `CAlignement` devra disposer d'une fonction `reset()`.

Lorsque nous nous chercherons à définir la classe `CAlignement`, l'ensemble des exigences que nous aurons formulées à son sujet au cours de l'écriture du reste du programme nous servira de guide pour décider de son fonctionnement interne.

```
   //remettre à zéro tous les alignements
21 diagonaleUn.reset();
22 diagonaleDeux.reset();
23 ligneDuHaut.reset();
24 ligneDuMilieu.reset();
25 ligneDuBas.reset();
26 colonneDeGauche.reset();
27 colonneDuMilieu.reset();
28 colonneDeDroite.reset();
```

La classe `TD05DialogImpl` devra comporter huit variables membre de type `CAlignement`

Désignation du joueur qui va commencer

Comme dans le cas de la mémorisation de l'opération choisie lors de l'utilisation de la calculette (TD 3), le recours à une variable d'un type énuméré semble ici s'imposer. Si le dialogue possède une variable membre de ce type nommée `joueurActif`, indiquer que c'est au premier joueur de placer sa marque revient simplement à donner la bonne valeur à cette variable :

```
   //indiquer que c'est au premier joueur de mettre une marque
29 joueurActif = JOUEUR_UN;
30 } //fin de la fonction f_effacer()
```

Il faudra créer un type `EJoueur` admettant les valeurs `JOUEUR_UN` et `JOUEUR_DEUX`.
Il faudra ajouter à la classe `TD05DialogImpl` un membre de type `EJoueur` nommé `joueurActif`.

Créez la fonction `f_effacer()` ☐ (clic droit sur la classe TD05DialogImpl dans l'onglet ClassView de la fenêtre Workspace, puis choix de l'option "Add member function").

Insérez dans le corps de cette fonction le code que nous venons de décrire ☐.

Insérez également, au début du fichier TD05DialogImpl.cpp, les directives qui permettent au compilateur de savoir ce qu'est un `QPushButton` et une `QMessageBox`[3] ☐, ainsi que l'appel de la fonction `f_effacer()` ☐ qui doit être exécuté en début de programme :

```
1 #include "td05dialogImpl.h"
2 #include "QPushButton.h"
3 #include "QMessageBox.h"
4
5 TD05DialogImpl::TD05DialogImpl( QWidget* parent, const char* name, bool modal, WFlags f )
6    : TD05Dialog( parent, name, modal, f )
7 {
     // Add your code
8 f_effacer();
9 }
```

[3] La classe `QMessageBox` sera utilisée pour afficher le message signalant la victoire de l'un des joueurs.

Première ébauche d'une fonction associée à un bouton "case"

Les fonctions associées aux boutons représentant les cases du jeu constituent le cœur du programme. La tâche de ces fonctions comporte trois aspects : la matérialisation du coup exprimé par le clic sur le bouton, la mise à jour des variables représentant les alignements et, en cas de victoire de l'un des joueurs, l'arrêt de la partie et l'affichage d'un message.

Essayons, dans un premier temps, d'imaginer à quoi pourrait ressembler la fonction associée au bouton situé sur la première ligne, dans la première colonne.

Matérialisation du coup

Trois opérations sont nécessaires pour matérialiser l'occupation de la case sur laquelle un joueur vient de cliquer. Il faut en effet :
- désactiver le bouton concerné (pour qu'aucun joueur ne soit en mesure de l'utiliser à nouveau durant la partie en cours)
- faire apparaître la marque correspondant à l'utilisateur qui vient de cliquer sur le bouton
- indiquer que le prochain coup sera le fait de l'autre joueur.

Ces deux dernières opérations se traduisent par des actions dépendant de l'identité du `joueurActif`. Elles donnent donc lieu à une exécution conditionnelle contrôlée par cette identité :

```
1  void TD05DialogImpl::f_L1C1()
2  {
3  b_L1C1->setEnabled(false); //désactivation de la case qui vient d'être choisie
4  if(joueurActif == JOUEUR_UN)
5     {
6     b_L1C1->setText("X");        //affichage de la marque du JOUEUR_UN
7     joueurActif = JOUEUR_DEUX;   //changement de joueur (cas 1)
8     }
9  else
10    {
11    b_L1C1->setText("O");        //affichage de la marque du JOUEUR_DEUX
12    joueurActif = JOUEUR_UN;     //changement de joueur (cas 2)
13    }
```

Mise à jour des alignements

La mise a jour des alignements n'est pas, en elle-même une opération très compliquée : il suffit de "prévenir" chacun des alignements concernés par la case qui vient d'être choisie du fait que le `joueurActif` vient d'y placer sa marque. Cette opération peut être facilement effectuée si l'on dote la classe `CAlignement` d'une fonction `ajouteMarque()`.

Un facteur supplémentaire doit cependant être pris en compte : chacun des alignements concernés risque de se voir rempli par la marque ajoutée, ce qui indique que le `joueurActif` vient de gagner la partie. Plutôt que de chercher à détecter cette situation en pratiquant un test après chaque mise à jour d'un alignement, il est plus "économique" de transmettre une même variable à chaque appel de `ajouteMarque()`, étant entendu que cette fonction ne modifiera le contenu de cette variable que si elle diagnostique la victoire du `joueurActif`.

```
14 bool partieGagnee = false;
15 ligneDuHaut.ajouteMarque(joueurActif, & partieGagnee);
16 colonneDeGauche.ajouteMarque(joueurActif, & partieGagnee);
17 diagonaleUn.ajouteMarque(joueurActif, & partieGagnee);
```

La variable `partieGagnee` étant initialisée avec la valeur fausse, il suffit que l'un des trois appels à `ajouteMarque()` donne lieu au diagnostic de victoire pour que `partieGagnee` devienne vraie.

La classe `CAlignement` doit comporter une fonction membre nommée `ajouteMarque()` et comportant deux paramètres. Le premier est de type `EJoueur` et il indique quel joueur vient d'occuper une des cases de l'alignement. Le second est l'adresse d'un booléen que la fonction doit rendre vrai si l'ajout de la marque conduit le joueur à la victoire.

Il aurait été possible de faire de ce second paramètre une référence à un booléen plutôt qu'un pointeur sur un booléen. On peut cependant préférer rendre bien visible le fait que la fonction `ajouteMarque()` est susceptible de modifier la valeur de la variable `partieGagnee`.

En cas de victoire…

Il faut rendre indisponibles tous les boutons, de façon à ce qu'il ne soit plus possible d'ajouter des marques. Pour éviter tout risque de malentendu (au cas où les joueurs n'auraient pas remarqué que l'un d'entre eux a gagné…), cet arrêt de la partie sera accompagné d'un message explicatif.

```
18 if(partieGagnee)
19    {
20    b_L1C1->setEnabled(false); //on met fin à la partie
21    b_L1C2->setEnabled(false); //en désactivant tous les boutons
22    b_L1C3->setEnabled(false);
23    b_L2C1->setEnabled(false);
24    b_L2C2->setEnabled(false);
25    b_L2C3->setEnabled(false);
26    b_L3C1->setEnabled(false);
27    b_L3C2->setEnabled(false);
28    b_L3C3->setEnabled(false);
29    QMessageBox::information(0, "Tic tac toe", "Bravo !");
30    }
31 } //fin de la fonction f_L1C1()
```

Comme dans la fonction `f_effacer()`, il est ici préférable de ne pas chercher à éviter les opérations inutiles. Déterminer si un bouton est ou non disponible nécessite un appel de fonction, et la logique conditionnelle qu'il faudrait mettre en place pour ne désactiver que les boutons disponibles alourdirait le code source sans améliorer l'efficacité du programme.

La fonction utilisée (*29*) pour afficher un message (dans une fenêtre qui lui est propre et qui se superpose à celle représentant le jeu) utilise trois paramètres. Nous reviendrons plus en détails sur son utilisation lors d'un prochain TD.

La fonction `codeCommunATousLesCoups()`

Si l'on examine notre première ébauche de fonction `f_L1C1()`, on remarque que seule la mise à jour des alignements est spécifique à ce bouton. Tout le reste de la fonction pourrait être déplacé dans une fonction à laquelle il suffirait de transmettre le verdict des alignements et l'adresse du bouton qui vient d'être cliqué pour qu'elle puisse effectuer tous les traitements nécessaires :

```
1  void TD05DialogImpl::codeCommunATousLesCoups(bool victoire,QPushButton * bouton)
2  {
   //Matérialisation du coup
3  bouton->setEnabled(false); //désactivation de la case qui vient d'être choisie
4  if(joueurActif == JOUEUR_UN)
5     {
6     bouton ->setText("X");       //affichage de la marque du JOUEUR_UN
7     joueurActif = JOUEUR_DEUX;   //changement de joueur (cas 1)
8     }
9  else
10    {
11    bouton ->setText("O");       //affichage de la marque du JOUEUR_DEUX
12    joueurActif = JOUEUR_UN;     //changement de joueur (cas 2)
13    }
   //en cas de victoire
14 if(victoire)
15    {
16    b_L1C1->setEnabled(false); //on met fin à la partie
17    b_L1C2->setEnabled(false); //en désactivant tous les boutons
18    b_L1C3->setEnabled(false);
19    b_L2C1->setEnabled(false);
20    b_L2C2->setEnabled(false);
21    b_L2C3->setEnabled(false);
22    b_L3C1->setEnabled(false);
23    b_L3C2->setEnabled(false);
24    b_L3C3->setEnabled(false);
25    QMessageBox::information(0, "Tic tac toe", "Bravo !");
26    }
27 }
```

Les fonctions associées aux boutons "case"

Si la fonction la fonction codeCommunATousLesCoups() est disponible, la fonction f_L1C1() se trouve considérablement simplifiée :

```
1 void TD05DialogImpl::f_L1C1()
2 {
3 bool partieGagnee = false;
4 ligneDuHaut.ajouteMarque(joueurActif, & partieGagnee);
5 colonneDeGauche.ajouteMarque(joueurActif, & partieGagnee);
6 diagonaleUn.ajouteMarque(joueurActif, & partieGagnee);
7 codeCommunATousLesCoups(partieGagnee, b_L1C1) ;
8 }
```

Créez les fonctions codeCommunATousLesCoups() ☐ et f_L1C1() ☐ et donnez-leur le contenu décrit ci-dessus ☐.

Créez les fonctions f_L1C2() à f_L3C3() et placez dans chacune d'entre-elles le code nécessaires au bon fonctionnement du jeu ☐.

> Ces huit fonctions ne se distinguent de f_L1C1() que par l'ensemble des alignements concernés et par l'adresse du bouton devant être mis à jour.

Création du type EJoueur

Comme nous l'avons vu dans le TD02, il est préférable de créer un fichier .h spécifique pour y définir le type dont nous avons besoin.

Dans le menu "File" de Visual C++, choisissez la commande "New.." ☐.

Dans l'onglet "Files" du dialogue qui s'ouvre alors, sélectionnez l'option "C/C++ Header File" ☐ et tapez "JOUEUR.H" dans la zone d'édition "File Name:" ☐, puis cliquez sur le bouton [OK] ☐.

Dans le fichier vierge qui est alors créé, tapez la définition de notre type ☐:

```
1 #pragma once
2 enum EJoueur {JOUEUR_UN, JOUEUR_DEUX, ERREUR};
```

> Même si vous n'avez pas prévu d'utiliser cette possibilité, il est préférable que les instances d'un type énuméré soient en mesure d'adopter une valeur explicitement illégitime.

> La directive #pragma once qui figure sur la ligne *1* est une commande propre au compilateur Microsoft qui permet d'éviter le problème de la redéfinition des valeurs en cas de #inclusion multiple du fichier définissant un type énuméré.

La classe CAlignement

Nous savons que les variables de type CAlignement doivent être capables de tenir à jour le nombre de marques que chaque joueur a placé dans les cases correspondantes. Ce décompte peut être réalisé à l'aide de deux variables membre de type int, que nous nommerons nbMarquesUn et nbMarquesDeux.

De plus, l'écriture du programme nous a conduit à émettre deux exigences concernant la classe CAlignement :

- La classe CAlignement devra disposer d'une fonction reset().
- La classe CAlignement doit comporter une fonction membre nommée ajouteMarque() et comportant deux paramètres. Le premier est de type EJoueur et il indique quel joueur vient d'occuper une des cases de l'alignement. Le second est l'adresse d'un booléen que la fonction doit rendre vrai si l'ajout de la marque conduit le joueur à la victoire.

Création de la classe

Dans l'onglet "ClassView" de le fenêtre "Workspace", cliquez avec le bouton droit sur la ligne TD05 Classes. Dans le menu qui apparait alors, choisissez la commande "New Class..." ☐.

Dans la zone d'édition "Name:" du dialogue de création de classe, tapez CAlignement, puis cliquez sur le bouton [OK] ☐.

Création des membres

Ajoutez à la classe CAlignement les deux variables membres nbMarquesUn et nbMarquesDeux ☐.

Ajoutez aussi à la classe CAlignement les fonctions membre reset() et ajouteMarque() ☐.

Etant donné que la fonction ajouteMarque() utilise un paramètre de type EJoueur, il faut ajouter une directive #include "joueur.h" dans le fichier CAlignement.h, avant le début de la définition de la classe ☐.

La fonction reset()

Cette fonction se borne à ramener les deux variables membre à zéro. Insérez donc dans son corps les deux instructions nécessaires ☐:

```
1  void CAlignement::reset()
2  {
3  nbMarquesUn = 0;
4  nbMarquesDeux = 0;
5  }
```

La fonction ajouteMarque()

Le rôle de cette fonction est d'ajouter un à l'une ou l'autre des variables membre, en fonction de la valeur qui lui a été transmise pour initialiser son premier paramètre. Si l'une des deux variables membre atteint la valeur 3, c'est que l'alignement au titre duquel la fonction est exécutée vient d'être réalisé par leJoueur. Il faut donc que la fonction signale la fin de la partie en plaçant la valeur vraie dans la variable booléenne dont l'adresse figure dans son second paramètre :

```
1  void CAlignement::ajouteMarque(EJoueur leJoueur, bool *victoire)
2  {
3  (leJoueur == JOUEUR_UN) ? ++nbMarquesUn : ++nbMarquesDeux;
4  if(nbMarquesUn == 3 || nbMarquesDeux == 3) //partie finie ?
5     *victoire = true;
6  }
```

Remarquez bien que, si le test de la ligne 4 échoue, il faut se garder de modifier la valeur pointée par victoire, car un autre alignement se trouve peut-être réalisé.

Insérez le code ci-dessus dans le corps de la fonction ajouteMarque()☐.

Les variables membre de la classe TD05DialogImpl

Pour que le programme soit complet, il reste encore à ajouter à la classe TD05DialogImpl les huit variables membre de type CAlignement promises ☐, ainsi que la variable membre joueurActif ☐.

Le contenu du fichier TD05DialogImpl.h doit finalement être :

```
1  #include "td05dialog.h"
2  #include "JOUEUR.H"     // Added by ClassView
3  #include "Alignement.h" // Added by ClassView
4  class TD05DialogImpl : public TD05Dialog
5  {
6     Q_OBJECT
7  public:
8     TD05DialogImpl(QWidget* parent=0,const char* name=0,bool modal=FALSE,WFlags f=0);
9     EJoueur joueurActif;
10    CAlignement diagonaleUn;
11    CAlignement diagonaleDeux;
12    CAlignement ligneDuHaut;
13    CAlignement ligneDuMilieu;
14    CAlignement ligneDuBas;
15    CAlignement colonneDeGauche;
16    CAlignement colonneDuMilieu;
17    CAlignement colonneDeDroite;
```

```
18     void f_effacer();
19     void f_L1C1();
20     void f_L1C2();
21     void f_L1C3();
22     void f_L2C1();
23     void f_L2C2();
24     void f_L2C3();
25     void f_L3C1();
26     void f_L3C2();
27     void f_L3C3();
28     void codeCommunATousLesCoups(bool victoire, QPushButton * leBouton);
29  };
```

Lorsque vous utilisez la commande "Add member variable" pour ajouter à une classe une variable membre qui est une instance d'un type que vous avez précédemment défini, Visual C++ insère automatiquement dans le fichier .h qui contient la définition de la classe la directive `#include` nécessaire (ligne 2 et 3) pour que ce type soit reconnu lors de la compilation.

Si vous utilisez une autre méthode pour ajouter de telles variables membre (en tapant directement leur déclaration dans le fichier .h, par exemple), il vous faut en revanche veiller vous-même à la #inclusion des fichiers .h définissant les types concernés.

Vous pouvez maintenant compiler ☐ et exécuter ☐ votre programme.

4 - Questions et exercices

1) Pourquoi la fonction `ajouteMarque()` doit-elle utiliser un paramètre de type `EJoueur`, alors que la fonction `codeCommunATousLesCoups()`, par exemple, s'en passe parfaitement ?

2) Si la fonction suivante est définie :

```
1  void TD05DialogImpl::resetLaCase(QPushButton *laCase)
2  {
3  laCase->setText("");
4  laCase->setEnabled(true);
5  }
```

comment pouvez-vous réécrire la fonction `f_effacer()` ?

3) Un programmeur a tenté de définir ainsi la fonction `ajouteMarque()` :

```
1  void CAlignement::ajouteMarque(EJoueur leJoueur, bool *victoire)
2  {
3  int *aModifier =
          (leJoueur == JOUEUR_UN)? &m_scoreJoueurDeux : &m_scoreJoueurUn;
4  aModifier = aModifier + 1;
5  if (aModifier == 3)
6     *victoire = true;
7  }
```

Quelles erreurs a-t-il commises ?

5 - Qu'avons-nous appris ?

1 – Il faut commencer par se faire une bonne représentation de ce que le programme qu'on va écrire va faire et de comment il va fonctionner.

2 – Une fois cette représentation acquise, on peut commencer à la concrétiser. Les détails se précisent souvent d'eux mêmes, à mesure qu'on avance. Il faut simplement commencer par ce qui paraît évident et noter scrupuleusement toutes les suppositions que l'on fait, de façon à pouvoir ensuite s'assurer qu'elles sont toutes finalement justifiées.