Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator
Etape 11 : Boucles et opérateurs logiques

Au cours de cette étape, nous allons poursuivre la mise au point de notre programme de Tic Tac Toe. Il s'agit maintenant de rendre le programme capable de détecter la victoire de l'un des joueurs. Avant de reprendre le projet, nous allons toutefois ajouter encore quelques armes à notre arsenal

1 - Opérateurs logiques

Il arrive fréquemment que l'opportunité d'une action ne dépende pas d'une seule condition, mais de plusieurs. Il est alors possible de procéder à des tests successifs :

```
if(ilPleut)
 if(ilFautSortir)
  prendreParapluie();         //ne sera exécutée que si ilPleut ET qu'ilFautSortir
```

```
if(ilPleut)
  prendreParapluie();
else
  if(ilNeige)
    prendreParapluie();
//prendreParapluie() sera exécutée si ilPleut OU si ilNeige
```

Ces exemples supposent évidemment que `ilPleut`, `ilFautSortir` et `ilNeige` sont des booléens dont les valeurs ont été précédemment calculées.

Le langage C++ offre des opérateurs logiques qui permettent parfois de simplifier l'expression de ce type de conditions :

```
if(ilPleut && ilFautSortir)
  prendreParapluie();         //ne sera exécutée que si ilPleut ET qu'ilFautSortir
```

```
if(ilPleut || ilNeige)
  prendreParapluie();   //sera exécutée si ilPleut OU si ilNeige
```

La seule "difficulté" que présentent ces opérateurs est qu'il ne faut pas oublier qu'ils s'appliquent chacun à deux valeurs booléennes (éventuellement elles-mêmes produites par des opérateurs de comparaison). En conséquence, il faudra écrire :

```
if(note > 10 && note < 14)
  mention = "passable";
```

et non

```
if(note > 10 && < 14)   //CA NE VEUT RIEN DIRE !
  mention = "passable";
```

Le langage propose aussi un opérateur de négation qui permet de traiter élégamment le cas où on ne s'intéresse qu'au `else` d'un `if() else` :

```
if (ilAReussi)
  {//rien à faire...}
else
  recommencer();
```

peut ainsi devenir

```
if (! ilAReussi)
  recommencer();
```

2 - Boucles

Jusqu'à présent, la seule forme de boucle que nous ayons rencontrée est la boucle `do { } while();`. Le langage C++ permet d'exprimer la même idée à l'aide de deux autres constructions.

Toutes les formes de boucles sont formellement équivalentes.

En d'autres termes, il n'existe pas de situation où une forme de boucle permettrait d'écrire le programme souhaité alors que les autres ne le permettraient pas. Le choix d'une forme de boucle est une pure question de confort pour le programmeur.

La boucle `while() { }`

Proche cousine de la boucle `do () while { }`, cette structure de contrôle s'en distingue essentiellement par le fait que le test est effectué avant toute exécution du bloc d'instructions concerné. Ceci est notamment pratique dans les cas où il est possible que ce bloc ne doive pas être exécuté du tout :

```
3 int position(0);
4 while (position < laChaine.length() && laChaine[position] != 'X')
5   ++position;
```

Dans cet exemple, il s'agit de trouver la première occurrence du caractère X dans `laChaine`. Si le premier caractère de `laChaine` est un X, l'instruction `++position` ne doit pas être exécutée. L'utilisation d'un `do () while` imposerait ici l'introduction d'un `if` :

```
int position(0);
if (laChaine[position] != 'X')
  do {
     ++position;
  } while (position < laChaine.length() && laChaine[position] != 'X');
```

La boucle `for( ; ; )`

Cette structure de contrôle est sans doute la plus fréquemment rencontrée dans les programmes réels. Il s'agit en effet d'une notation abrégée d'un cas très fréquent, celui où la boucle est contrôlée par un compteur :

```
1 int doigt(0);
2 do {
    //un traitement quelconque...
3 ++doigt;
4 } while (doigt < MAX);
```

peut ainsi devenir

```
1 int doigt;
1 for(doigt = 0 ; doigt < MAX ; ++doigt) {
    //un traitement quelconque...
2 }
```

En raison de l'extrême popularité de cette construction, une possibilité supplémentaire lui a été accordée : le compteur peut être rendu local au bloc contrôlé par le `for( ; ; )` :

```
1 for(int doigt(0) ; doigt < MAX ; ++doigt) {
    //un traitement quelconque...
2 }
```

3 - Tic tac toe, version sérieuse

Avant de chercher à ajouter une nouvelle fonctionnalité à notre projet, nous allons procéder à de nouvelles simplifications du code, de façon à partir sur des bases réellement saines.

Élimination des fonctions associées aux cases

Plutôt que de connecter chaque bouton à une fonction différente, nous allons regrouper tous les boutons dans un widget spécial, un `QButtonGroup`, qui dispose d'un signal nommé `buttonClicked()`. Ce signal est évidemment émis lorsque l'un des boutons du groupe est cliqué, mais sa particularité remarquable est qu'il transmet un numéro d'identification qui va permettre au slot qui lui est connecté de savoir *quel* bouton a déclenché l'appel.

Avec l'éditeur graphique, supprimez la connexion entre le bouton `b0` et le slot `f(0)` ☐.

Dans notreClasse.h, ajoutez une variable membre de type `QButtonGroup` nommée `m_leGroupe` ☐, supprimez la déclaration des fonctions `f0()`,... `f8()` ☐ et déplacez celle de la fonction `joue()` pour en faire un slot ☐.

L'utilisation du type `QButtonGroup` exige une directive `#include <qbuttongroup>`

Dans le fichier notreClasse.cpp, supprimez purement et simplement les fonctions `f0()`, ..., `f8()` ☐.

Donnez au constructeur de `notreClasse` l'aspect suivant :

```
1  notreClasse::notreClasse(QWidget *parent) : QDialog(parent), ui(new Ui::notreClasse)
2  {
3      ui->setupUi(this);
       //stockage des adresses des "boutons cases" dans une QMap
4      int n(0) ;
5      m_lesBoutons[n++] = ui->b_0;
6      m_lesBoutons[n++] = ui->b_1;
7      m_lesBoutons[n++] = ui->b_2;
8      m_lesBoutons[n++] = ui->b_3;
9      m_lesBoutons[n++] = ui->b_4;
10     m_lesBoutons[n++] = ui->b_5;
11     m_lesBoutons[n++] = ui->b_6;
12     m_lesBoutons[n++] = ui->b_7;
13     m_lesBoutons[n++] = ui->b_8;
       //intégration des boutons au buttonGroup
14     for(int n(0); n < 9 ; ++n)
15        m_leGroupe.addButton(m_lesBoutons[n], n);
       //connexion directe du groupe à la fonction joue()
16     connect(&m_leGroupe, SIGNAL(buttonClicked(int)), this, SLOT(joue(int)));
       //début de partie
17     f_recommencer(); //ne pas appeler f_recommencer() avant de remplir la collection !
18 }
```

Les lignes *14* et *15* constituent une boucle qui ajoute les 9 boutons au groupe. Remarquez d'une part l'utilisation de la variable `n` et de la `QMap m_lesBoutons` qui rendent possible cette boucle et, d'autre part, le fait que l'ajout d'un bouton au groupe s'accompagne de la spécification du numéro d'identification qui va lui être associé et qui est identique à l'index de son adresse dans la `QMap`.

La ligne *16* établit une unique connexion qui lie directement notre fonction `joue()` au signal qui l'intéresse. La fonction `connect()` attend qu'on lui désigne le widget émetteur en lui en passant l'adresse. Celle-ci est obtenue en appliquant l'opérateur "adresse de" au widget en question.

Si `machin` est une variable, l'expression `& machin` a pour valeur l'adresse de cette variable.

Vérifiez que votre programme peut être compilé et que son exécution produit toujours l'effet attendu □.

Détection de la fin de la partie

Il s'agit maintenant de rendre le programme capable de détecter la fin de la partie, c'est à dire la réalisation d'un alignement de trois marques identiques sur une ligne, une colonne ou une diagonale.

Représenter l'état de la partie

Dès qu'un programme effectue des traitements non triviaux, il faut le doter d'une **représentation** des données distincte de la **présentation** de celle-ci qui est faite à l'utilisateur.

Dans le cas présent, la détection de la victoire d'un des joueurs n'est pas quelque chose d'absolument évident. Nous ne pouvons donc nous contenter d'une représentation de l'état de la partie qui se résume à l'état des boutons présents à l'écran.

> Toutes les caractéristiques de la représentation visuelle de la situation ne sont pas pertinentes pour les traitements que nous envisageons. Notre représentation ne va donc s'attacher à capturer que certains aspects de la "réalité" présentée à l'utilisateur. La taille des boutons ou la nature de l'indice visuel qui distingue les cases occupées par chacun des joueurs, par exemple, n'ont aucune pertinence pour détecter la fin de la partie. C'est parce que cette représentation est **partielle** et **orientée** vers une utilisation particulière qu'on l'appelle communément un **modèle** de la partie en cours.

La structure de données la plus simple[1] qu'on puisse imaginer pour représenter l'état d'une partie de Tic tac toe est sans doute une `QMap` indexée par des `int` (les numéros des cases, de 0 à 8). Si plusieurs façons de représenter l'état d'une case peuvent être envisagées, il peut sembler naturel, au moins dans un premier temps, de stocker simplement le texte affiché par le bouton.

> Cette représentation peut sembler parfaitement redondante, puisqu'elle stocke exactement la même information que les boutons et offre une organisation (l'indexation par des numéros) qui est déjà

1 Ce n'est pas nécessairement le meilleur choix possible, mais il peut suffire pour les besoins de ce TP.

disponible pour les boutons (grâce à la `QMap m_lesBoutons`). L'évolution du projet (dans le cas présent, faire tenir le rôle d'un des joueurs par l'ordinateur, par exemple) conduira toujours à une dissociation entre **présentation** (à l'écran, pour les besoins de l'utilisateur) et **représentation** (en mémoire, pour les besoins du calcul). Il est donc préférable d'établir **systématiquement** cette distinction, et de la mettre en place aussi tôt que possible.

Ajoutez à `notreClasse` une variable membre de type `QMap<int, QString>` nommée `m_odele` ☐.

Actualiser la représentation

Pour être utilisable, notre modèle doit évidemment être tenu à jour au fur et à mesure de l'évolution de la partie !

Tous les instructions qui font apparaître ou disparaître une marque dans une case doivent donc s'accompagner d'une instruction effectuant une modification analogue sur `m_modele`. La fonction `joue()` doit par conséquent comporter une instruction

```
m_odele[cle] = m_laMarque;
```

De même, la boucle de la fonction `f_recommencer()` doit inclure

```
m_odele[doigt] = "";
```

Effectuez ces deux ajouts dans votre fichier notreClasse.cpp ☐.

Détecter les alignements verticaux et horizontaux

La fonction `joue()` est idéalement placée pour assurer cette détection : elle est exécutée chaque fois qu'une nouvelle case est occupée, c'est à dire chaque fois que la partie est susceptible de s'achever. La détection de la fin de la partie présente toutefois deux caractéristiques qui font qu'il n'est pas souhaitable que le code correspondant figure dans la fonction `joue()` :
- c'est une tâche annexe, et non la raison d'être de la fonction `joue()`
- les calculs nécessaires sont relativement complexes, au point que leur présence dans `joue()` noierait complètement le code qui constitue l'essence de cette fonction, ce qui rendrait difficile la lecture du programme.

Il est donc préférable de déporter cette détection dans une autre fonction, que `joue()` peut se contenter d'appeler :

```
1  void notreClasse::joue(int cle)
2  {
3      m_lesBoutons[cle]->setText(m_laMarque);
4      m_lesBoutons[cle]->setEnabled(false);
5      m_odele[cle] = m_laMarque;
6      if(m_laMarque == "X")
7          m_laMarque = "O";
8      else
9          m_laMarque = "X";
10   if(coupGagnant(cle))
11     QMessageBox::information(this, "Tic tac toe", "C'est fini !");
12 }
```

Introduisez ce test dans votre fonction `joue()` ☐.

La ligne *11* utilise la classe `QMessageBox` pour afficher une fenêtre portant le titre "Tic tac toe" et affichant simplement le message "C'est fini !".

La fonction `coupGagnant()` va donc recevoir le numéro du bouton sur lequel une marque vient d'être ajoutée, et elle devra, comme son nom le suggère, renvoyer un booléen indiquant s'il s'agit d'un coup mettant fin à la partie.

Dans ce jeu, il s'agit de réaliser des alignements de marques identiques. Les règles du jeu imposent de prendre en compte trois directions (horizontale, verticale, diagonale) et s'expriment donc dans un espace à deux dimensions (un plan).

Le problème est que notre modèle de la partie est, lui, unidimensionnel. En effet, la numérotation des cases de 0 à 8 ne capture pas du tout les relations de contiguïté créées par leur disposition en trois lignes et trois colonnes.

Pourquoi n'adoptons-nous pas une représentation de la partie qui serait elle-même bi-dimensionnelle ?

> Cela rendrait certainement la tâche de la fonction `coupGagnant()` bien plus facile (le modèle serait mieux adapté au traitements que nous avons à effectuer) , mais c'est malheureusement impossible : la mémoire d'un ordinateur ne propose qu'un espace d'adressage unidimensionnel. En effet, les cases mémoires sont elle-même numérotées à partir de 0 et n'entretiennent entre-elles aucune relation particulière autre que l'ordre des adresses. En d'autre termes, si on peut considérer que chaque case mémoire à deux voisines (une "précédente" et une "suivante"), il n'est pas possible de dire qu'elle en a huit (comme c'est le cas de la case centrale du Tic tac toe, par exemple).

Les relations de contiguïté qui nous intéressent n'étant pas présentes dans la *structure* de données qui nous sert de modèle (la `QMap m_odele`), il faut les faire apparaître dans le *traitement* qui est appliqué à ces données.

A partir du numéro d'identification d'une case, il est facile de déterminer sur quelle colonne et sur quelle ligne elle se situe :

	colonne 0	colonne 1	colonne 2
ligne 0	0	1	2
ligne 1	3	4	5
ligne 2	6	7	8

Du fait que nous numérotons systématiquement à partir de 0, `ligne = numero / 3` et `colonne = numero % 3`

> Rappels : en C++, le signe `%` désigne l'opérateur modulo, alias "reste de la division entière", et, lorsqu'il est appliqué à des entiers, l'opérateur `/` effectue une DIVISION ENTIERE.

Le numéro de la première case d'une colonne est identique à celui de la colonne. Le numéro de la seconde case de cette colonne peut être obtenu en ajoutant 3 à celui de la première case. Le numéro de la troisième peut être obtenu en ajoutant 3 à celui de la seconde.

Le numéro de la première case d'une ligne est le triple du numéro de la ligne. Le numéro de la seconde case de cette ligne peut être obtenu en ajoutant 1 à celui de la première case. Le numéro de la troisième peut être obtenu en ajoutant 1 à celui de la seconde.

Ces quelques points étant établis, il devient facile de détecter les alignements verticaux et horizontaux : sachant quelle case vient de recevoir une marque, on détermine quelle est la colonne (resp. la ligne) concernée et on vérifie si cette colonne (resp. cette ligne) contient la même marque sur toute les lignes (resp. dans toutes les colonnes).

```
1  bool notreClasse::coupGagnant(int numBouton)
2  {
3     int colonneConcernee = numBouton % 3;
4     int N1 = colonneConcernee;    //le numéro de la première case de cette colonne
5     int N2 = N1 + 3;              //le numéro de la deuxième case de cette colonne
6     int N3 = N2 + 3;              //le numéro de la troisième case de cette colonne
7     if(m_odele[N1] == m_odele[N2] && m_odele[N2] == m_odele[N3])
8        return true; //colonne complète
9     int ligneConcernee = numBouton / 3;
10    N1 = ligneConcernee * 3;  //le numéro de la première case de cette ligne
11    N2 = N1 + 1;              //le numéro de la deuxième case de cette ligne
12    N3 = N2 + 1;              //le numéro de la troisième case de cette ligne
13    if(m_odele[N1] == m_odele[N2] && m_odele[N2] == m_odele[N3])
14       return true; //ligne complète
15   return false; //on n'a détecté aucun alignement
16 }
```

Ajouter cette fonction à votre projet et vérifiez que le programme détecte maintenant les victoires par alignement vertical ou horizontal ☐.

Exercice : détecter les alignements diagonaux

Il s'agit donc de compléter le code de la fonction `coupGagnant()` et de vérifier que le programme détecte enfin tous les cas de victoire ☐.

Indices :

1 - Aucun alignement diagonal ne peut être réalisé si la case centrale ne contient pas la même marque que la case sur laquelle une marque vient d'être ajoutée.

2 - Si on sait que la case centrale contient la même marque que la case sur laquelle une marque vient d'être ajoutée, on peut détecter un alignement diagonal sans même savoir sur quelle case une marque vient d'être ajoutée.

4 - Conclusion

Un programme n'est généralement intéressant que s'il effectue des traitements sur les données.

Beaucoup de programmes offrent également une présentation (visuelle, par exemple) d'un certain aspect de certaines données.

Il est rarement payant à moyen ou long terme d'essayer d'effectuer les traitements sans mettre en place une représentation des données indépendante de leur présentation à l'utilisateur.

Dans l'exemple du Tic tac toe, c'est la détection des alignements (ie. un traitement non trivial) qui bénéficie de la disponibilité de la variable m_odele.

En l'absence de cette variable, la fonction aurait été condamnée à interroger l'état des widgets pour connaître l'état de la partie. Toute modification du choix des widgets visant à améliorer l'esthétique de l'interface (utilisation d'images à la place du texte affiché par les boutons, par exemple) remettrait alors en cause les fonctions de traitement (`coupGagnant()`, en l'occurrence). Plus le programme se complexifie, plus ces dépendances entre présentation et traitements deviennent ingérables.

Dans bien des cas, les traitements à effectuer exigent de toute façon la prise en compte de données qui ne sont pas présentées à l'utilisateur. Il devient alors encore plus évident qu'il convient de représenter les données et les traitements qui peuvent leur être appliqués d'une façon qui doit être aussi indépendante que possible de l'interface utilisateur.