Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator
Etape 10 : Collectionner les pointeurs

1 - Ne pas appeler les choses par leur nom : introduction aux pointeurs....2
2 - Tic tac toe....3
Création du projet....3
Dessin du dialogue....3
Le constructeur de notreClasse....3
La fonction f_recommencer()....4
Les fonctions f0() à f8()....4
La fonction joue()....5
Le fichier notreClasse.h....5
3 - Notions de déréférencement et de pointeur invalide....6
4 - Illustration....6

L'usage d'une interface utilisateur réalisée en positionnant des widgets à l'aide de QtDesigner conduit souvent à écrire des fonctions qui ont besoin d'accéder aux widgets en question (pour en observer l'état, ou pour en changer le contenu ou l'aspect, par exemple).

Cet accès aux widgets se fait de façon indirecte, en utilisant des adresses plutôt que des noms de variables qui désigneraient directement les objets. L'utilisation d'adresses présente un intérêt particulier, car il s'agit de valeurs qui peuvent être stockées dans une collection, ce qui ouvre la possibilité d'utiliser une boucle pour traiter un grand nombre de widgets.

En revanche, la manipulation d'adresses exige la maîtrise d'une notion nouvelle, celle de pointeur. Pour nous familiariser avec ce nouveau genre de variables, nous allons mettre au point un petit programme qui permet de jouer au Tic Tac Toe[1].

1 - Ne pas appeler les choses par leur nom : introduction aux pointeurs

Pourquoi diable aurions-nous envie de ne pas appeler les choses par leur nom ?

Le problème d'un nom, c'est que ce n'est pas une valeur manipulable par le programme. En d'autres termes, un programme ne peut pas stocker un nom quelque part (pour se souvenir de quelle variable il a utilisé, par exemple) ou modifier un nom (pour utiliser une variable différente).

Si une instruction placée dans une boucle désigne une variable par son nom, ce sera forcément la même variable qui sera utilisée lors de toutes les exécutions de l'instruction. Il est ainsi facile de faire beaucoup d'opérations sur une variable (le "beaucoup" étant pris en charge par la boucle), mais il est impossible d'utiliser une telle boucle pour, par exemple, remettre à 0 des milliers de variables.

Nous savons que les variables sont représentées par l'état électrique d'une zone de mémoire et que, de ce fait, elles ont une adresse (le numéro de la première case de mémoire de la zone qui leur est attribuée). Le langage C++ permet de désigner les variables par leur adresse.

Une variable peut être désignée soit par son **nom**, soit par une **référence** qui lui a été attribuée, soit par son **adresse**.

A la différence des noms (ou des références), les adresses sont des valeurs manipulables par le programme, puisque ce sont de simples nombres entiers. Il faut toutefois tenir compte d'un détail important : bien qu'étant un nombre entier, une adresse n'a d'intérêt que dans la mesure où elle permet d'accéder à la zone de mémoire correspondante pour récupérer ou modifier la valeur qui est stockée à cet endroit.

Or, si la connaissance de son adresse permet d'accéder à une zone de mémoire, elle ne suffit en revanche pas pour obtenir la **valeur** représentée par l'état électrique de cette zone. En effet, cette valeur ne peut être obtenue qu'en appliquant à cet état électrique une **interprétation** dont les règles dépendent du **type** de la valeur dont il s'agit.

Si vous disposiez d'un appareil vous permettant de voir l'état électrique de la mémoire (sous forme de 0 et de 1) et qu'on vous demande quelle est la valeur stockée à l'adresse 3212, par exemple, vous ne pourriez pas répondre. Vous pourriez regarder au bon endroit et y voir des 0 et des 1, mais vous n'auriez aucune idée d'où s'arrête la représentation de la valeur qu'on vous demande et ignoreriez tout des règles de lecture à utiliser (s'agit-il d'un `int` ? d'un `double` ? d'un simple booléen ? d'une chaîne de caractères ?).

Pour contourner ce problème, le langage C++ ne stocke pas les adresses dans des variables de type `int`, mais introduit des types spécialement destinés à cet usage, les pointeurs. Il existe autant de types de pointeurs qu'il existe de types :

Si une variable est de type **`TRUC`**, son adresse peut être stockée dans une variable de type **`TRUC *`**

Le type `TRUC *` se lit "pointeur sur TRUC" plutôt que "TRUC étoile".

Comme avec n'importe quel autre type, on peut créer des variables de type "pointeur sur...", mais il faut toujours préciser le type de l'objet pointé :

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.

```
1 char * pointeurUn ;  //la variable pointeurUn doit contenir l'adresse d'un char
2 bool * ptrDeux ;     //la variable ptrDeux doit contenir l'adresse d'un bool
```

Les pointeurs sont un outil fondamental en C++ mais, pour l'instant, une seule information supplémentaire nous est nécessaire :

Lorsqu'une instance n'est pas désignée par son nom mais par celui d'un pointeur qui contient son adresse, on utilise une **->** plutôt qu'un **.** comme opérateur de sélection de membre.

> L'éditeur de code intégré à QtCreator est le plus souvent capable de substituer une flèche à un point lorsque cela s'impose, ce qui permet au programmeur de ne pas se focaliser excessivement sur la question "Le nom que j'utilise est-il celui de l'objet qui m'intéresse, ou celui d'un pointeur qui contient l'adresse de cet objet ?"

2 - Tic tac toe

Pour bien comprendre la situation, commençons par réaliser une version du programme qui ne fait appel qu'à des techniques simples et, pour la plupart, déjà connues.

Création du projet

La procédure à suivre est celle décrite lors de l'étape 8 : Dans le menu <Fichier>, choisissez la commande <Nouveau fichier ou projet...> ☐. Dans la liste des types de projets proposés, choisissez <Projet QtWidget> <Application graphique Qt> ☐. Choisissez ensuite un nom et un emplacement pour votre projet ☐.

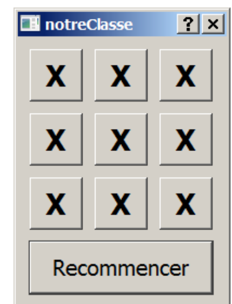
Dans le champ "Nom de la classe :", tapez `notreClasse` ☐.

Dans la liste "Classe parent :", choisissiez `QDialog` ☐.

Dessin du dialogue

L'interface de notre programme comporte dix pushButtons :

- 9 boutons carrés, nommés `b_0`, `b_1`, `b_2`, ...`b_8`, qui vont figurer les cases du jeu ;
- 1 bouton rectangulaire, nommé `b_recommencer`, qui permet d'effacer le contenu des cases pour faire une nouvelle partie.



Disposez ces boutons à l'aide de l'éditeur graphique et donnez à chacun d'eux l'aspect et le nom qui convient ☐.

Connectez le signal `clicked()` du bouton `b_recommencer` à un slot nommé `f_recommencer()` ☐.

La connexion des autres boutons sera réalisée, le moment venu, de façon moins fastidieuse. Vous pouvez donc sauvegarder le fichier notreClasse.ui et passer à l'édition du fichier notreClasse.cpp ☐.

Le constructeur de `notreClasse`

Les boutons qui représentent des cases du jeux vont devoir être modifiés en cours de partie (apparition des marques). Pour simplifier ces opérations, nous allons utiliser une collection de pointeurs sur `QPushButtons` contenant les adresses de ces widgets. Comme plusieurs fonctions devront utiliser cette collection, celle-ci doit être une variable membre de `notreClasse`, et nous l'appellerons `m_lesBoutons`.

Le constructeur (dont nous savons qu'il sera automatiquement exécuté lors de l'instanciation de `notreClasse` par `main()`) doit donc procéder à trois opérations :

- Connexion des boutons `b_0` à `b_8` à des slots que nous appellerons `f0()`, `f1()`, ... `f(8)` :

```
1 notreClasse::notreClasse(QWidget *parent) : QDialog(parent), ui(new Ui::notreClasse)
2 {
3 ui->setupUi(this);
    //connexion des "boutons cases" à des slots
4   connect(ui->b_0, SIGNAL(clicked()), this, SLOT(f0()) );
5   connect(ui->b_1, SIGNAL(clicked()), this, SLOT(f1()) );
6   connect(ui->b_2, SIGNAL(clicked()), this, SLOT(f2()) );
7   connect(ui->b_3, SIGNAL(clicked()), this, SLOT(f3()) );
8   connect(ui->b_4, SIGNAL(clicked()), this, SLOT(f4()) );
9   connect(ui->b_5, SIGNAL(clicked()), this, SLOT(f5()) );
```

```
10    connect(ui->b_6, SIGNAL(clicked()), this, SLOT(f6()) );
11    connect(ui->b_7, SIGNAL(clicked()), this, SLOT(f7()) );
12    connect(ui->b_8, SIGNAL(clicked()), this, SLOT(f8()) );
```

Plutôt que de procéder à ces connexions avec l'éditeur graphique, nous utilisons ici la fonction connect() qui permet d'atteindre le même résultat en quelques lignes de code. La logique reste la même : il convient de préciser le widget concerné, l'événement qui nous intéresse, le widget qui doit réagir et la fonction qui doit être exécutée pour obtenir cette réaction.

Le mécanisme signaux/slots n'appartient pas au langage C++ mais est propre à la librairie Qt. Sa mise en place exige quelques notations spécifiques, qui se traduisent ici par les pseudo-fonctions `SIGNAL()` et `SLOT()` qui permettent de conférer ces statuts aux objets concernés. La désignation du widget qui doit réagir, qui aurait été obtenue en pointant sur le fond de la fenêtre si nous avions utilisé l'éditeur graphique, utilise ici un mot du langage C++ sur lequel nous reviendrons lors d'une prochaine étape, `this`.

- Création des éléments de la collection qui vont permettre d'accéder aux boutons :

```
      //stockage des adresses des "boutons cases" dans une QMap
13    int n(0) ;
14    m_lesBoutons[n++] = ui->b_0;
15    m_lesBoutons[n++] = ui->b_1;
16    m_lesBoutons[n++] = ui->b_2;
17    m_lesBoutons[n++] = ui->b_3;
18    m_lesBoutons[n++] = ui->b_4;
19    m_lesBoutons[n++] = ui->b_5;
20    m_lesBoutons[n++] = ui->b_6;
21    m_lesBoutons[n++] = ui->b_7;
22    m_lesBoutons[n++] = ui->b_8;
```

Remarquez l'usage de la variable `n`, dont les incrémentations successives offrent une protection minimale contre les accidents de copier/coller (il est important que les clés soient des entiers consécutifs).

- Préparation d'une nouvelle partie :

```
      //début de partie
23    f_recommencer(); //ne pas appeler f_recommencer() avant de remplir la collection !
24  }
```

Les opérations de préparation d'une nouvelle partie devant être réitérées entre chaque partie, elles sont logiquement placées dans la fonction `f_recommencer()`, qui est liée au bouton du même nom. Il suffit donc ici d'appeler cette fonction.

La fonction `f_recommencer()`

Pour débuter une nouvelle partie, il faut effacer toutes les marques présentes *(5)*, ré-activer *(6)* tous les "boutons cases" (en cours de partie, les boutons utilisés seront désactivés pour interdire que la marque qui y a été placée soit effacée) et choisir la marque qui apparaîtra dans la première case cliquée *(9)*.

La gestion de l'alternance des marques (X et O) exige le recours à une variable membre permettant au programme de "savoir où il en est". Cette variable sera baptisée `m_laMarque`.

```
1   void notreClasse::f_recommencer()
2   {
3     int doigt(0);
4     do {
5        m_lesBoutons[doigt]->setText("");
6        m_lesBoutons[doigt]->setEnabled(true);
7        ++doigt;
8     } while (doigt < m_lesBoutons.count());
9     m_laMarque = "X";
10  }
```

Le parcours de la `QMap` effectué ici est tout à fait analogue à celui que nous aurions pu faire d'une `QString` pour examiner ses caractères, par exemple. La seule différence notable est

que la fonction permettant de connaître le nombre d'éléments s'appelle `count()` dans le cas d'une `QMap`, alors qu'elle s'appelle `length()` dans le cas d'une `QString`...

Les fonctions `f0()` à `f8()`

Ces fonctions doivent faire apparaître une marque sur le bouton auquel elles sont associées, puis le désactiver de façon à le rendre indisponible jusqu'à la fin de la partie. Elles doivent en outre assurer l'alternance des valeurs "X" et "O" dans la variable `m_laMarque`.

Étant donné que ces neuf fonctions font exactement la même chose (seul le bouton concerné diffère), il est préférable qu'elles ne contiennent qu'une seule instruction : un appel à une même fonction, que nous nommerons `joue()` et à laquelle il faut évidemment préciser le bouton concerné (en lui passant la clé permettant d'accéder à l'adresse de ce bouton dans la `QMap`) :

```
void notreClasse::f0() { joue(0); } //tout sur la même ligne = économie de papier...
void notreClasse::f1() { joue(1); }
void notreClasse::f2() { joue(2); }
void notreClasse::f3() { joue(3); }
void notreClasse::f4() { joue(4); }
void notreClasse::f5() { joue(5); }
void notreClasse::f6() { joue(6); }
void notreClasse::f7() { joue(7); }
void notreClasse::f8() { joue(8); }
```

La fonction `joue()`

La seule particularité remarquable de cette fonction est que, grâce à son paramètre et à l'utilisation de la collection, elle est capable d'agir sur n'importe lequel des neuf "boutons case" :

```
1  void notreClasse::joue(int cle)
2  {
   //affichage de la marque et désactivation du bouton
3  m_lesBoutons[cle]->setText(m_laMarque);
4  m_lesBoutons[cle]->setEnabled(false);
   //alternance des marques
5  if(m_laMarque == "X")
6     m_laMarque = "O";
7  else
8     m_laMarque = "X";
9  }
```

Le fichier notreClasse.h

Une fois complété avec les déclarations des fonctions décrites ci-dessus et des deux variables membre dont nous avons découvert au passage la nécessité, la définition de `notreClasse` a l'aspect suivant :

```
1  class notreClasse : public QDialog
2  {
3      Q_OBJECT
   //déclaration des fonctions membre
4  public slots:
5      void f0();
6      void f1();
7      void f2();
8      void f3();
9      void f4();
10     void f5();
11     void f6();
12     void f7();
13     void f8();
14     void f_recommencer();
15 public:
16     explicit notreClasse(QWidget *parent = 0);
17     ~notreClasse();
18     void joue(int cle);
```

```
//déclaration des variables membre
19 private:
20     Ui::notreClasse *ui;
21     QString m_laMarque;
22     QMap <int, QPushButton*> m_lesBoutons;
23 };
```

Définissez `notreClasse` et ses fonctions membre conformément aux descriptions proposées et vérifiez que votre programme fonctionne ☐.

3 - Notions de déréférencement et de pointeur invalide

Une variable de type "pointeur sur..." n'est, à vrai dire, qu'une variable entière un peu particulière.

A ce titre, elle est capable de prendre n'importe quelle valeur entière.

Mais, dans le cas des pointeurs, un problème se pose : le programme est peut manipuler non seulement la valeur du pointeur, mais aussi *celle de l'objet se trouvant à l'adresse* contenue dans le pointeur.

La fonction `joue()` contient, par exemple, l'instruction

```
m_lesBoutons[cle]->setText(m_laMarque);
```

Il est évident que ce n'est pas l'adresse contenue dans la `QMap` qui doit afficher `laMarque`, mais bien le `QPushButton` désigné par cette adresse.

On appelle "**déréférencement d'un pointeur**" l'action consistant à accéder non pas au pointeur lui-même mais à l'objet dont il contient l'adresse.

Le déréférencement n'a évidemment aucun effet sur le pointeur lui-même. Les programmes "déréférencent les pointeurs" comme les automobilistes "suivent les panneaux".

Le déréférencement est une action dangereuse : le langage et les compilateurs manquent de moyens pour garantir que, au moment où il sera effectué, l'adresse contenue dans le pointeur sera effectivement celle d'un objet du type implicitement promis par le type du pointeur.

On appelle **pointeur invalide** un pointeur qui ne contient pas l'adresse d'un objet du type sur lequel il est sensé pointer.

Déréférencer un pointeur invalide conduit généralement à une grave erreur d'exécution et à l'interruption brutale du programme en cours.

4 - Illustration

Dans la fonction `joue()`, modifiez ainsi la ligne *3* :

```
m_lesBoutons[cle + 1]->setText(m_laMarque);
```

Lancez le programme ainsi modifié et cliquez successivement sur les boutons `b_0` à `b_8`, dans l'ordre ☐.

Pour les boutons `b_0` à `b_7`, le comportement du programme est facile à comprendre : au lieu d'apparaître sur le bouton cliqué, la marque ajoutée apparaît sur le bouton suivant.

Mais que ce passe-t-il lorsqu'on clique sur le bouton `b_8` ?

La fonction `joue()` est appelée et reçoit la valeur `8` pour initialiser son paramètre `cle`. C'est donc

```
m_lesBoutons[8 + 1]->setText(m_laMarque);
```

qui est exécutée, mais notre collection ne contient aucun élément dont la clé serait `9`.

Comme nous le savons, les `QMap` réagissent à une telle tentative d'accès en créant l'élément manquant. Toutefois, rien ne permet ici d'attribuer au nouvel élément une valeur qui serait l'adresse d'un des boutons présents dans l'interface. La "valeur" du nouvelle élément est donc imprévisible (tout comme la "valeur" d'un `int` non initialisé, par exemple).

C'est cette "adresse" imprévisible qui est utilisée par le programme pour exécuter la fonction `setText()` au titre de l'instance de `QPushButton` qui est sensée y habiter. Comme il y a vraiment très peu de chances pour que cette adresse se trouve précisément être celle d'un des boutons, la catastrophe est quasi certaine...