Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator
Etape 9 : Collectionner des contacts

La réalisation de projets intéressants nécessite généralement le recours à des outils permettant la gestion d'un nombre important de données. Deux problèmes se posent : le stockage (il n'est pas réaliste d'envisager de créer des milliers de variables) et la description des traitements devant être effectués (il faut pouvoir mettre en place des boucles qui ne se contentent pas de modifier la valeur d'une variable, mais opèrent sur des objets différents lors de chaque passage).

Deux techniques différentes (bien que proches dans leur principe) permettent d'obtenir ce genre d'effets : les collections et les pointeurs. Un premier projet, très simple, va nous fournir un exemple d'utilisation d'une collection. L'utilisation de pointeurs sera, pour sa part abordée à l'étape suivante.

1 - Ébauche d'un répertoire téléphonique

La gestion d'un répertoire téléphonique élémentaire pose clairement le type de problème évoqué ci-dessus : le nombre de contacts est potentiellement important et, en tous cas, il est inconnu du programmeur, qui ne peut donc prévoir une variable pour chacun d'entre eux.

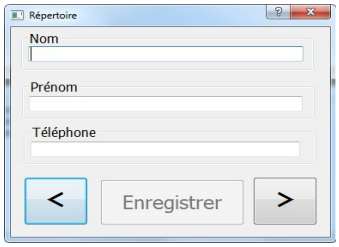
Création du projet

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 `monDialogue` ☐.

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

Dessinez une interface graphique ayant l'aspect général suggéré ci-contre.



Donnez au trois LineEdit les noms `nom`, `prenom` et `telephone` ☐.

> Attention à bien renommer les LineEdit, et non les GroupBox qui entourent chacun d'entre eux et n'ont qu'un rôle cosmétique...

Liez les trois boutons à trois slots nommés `f_reculer()`, `f_enregistrer()` et `f_avancer` ☐.

> Les boutons [<] et [>] seront utilisés pour naviguer dans le répertoire.

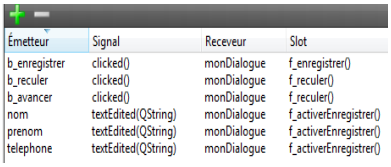
Décochez la propriété `enabled` du bouton [Enregistrer] ☐.

Ce bouton ne sera rendu disponible que si le contenu d'au moins un des LineEdit a été modifié par l'utilisateur, dans le but de décrire un nouveau contact. Il faut donc qu'une fonction de notre programme soit exécutée lorsqu'un tel événement survient.

Connectez l'événement `textEdited()` de chacun des LineEdit à un slot nommé `f_activerEnregistrer()` ☐.

> Il est inutile de créer trois slots : les trois évènements peuvent déclencher l'appel d'une même fonction, puisque la tâche à effectuer est la même (l'activation du bouton).

Le bilan de vos connections devrait être celui représenté ci-contre.



Émetteur	Signal	Receveur	Slot
b_enregistrer	clicked()	monDialogue	f_enregistrer()
b_reculer	clicked()	monDialogue	f_reculer()
b_avancer	clicked()	monDialogue	f_reculer()
nom	textEdited(QString)	monDialogue	f_activerEnregistrer()
prenom	textEdited(QString)	monDialogue	f_activerEnregistrer()
telephone	textEdited(QString)	monDialogue	f_activerEnregistrer()

> En cas de besoin, reportez-vous à la description des procédures proposée lors de l'étape 8.

Enregistrez le dessin de l'interface et fermez-le ☐.

Création des fonctions

Les fonctions liées aux boutons doivent être déclarées dans le fichier monDialogue.h :

```
1  class monDialogue : public QDialog
2  {    Q_OBJECT
   //déclaration des fonctions membre
3  public slots:
4      void f_reculer();
5      void f_enregistrer();
6      void f_avancer();
7      void f_activerEnregister();
8  public:
9      explicit monDialogue(QWidget *parent = 0);
10     ~monDialogue();
```

```
    //déclaration des variables membres
11  private:
12      Ui::monDialogue *ui;
13  };
```

Par ailleurs, les définitions de ces fonctions doivent être ajoutées dans le fichier monDialogue.cpp :

```
    //*************************************************************
14  void monDialogue::f_reculer()
15  {
16  }
    //*************************************************************
17  void monDialogue::f_enregistrer()
18  {
19  }
    //*************************************************************
20  void monDialogue::f_avancer()
21  {
22  }
    //*************************************************************
23  void monDialogue::f_activerEnregistrer()
24  {
25  ui->b_Enregistrer->setEnabled(true); //active le bouton [Enregistrer]
26  }
```

Vérifiez que votre programme ainsi complété peut être compilé et exécuté □.

Création de la classe `CContact`

Pour stocker les informations relatives à un contact, il est pratique de disposer d'un type de variable spécialement adapté. Nous allons donc créer une classe `CContact` comportant trois variables membre destinées à accueillir les trois informations que le programme gère : les nom, prénom et numéro de téléphone de la personne concernée.

Les fonctionnalités offertes par le programme sont très limitées : il s'agit essentiellement de stocker des contacts et d'afficher les informations concernant l'un d'entre eux. La classe `CContact` peut donc être très rudimentaire, et elle ne va, en particulier, comporter aucune fonction membre.

> Dans les ouvrages techniques, on qualifie souvent de POD (**p**lain **o**ld **d**ata) les classes qui n'offrent que des capacités de stockage, sans fournir aucun moyen particulier de traiter les données (à l'aide de fonctions membre).

Ajoutez à votre projet un fichier d'en-tête nommé contact.h □ et donnez-lui le contenu suivant □ :

```
1   #ifndef CONTACT_H
2   #define CONTACT_H
3   class CContact
4   {
5   public:
6       QString m_nom;
7       QString m_prenom;
8       QString m_telephone;
9   };
10  #endif // CONTACT_H
```

> Cette classe étant dépourvue de fonction membre, il n'y aura pas de fichier contact.cpp

Création d'un contact

Cette classe permet de créer un contact "vide" qui pourra être utilisé pour saisir les informations concernant un nouveau contact devant être ajouté au répertoire. Comme ce pseudo-contact doit figurer dans le répertoire dès le lancement du programme, nous pouvons le créer dans le constructeur.

> Nous savons en effet que le constructeur (qui a été ajouté automatiquement au programme lors de la création du projet) est exécuté lors de l'instanciation de la classe `monDialogue` (opération effectuée par `main()`, dès le lancement du programme).

```
monDialogue::monDialogue(QWidget *parent) : QDialog(parent), ui(new Ui::monDialogue)
{
    ui->setupUi(this);
    CContact bidon; //création du pseudo-contact ("vide")
    bidon.m_nom = "Saisir ici un nom";
    bidon.m_prenom = "Saisir ici un prénom";
    bidon.m_telephone = "Saisir ici un numéro de téléphone";
}
```

Complétez le constructeur comme suggéré ci-dessus □.

Si elle est facile à créer et à manipuler, la variable `bidon` pose au moins deux problèmes du point de vue de notre projet :
- sa durée de vie est limitée au temps d'exécution du constructeur (puisqu'elle en est une variable locale), ce qui est contraire à l'idée même du programme, qui est de stocker les informations relatives au contacts (au moins pendant la durée d'exécution du programme[1]).
- l'idée de stocker les informations relatives à chaque contact dans une variable différente est, nous l'avons vu, condamnée par le fait que l'utilisateur doit pouvoir créer à volonté de nouveaux contacts.

Le premier problème peut être facilement résolu : au lieu d'une variable locale au constructeur, il suffirait d'utiliser une variable membre de la classe `monDialogue`.

> Les variables membre ont évidemment une durée de vie qui est celle de l'instance à laquelle elles appartiennent. Dans le cas présent, la seule instance de `monDialogue` qui existe (et qui fait tout le travail...) est la variable locale `w` créée par `main()`. Sa durée de vie est donc celle de l'exécution de `main()`, c'est à dire toute la durée d'exécution du programme.

Le second problème, en revanche, ne peut être traité que si l'on dispose de variable capables de stocker simultanément plusieurs valeurs et acceptant, en outre, que de nouvelles valeurs leur soit ajoutées à n'importe quel moment. Ces caractéristiques sont précisément celles qu'offrent les collections.

2 - Collections : introduction aux `QMap`

La classe `QMap` permet de créer des variables capables de stocker simultanément plusieurs valeurs, un peu comme une `QString` contient simultanément plusieurs caractères.

Une première différence est que les `QMap` ne sont pas limitées aux caractères : elles peuvent contenir n'importe quel type de données (`int`, `double`, `CEtudiant`, etc.). Notez bien, toutefois, qu'une `QMap` particulière ne peut contenir qu'un seul type de données.

> En clair : il peut exister des `QMap` de `bool` et des `QMap` de `double`, par exemple, mais une même `QMap` ne peut pas contenir à la fois des `bool` et des `double`.

Une seconde différence est que les éléments d'une `QString` sont indexés par des entiers, alors que le mécanisme d'indexation des `QMap` peut utiliser n'importe quel type.

> Si `maChaine` est une `QString`, `maChaine[0]` désigne son premier caractère, `maChaine[1]` le second, et ainsi de suite. Si `traduction` est une `QMap` indexée par des `QString`, `traduction["lapin"]` peut désigner un de ses éléments, `traduction["machine à coudre"]` en désignant un autre.

Cette double souplesse offerte par les `QMap` s'accompagne évidemment de la nécessité de spécifier les options choisies. Créer une variable de type `QMap` exige donc de préciser non seulement qu'il s'agit d'une `QMap`, mais aussi quel type d'index elle utilise et quel type de données elle peut contenir :

```
QMap <int, QString> maCollectionDeChaines;
```

Une fois la variable créée, elle peut stocker autant de valeurs que nécessaire, à condition bien entendu de disposer d'assez de valeurs différentes pour les index.

> Le problème ne se pose évidemment pas si les index sont des `int` ou des `QString`, mais si vous créez une `QMap` indexée avec des `bool`, votre collection sera limitée à deux objets : celui ayant pour index `true` et celui ayant pour index `false`...

```
maCollectionDeChaines[0] = "zéro";
maCollectionDeChaines[1] = "un";
std::cout << maCollectionDeChaines[0]; //affiche "zéro"
```

1 Il serait légitime d'attendre d'un tel programme un stockage à plus long terme, permettant de retrouver les informations saisies même après avoir quitté et relancé le programme. Ce type de performance exige que le programme écrive et lise des fichiers, et nous devrons franchir encore quelques étapes dans notre étude du langage avant d'en arriver là.

La souplesse des `QMap` peut être illustrée par une collection "inverse" de la précédente :

```
5  QMap<QString, int> maCollectionDEntiers;
6  maCollectionDEntiers["zéro"] = 0;
7  maCollectionDEntiers["un"] = 1;
8  std::cout << maCollectionDEntiers["un"]; //affiche 1
9  ++maCollectionDEntiers["un"]; //maCollectionDEntiers["un"] contient maintenant 2
```

L'usage du type `QMap` exige la présence d'une directive `#include <qmap>`

Une particularité des `QMap` qu'il est important de remarquer est qu'il est possible de créer un élément simplement en le mentionnant. Ainsi, lors de l'exécution de :

```
10 if(maCollectionDEntiers["trente-six"] == 36) {
11   ...
12   }
```

L'exécution de la ligne *10* se solde par la **création** d'un nouvel élément dont l'index est bien "trente-six", mais dont la valeur est, comme celle de tout `int` non initialisé, parfaitement imprévisible...

La création d'éléments indésirables peut être évitée en s'assurant qu'ils existent avant de les mentionner. Les `QMap` proposent une fonction `contains()` qui permet par exemple d'écrire :

```
10 if( maCollectionDEntiers.contains("trente-six")
11   if(maCollectionDEntiers["trente-six"] == 36) {
12     ...
13     }
```

Le test de la ligne *11* n'étant effectué que si l'élément d'index "trente-six" existe déjà, il n'aura jamais pour effet de le créer...

Comme le nombre d'éléments contenus dans une `QMap` est susceptible de varier au cours de l'exécution du programme, une autre fonction importante est la fonction `count()`, qui permet justement d'obtenir d'une `QMap` le nombre d'éléments qu'elle contient :

```
14 int nbElements = maCollectionDEntiers.count() ;
```

3 - Collectionner les contacts

Notre problème étant le stockage de contacts multiples et en nombre imprévisible et variable, il est clair que nous avons besoin d'une collection dont les valeurs sont de type `CContact`.

La nature de l'index devant être utilisé est peut-être moins évidente. Étant donné que les seules fonctionnalités de navigation que propose notre programme sont "passer au suivant" et "revenir au précédant", il peut toutefois sembler naturel d'utiliser des entiers comme index[2]. Les entiers sont en effet en nombre infini (l'utilisateur pourra donc créer autant de contacts qu'il le souhaite, sans jamais buter sur un problème d'épuisement des clés utilisables) et offrent une relation d'ordre qui se prête facilement à l'implémentation des notions de "précédant" et "suivant" : le contact suivant est celui dont l'index vaut 1 de plus, le précédent celui dont l'index vaut 1 de moins.

Pour gérer ce type de navigation dans la collection, le programme aura donc besoin d'utiliser l'index du contact actuellement représenté à l'écran, et cette information sera donc stockée dans une autre variable membre de `monDialogue`, que nous baptiserons `m_positionActuelle`.

Création des variables `m_leRepertoire` et `m_positionActuelle`

Complétez votre définition de la classe `monDialogue` comme suggéré ci-dessous ☐ :

```
1  class monDialogue : public QDialog
2  {
3  Q_OBJECT
4  //déclaration des fonctions membre
5  public slots:
6      void f_reculer();
7      void f_enregistrer();
```

2 Si nous devions proposer une fonction de recherche à partir d'un nom, par exemple, nous pourrions nous poser la question du choix d'un index permettant de trouver le contact recherché sans avoir à parcourir systématiquement la collection...

```
8     void f_avancer();
9     void f_activerEnregistrer();
10 public:
11     explicit monDialogue(QWidget *parent = 0);
12     ~monDialogue();
13 //déclaration des variables membre
14 private:
15     Ui::monDialogue *ui;
16     int m_positionActuelle;
17     QMap <int, CContact> m_leRepertoire;
18 };
```

Version finale du constructeur

Maintenant que nous disposons des variables nécessaires, il est facile de stocker notre pseudo-contact dans le répertoire. Modifiez votre constructeur pour qu'il devienne :

```
1  monDialogue::monDialogue(QWidget *parent) :
2      QDialog(parent),
3      ui(new Ui::monDialogue)
4  {
5      ui->setupUi(this);
6      //création du pseudo-contact ("vide")
7      CContact bidon;
8      bidon.m_nom = "Saisir ici un nom";
9      bidon.m_prenom = "Saisir ici un prénom";
10     bidon.m_telephone = "Saisir ici un numéro de téléphone";
11     m_leRepertoire[0] = bidon;
12     affiche(0) ;
13 }
```

Le répertoire étant initialement vide, il est logique de donner au premier contact l'index 0.

Pour faire apparaître ce premier contact à l'écran, nous faisons appel à une fonction `affiche()` qui devra être capable, étant donné un index, d'afficher dans le dialogue le contact correspondant.

La fonction `affiche()`

L'affichage d'un texte dans un TextEdit peut être obtenu directement à l'aide de la fonction `setText()`. Comme la connaissance de l'index donne un accès direct au `CContact` qui doit être affiché, la définition de la fonction `affiche()` ne présente aucune difficulté particulière :

```
1  void monDialogue::affiche(int index)
2  {
3      CContact aAfficher = m_leRepertoire[index];
4      ui->nom->setText(aAfficher.m_nom);
5      ui->prenom->setText(aAfficher.m_prenom);
6      ui->telephone->setText(aAfficher.m_telephone);
7      m_positionActuelle = index;
8      ui->b_enregistrer->setEnabled(false);
9  }
```

Remarquez que la ligne *7* garantit que `m_positionActuelle` contiendra toujours l'index du contact affiché à l'écran, ce qui est précisément la raison d'être de cette variable. La ligne *8*, pour sa part, empêche l'utilisateur d'enregistrer un contact qui existe déjà (puisqu'on vient de l'afficher).

Ajoutez la fonction `affiche()` à votre classe `monDialogue`.

N'oubliez pas d'ajouter la déclaration de cette fonction dans monDialogue.h (elle peut prendre place dans la section `public :` de la classe).

La fonction `f_enregistrer()`

Le rôle de cette fonction est de récupérer le texte présent dans les LineEdit, de créer un `CContact` contenant ces informations et de l'ajouter au répertoire.

La fonction `text()` permet d'obtenir d'un LineEdit qu'il renvoie le texte qu'il contient.

La seule petite difficulté concerne l'index qui doit correspondre à ce nouveau contact. Elle est aisément résolue en remarquant que, puisque nos index commencent à 0, le nombre d'éléments actuellement présents dans la collection donne directement le premier index non utilisé.

Vous pouvez donc définir ainsi votre fonction `f_enregistrer()` :

```
1  void monDialogue::f_enregistrer()
2  {
       //création du nouveau contact
3      CContact nouveau;
4      nouveau.m_nom = ui->nom->text();
5      nouveau.m_prenom = ui->prenom->text();
6      nouveau.m_telephone = ui->telephone->text();
7      //insertion du nouveau contact dans le répertoire
8      int premierIndexLibre(m_leRepertoire.count());
9      m_leRepertoire[premierIndexLibre] = nouveau;
10     affiche(premierIndexLibre);
11 }
```

Remarquez que l'appel à `affiche()` (ligne 9) a deux effets : il désactive le bouton [Enregistrer] (ce qui empêche d'enregistrer plusieurs fois le même nouveau contact) et il met à jour la variable `m_positionActuelle` (ce qui permet un fonctionnement cohérent des boutons [<] et [>].

La fonction `f_avancer()`

Fondamentalement, le rôle de cette fonction n'est que de demander l'affichage du contact dont l'index est `m_positionActuelle+1`.

Il faut toutefois s'assurer qu'il existe bien un tel contact et, dans le cas contraire, on peut choisir de revenir au premier contact (effet de "circularité" du répertoire).

Il aurait aussi été possible de ne rien faire du tout lorsque l'utilisateur essaie d'avancer au-delà du dernier élément (effet de blocage à l'extrémité du répertoire), ou même de désactiver le bouton [>] lorsque l'élément affiché est le dernier.

Définissez donc ainsi votre fonction `f_avancer()` :

```
1  void monDialogue::f_avancer()
2  {
3      if(m_leRepertoire.contains(m_positionActuelle+1))
4          affiche(m_positionActuelle+1);
5      else
6          affiche(0);
7  }
```

La fonction `f_reculer()`

Par analogie avec la fonction `f_avancer()`, vous devriez être en mesure d'écrire vous même le corps de cette fonction...

Seule "difficulté" : comment désigne-t-on le dernier élément de la collection `m_leRepertoire` ?
Indice : il faut utiliser `count()`...

Vérifiez que votre programme fonctionne comme prévu .