Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator
Conteneurs

Lorsque le nombre de données manipulées par un programme est important, il n'est guère envisageable de créer une variable différente pour chacune d'entre elles. Les structures de contrôles étudiées au cours de la Leçon 4 ne sont en effet pas adaptées au traitement de données distinguées les unes des autres par leur nom : il n'existe, par exemple, aucun moyen de faire en sorte qu'une instruction figurant dans une boucle agisse sur une variable différente lors de chaque passage dans la boucle. Pour obtenir ce genre de possibilité, il faut recourir à des *structures de données* qui permettent une organisation plus claire et plus efficace du stockage de l'information par le programme. La librairie Qt propose plusieurs classes qui permettent de mettre en place de telles structures, et nous allons ici décrire deux des plus utiles et des plus faciles à utiliser.

1 - Notion de conteneur

Lorsqu'une variable est capable de stocker simultanément non pas une seule mais plusieurs valeurs, elle mérite le titre de conteneur.

Les `QString` (Leçon 6) sont donc des conteneurs. En effet, à la différence d'un `char` qui n'est capables de stocker qu'une seule valeur, une `QString` peut en contenir un grand nombre. La classe `QString` est cependant extrêmement spécialisée : elle n'est adaptée qu'au stockage de texte. On réserve habituellement la qualification de "conteneur" aux classes qui sont capables de gérer des collections de données, quelle que soit la nature de celles-ci.

Cette indépendance des conteneurs par rapport au type des données qu'ils contiennent présente une limite :

Toutes les valeurs stockées dans un conteneur doivent être du même type.

Pour créer un conteneur, il nous faudra donc indiquer non seulement de quel genre de structure de données nous souhaitons disposer, mais également quel type de données vont prendre place dans la structure. La création d'une instance d'une classe conteneur prend donc la forme suivante :

```
uneStructure <typeDeDonnee> monConteneur;
```

Les conteneurs sont généralement programmés sous la forme de patrons de classes (class templates, en anglais), une technique que nous étudierons au cours de la Leçon 18. Les patrons exigent une notation spéciale lors de l'instanciation, pour indiquer à la variable créée quel genre d'objets elle doit s'attendre à stocker.

Que peut-on mettre dans un conteneur ?

Les conteneurs ne sont pas limités aux types prédéfinis : ils peuvent parfaitement contenir des collections d'instances de classes que vous avez définies vous-même dans votre programme, ou qui sont définies dans la librairie Qt. Notez toutefois que

Si les valeurs contenues dans un conteneur sont des instances d'une classe que vous avez créée, celle-ci ne doit être privée ni de ses constructeurs par défaut et par copie (cf. Leçon 10), ni de son opérateur d'affectation (cf. Leçon 11).

Pas de panique : cette restriction n'a aucune conséquence concrète tant que vous n'utilisez pas les mécanismes présentés dans les Leçons 10 et 11...

Certaines fonctions du conteneur ne seront utilisables que si votre classe dispose d'un opérateur == permettant de tester l'égalité de deux valeurs (cf. Leçon 11).

Dans la suite de ce document, les fonctions qui exigent la présence d'un opérateur == seront signalées par une note de bas de page. Si vous avez besoin d'utiliser l'une d'entre-elles pour traiter une collection d'instances d'une de vos classes, vous devrez donc étudier la Leçon 11.

En outre, l'insertion dans un conteneur de valeurs de type `QFile` ou `QTextStream` (ou d'instances d'une classe dont un membre est de l'un de ces types) se traduirait par la copie d'un `QFile` ou d'un `QTextStream` et doit donc être proscrite (cf. Leçon 7).

La Leçon 17 nous fournira un moyen de contourner cette limitation en évitant la création de variables membre dont la copie pose problème.

2 - La classe `QLinkedList`

Une des structures de données les plus simples est la liste chainée. Elle n'offre qu'une seule façon d'accéder aux valeurs : il faut parcourir la liste séquentiellement.

La classe `QLinkedList` permet de parcourir les listes en commençant par leur début ou par leur fin, mais le temps nécessaire pour atteindre la valeur centrale croîtra inexorablement avec le nombre de données stockées.

Si les données sont peu nombreuses, ou s'il est rarement nécessaire de rechercher l'une d'entre-elles, la structure de liste est un bon choix. Un programme qui effectue des millions de recherche dans une liste comportant elle-même des milliers d'éléments va, en revanche, devoir parcourir des *milliards* de liens pour passer d'un élément au suivant (ou au précédent), et il ne faudra donc pas s'étonner si sa lenteur d'exécution s'avère insupportable.

Instanciation

L'utilisation de la classe `QLinkedList` exige, bien entendu, la présence d'une directive

```
#include <QLinkedList>
```

Si cette formalité est remplie, l'instanciation se fait sur le modèle annoncé plus haut :

```
QLinkedList <double> lesValeurs; //une collection de nombres décimaux
```

Mettre des valeurs dans une liste

La classe `QLinkedList` offre deux fonctions nommées `prepend()` et `append()` qui permettent respectivement d'ajouter un élément en début ou en fin de liste. Bien entendu, ces deux fonctions attendent un paramètre du type accepté par le conteneur :

```
1 QLinkedList <double> lesValeurs;    //une collection de nombres décimaux
2 double x;
3 for(x = 25 ; x > 0.004 ; x = x / 2)//ajoute 13 valeurs dans la liste
4   lesValeurs.append(x);
```

> L'utilisation de `prepend()` en ligne *4* aurait permis d'obtenir une liste contenant les même 13 valeurs, mais rangées dans un l'ordre inverse (ie. croissant).

L'opérateur d'affectation permet de donner à une liste un contenu identique à celui d'une autre liste du même type :

```
5 QLinkedList <double> seconde;//une autre collection de décimaux
6 seconde = lesValeurs;        //recopie les 13 valeurs dans la nouvelle liste
```

Il est aussi possible de transférer dans une liste l'intégralité du contenu d'une seconde liste, sans pour autant perdre le contenu initial de la première. On utilise pour cela l'opérateur `+`, qui concatène les contenus de deux listes :

```
7 lesValeurs = lesValeurs + seconde; //lesValeurs en contient maintenant 26
```

Parcourir une liste

Une fois qu'une liste contient des valeurs, il faut être capable d'utiliser ces valeurs pour effectuer des traitements. Le parcours d'une liste (et, plus généralement, d'un conteneur quelconque) repose sur l'utilisation d'un itérateur.

Un itérateur est une sorte de pointeur perfectionné qui a pour vocation de désigner un élément d'une collection. Si vous savez créer une collection, vous savez créer un itérateur qui lui est adapté, puisqu'il suffit de préciser que c'est un itérateur que vous souhaitez créer, et non une nouvelle collection :

```
8 QLinkedList <double>::Iterator doigt;
```

Le mot Iterator s'écrit avec un i MAJUSCULE.

Lors du parcours d'une `QlinkedList`, l'itérateur joue un rôle analogue à celui du doigt qui nous aide à nous souvenir où on en est au cours du parcours d'une liste écrite sur une simple feuille de papier. Il faut simplement savoir que :

- la fonction `begin()` renvoie une valeur désignant le premier élément de la liste au titre de laquelle elle est invoquée (ce qui permet de mettre le doigt au début de la liste);

- l'opérateur `++` ordonne à l'itérateur sur lequel il est appliqué de désigner l'élément suivant celui qu'il désigne actuellement (ce qui permet d'avancer le doigt d'un cran dans la liste) ;

- la fonction `end()` renvoie la valeur d'un itérateur qui a avancé au delà de la fin de la liste (ce qui permet de voir qu'on est arrivé au bout de la liste) ;

- on accède à l'élément désigné par un itérateur en déréférençant celui-ci (ce qui permet d'utiliser la valeur actuellement pointée par le doigt).

Si nous souhaitons, par exemple, ajouter deux à chacune des valeurs contenues dans la liste `lesValeurs`, nous écrirons donc quelque chose comme :

```
9  for(doigt = lesValeurs.begin() ; doigt != lesValeurs.end() ; ++doigt)
10   *doigt = *doigt + 2;
```

Les opérateurs `-` et `--` permettent de parcourir une liste en commençant par la fin :

```
11 for(doigt = lesValeurs.end() - 1 ; doigt != lesValeurs.end() ; --doigt)
12   ui->ecran->append(QString::number(*doigt));
```

> Puisque `end()` renvoie une valeur pointant APRES le dernier élément, il faut enlever 1 à cette valeur pour désigner le dernier élément. Bien que le parcours rétrograde s'achève évidemment lors de la décrémentation d'un itérateur pointant sur le *premier* élément de la liste, c'est toujours la fonction `end()` qui fournit la valeur qu'aura l'itérateur après cette opération.

Parcourir une liste constante

Comme toutes les variables, les listes peuvent être rendues constantes. Ce phénomène apparaît souvent à l'occasion de la transmission d'une liste à une fonction : pour éviter la duplication locale d'un objet qui peut être très volumineux, il faut que la fonction dispose d'un paramètre de type référence. Si la fonction n'a pas vocation à modifier la liste, il est préférable d'utiliser un paramètre de type "référérence à une liste constante". Le parcours de la liste par la fonction exige alors le recours à un `ConstIterator`, qui ne permet pas la modification de l'objet qu'il désigne :

```
1  int calculeTailleTotale(const QLinkedList<QString> & liste)
2  {
3  int resultat = 0;
4  QLinkedList <QString>::ConstIterator doigt;
5  for(doigt = liste.begin() ; doigt != liste.end() ; ++doigt)
6    resultat += doigt->length();
7  return resultat;
8  }
```

Ajouter des données dans une liste

Lorsqu'un itérateur désigne l'un des éléments d'une liste, la fonction `insert()` permet, comme son nom l'indique, d'insérer un nouvel élément, qui prendra place avant l'élément désigné par l'itérateur. La fonction suivante insère la valeur 999 dans la liste qui lui est passée, de façon à ce que cette valeur figure immédiatement avant la première valeur inférieure à un :

```
1  void insere999(QLinkedList <int> lesValeurs)
2  {
3  QLinkedList <int>::Iterator doigt = lesValeurs.begin();
4  while(doigt != lesValeurs.end() && * doigt > 1)
5     ++doigt;
6  lesValeurs.insert(doigt, 999);
7  }
```

Retirer des valeurs d'une liste

La fonction `clear()` élimine toutes les valeurs de la liste au titre de laquelle elle est appelée.

La fonction `removeAll()` permet une élimination plus ciblée. Si on lui passe une valeur du type contenu dans la liste au titre de laquelle elle est appelée[1], cette fonction élimine tous les éléments de la liste qui sont égaux à cette valeur, et renvoie le nombre d'éléments éliminés :

```
lesValeurs.removeAll(25); //élimine toutes les occurrences de 25
```

La fonction `erase()` permet une élimination encore plus ciblée : elle a comme paramètre un itérateur, et seul l'élément que celui-ci désigne sera supprimé. La fonction `erase()` renvoie un itérateur désignant l'élément qui suivait celui qui vient d'être éliminé.

Puisqu'elle détruit l'objet concerné,

La fonction `erase()` rend invalide l'itérateur utilisé pour lui désigner la victime.

Il faut donc veiller à ce que le parcours de la liste ne soit pas rendu impossible par cette invalidité. La fonction suivante, par exemple, retire d'une liste les valeurs supérieures à un certain seuil :

```
1 void elimineLesGrands(QLinkedList<double> & liste, double seuil)
2 {
3 QLinkedList<double>::Iterator doigt = liste.begin();
4 while( doigt != liste.end())
5   if(* doigt > seuil)
6     doigt = liste.erase(doigt);
7   else
8     ++doigt;
9 }
```

L'appel à erase() (ligne *6*) s'accompagne d'un oubli immédiat de la valeur de l'itérateur qui désignait la victime, puisque cette valeur est remplacée dans la variable doigt par celle renvoyée par erase(). Comme cette fonction renvoie une valeur désignant l'élément qui suivait la victime, il faut utiliser un else pour éviter d'incrémenter l'itérateur dans ce cas (ce qui conduirait "oublier" d'examiner les éléments de la liste placés immédiatement après une valeur supprimée).

Les fonctions `pop_front()` et `pop_back()` éliminent respectivement le premier et le dernier élément de la liste au titre de laquelle elles sont appelées.

> Associées aux fonctions d'insertion `prepend()` et `append()`, les fonctions `pop_front()` et `pop_back()` permettent de gérer aisément une liste selon une logique FIFO ("first in, first out" c'est à dire "premier arrivé, premier servi", comme dans une file d'attente) ou selon une logique LIFO ("last in, first out", c'est à dire "dernier arrivé, premier reparti", comme dans une pile d'assiettes, par exemple).

Obtenir des renseignements sur le contenu d'une liste

Deux `QLinkedList` du même type peuvent être comparées à l'aide des opérateurs `==` et `!=`. Les listes sont déclarées identiques si elles contiennent les mêmes valeurs, dans le même ordre.

La fonction `isEmpty()` renvoie `true` si la liste au titre de laquelle elle est appelée est vide, et `false` dans le cas contraire.

La fonction `count()` renvoie le nombre d'éléments contenus dans la liste au titre de laquelle elle est appelée. Si on lui passe une valeur du type contenu dans la liste, elle renvoie le nombre d'apparitions de cette valeur dans la liste[2].

La fonction `contains()` renvoie un booléen indiquant si la valeur qui lui est passée comme argument figure ou non dans la liste au titre de laquelle elle est appelée[2].

1 Pour être utilisable, la fonction `removeAll()` exige que le type des données contenues dans la liste permette la comparaison de deux valeurs à l'aide de l'opérateur ==.

2 Cet usage de la fonction `count()` exige que le type des données contenues dans la liste permette la comparaison de deux valeurs à l'aide de l'opérateur ==.

3 - La classe QMap

La classe QMap est destinée à permettre de créer des conteneurs qui stockent des couples "clé/valeur". La fonction essentielle d'une QMap est donc, étant donnée une clé, de retrouver la valeur associée. Du fait de la technique utilisée, l'accès à la valeur recherchée est rapide, ce qui rend les QMap mieux adaptées que les QLinkedList aux cas où les données sont nombreuses et où de nombreuses recherches doivent être effectuées.

Instanciation

Les QMap gérant des couples, il est naturel que leur création implique de spécifier deux types : celui des clés et celui des valeurs.

```
1 QMap <QString, int> inventaire;
```

L'utilisation d'une QMap implique, évidemment, la présence d'une directive

```
#include <qmap>
```

Le type de la clé doit permettre la comparaison à l'aide de l'opérateur <.

Mettre des valeurs dans une QMap

La fonction insert() ajoute un couple dans la QMap au titre de laquelle elle est appelée. Elle attend, bien entendu, deux arguments dont les types doivent être compatibles avec ceux annoncés pour les clés et pour les valeurs :

```
2 inventaire.insert("pierre", 1);
3 inventaire.insert("maisons", 2);
4 inventaire.insert("ruines", 3);
```

L'insertion d'un couple dont la clé est identique à celle d'un couple déjà présent dans la QMap a pour effet de remplacer celui-ci.

L'ajout d'un élément peut également être obtenu à l'aide de l'opérateur [], la valeur à associer à la clé étant simplement affectée au nouvel élément :

```
5 inventaire["raton laveur"] = 1;
```

> L'usage des opérateurs [] et = présente l'avantage de ne laisser planer aucune ambiguïté sur ce qui se passe lorsqu'il existe déjà un élément utilisant la clé spécifiée...

L'opérateur d'affectation permet de transférer le contenu d'une QMap dans une autre, à condition, toutefois, que ces deux collections utilisent des clés et des valeurs de même type :

```
6 QMap <QString, int> uneAutre;
7 uneAutre = inventaire;
```

Obtenir des renseignements sur le contenu d'une QMap

La fonction isEmpty() renvoie true si la QMap au titre de laquelle elle est appelée est vide, et false dans le cas contraire.

La fonction count() renvoie le nombre d'éléments contenus dans la QMap au titre de laquelle elle est appelée.

La fonction contains() renvoie true si la valeur qui lui est passée est une clé utilisée par la QMap au titre de laquelle elle est appelée, et false dans le cas contraire3.

Parcourir une QMap

L'usage d'un itérateur d'un type adapté permet de parcourir une QMap tout aussi facilement qu'une QLinkedList. L'accès à l'élément désigné par l'itérateur pose cependant un problème, puisque cet élément est un couple. Les itérateurs sur QMap proposent deux fonctions membres nommées key() et value(), qui permettent respectivement d'accéder à la clé et à la valeur de

l'élément désigné par l'itérateur au titre duquel elles sont appelées. Le fragment de code suivant parcours une QMap en faisant la somme des valeurs contenues dans les éléments dont la clé n'est pas "maison" :

```
8  QMap <QString, int>::Iterator doigt;
9  int nbNonMaison = 0;
10 for( doigt = inventaire.begin() ; doigt != inventaire.end() ; ++doigt)
11   if(doigt.key() != "maison")
12     nbNonMaison = nbNonMaison + doigt.value();
```

Tout comme dans le cas des QLinkedList, le parcours d'une QMap constante devra évidemment faire appel à un ConstIterator.

Lorsque les clés utilisées peuvent être facilement générées, il est parfois possible de parcourir une QMap sans utiliser d'itérateur. C'est ce que fait la boucle 6-7 de l'exemple suivant :

```
1  QMap <int, int> donnees;
2  int n;
3  for (n=0 ; n < 542 ; ++n)
4    donnees[n] = rand(); //un nombre tiré au hasard (cf annexe 2)
5  int somme = 0;
6  for (n=0 ; n < donnees.count() ; ++n)
7    somme += donnees[n];
```

La simplicité de cette technique la rend séduisante, mais elle est intrinsèquement moins sûre que le recours à un itérateur : c'est **vous** qui devez veiller à l'**exhaustivité du parcours** de la QMap et prendre soin de ne pas lui ajouter au passage de nouveaux éléments.

Retirer des valeurs d'une QMap

La fonction clear() élimine toutes les valeurs de la QMap au titre de laquelle elle est appelée.

La fonction remove() permet une élimination plus sélective. Si on lui passe une valeur du type de la clé de la QMap au titre de laquelle elle est appelée, cette fonction élimine l'élément correspondant :

```
8  inventaire.remove("raton laveur");
```

La fonction erase() permet également d'éliminer une valeur de la collection, qui lui est désigné par un itérateur. Le fragment de code suivant élimine de la QMap tous les éléments dont la valeur est égale à 2 :

```
9  doigt = inventaire.begin();
10 while(doigt != inventaire.end())
11   if(doigt.value() == 2)
12     doigt = inventaire.erase(doigt);
13   else
14     ++doigt;
```

Accéder à une valeur stockée dans une QMap

L'opérateur [] offre un accès immédiat à un élément désigné par sa clé :

```
15 int nbRatonsLaveurs = inventaire["raton laveur"];
```

Il faut cependant tenir compte du fait que, lorsque la QMap ne contient aucun élément correspondant à la clé utilisée, l'opérateur [] à pour effet d'en créer un. Si une telle création n'est pas souhaitable, il faut donc commencer par s'assurer que l'élément existe :

```
16 int nbMaisons = 0;
17 if (inventaire.contains("maisons"))
18   nbMaisons = inventaire["maisons"];
```

Une autre façon de procéder est de faire appel à la fonction `find()`, qui renvoie un itérateur vers l'élément correspondant à la clé qu'on lui passe comme argument, ou la même valeur que `end()` si l'élément recherché n'existe pas :

```
19 int nbRuines = 0;
20 if(inventaire.find("ruines") != inventaire.end())
21   nbRuines = inventaire["ruines"];
```

4 - Bon, c'est gentil tout ça, mais ça fait déjà 6 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Pour stocker en mémoire de grandes quantités de données, on peut utiliser des conteneurs.

2) Les données stockées dans un conteneur sont toujours toutes du même type.

3) On peut créer un conteneur pour des données de n'importe quel type, mais :

   - si ce type est une classe, celle-ci ne doit pas avoir été privée de l'opérateur d'affectation ou des constructeurs par défaut et par copie dont elle dispose normalement ;

   - les clés d'une `QMap` doivent être d'un type qui permet la comparaison à l'aide de <.

   - les fonctions impliquant la recherche d'une valeur dans un conteneur (qu'il s'agisse de la supprimer, d'en vérifier la présence, d'en compter les occurrences ou simplement de la localiser) ne sont utilisables qu'avec les types de données supportant l'opérateur ==.

4) Selon le type de manipulations effectuées sur les données, les différents conteneurs disponibles sont plus ou moins appropriés.

5) Les listes ne conviennent pas lorsqu'on a besoin d'accéder fréquemment aux données, dans un ordre imprévisible.

6) Le parcours d'un conteneur pour utiliser les données qui y sont stockées fait le plus souvent appel à un itérateur.

7) Lorsque le parcours d'un conteneur s'accompagne de la suppression de certains éléments, il faut veiller à ne pas "perdre le fil" en rendant invalide l'itérateur utilisé.

8) Les `QMap` se prêtent parfois à un parcours ne reposant pas sur un itérateur mais sur la connaissance des clés utilisées. Cette possibilité doit être utilisée avec prudence.