



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 8 Conteneurs

1 - Notion de conteneur.....	2
Que peut-on mettre dans un conteneur ?	2
2 - La classe QList	3
Instanciation	3
Mettre des valeurs dans une liste	3
Parcourir une liste.....	3
Parcourir une liste constante.....	4
Ajouter (encore) des données dans une liste	5
Retirer des valeurs d'une liste.....	5
Obtenir des renseignements sur le contenu d'une liste	6
Chercher une valeur dans une liste	6
3 - La classe QMap	7
Instanciation	7
Mettre des valeurs dans une QMap	7
Obtenir des renseignements sur le contenu d'une QMap	7
Parcourir une QMap	8
Retirer des valeurs d'une QMap	8
Accéder à une valeur stockée dans une QMap.....	9
4 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?	9

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.

D'un certain point de vue, les `QString` (Leçon 6) sont donc des conteneurs. En effet, à la différence d'un `char` qui n'est capable de stocker qu'une seule valeur, un `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 23. 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...

De plus,

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, vous devrez donc étudier d'abord 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 12 nous fournira un moyen de contourner cette limitation.

2 - La classe QValueList

Une des structures de données les plus simples est la liste. Le principe de la liste est de n'offrir qu'une seule façon d'accéder aux valeurs qu'elle contient : il faut la parcourir séquentiellement.

La classe QValueList permet de créer des listes bi-directionnelles. Il sera donc possible de parcourir ces 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 QValueList exige, bien entendu, la présence d'une directive

```
#include "qvaluelist.h"
```

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

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

Mettre des valeurs dans une liste

La classe QValueList 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 QValueList <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 ordre croissant et non décroissant.

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

```
5 QValueList <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 QValueList <double>::Iterator unIt;
```

Attention : le mot `Iterator` s'écrit avec un `i` MAJUSCULE

Une fois l'itérateur créé, le parcours d'une liste est facile à obtenir, sachant que :

- la fonction `begin()` renvoie un itérateur désignant le premier élément de la liste au titre de laquelle elle est invoquée ;
- l'opérateur `++` ordonne à l'itérateur sur lequel il est appliqué de désigner l'élément suivant celui qu'il désigne actuellement ;
- la fonction `end()` renvoie la valeur qu'a un itérateur lorsqu'il est sorti de la liste ;
- on accède à l'élément désigné par un opérateur en `déréférençant` celui-ci.

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

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

Dans le cas de collections d'objets complexes, la description des traitements à effectuer est souvent alourdie par le fait que les itérateurs ne supportent pas l'opérateur `->`, qui permettrait la sélection indirecte. L'accès aux membres de l'élément en cours de traitement nécessite donc l'emploi de `parenthèses` pour exiger que le `déréférencement` ait lieu avant la `sélection` :

```
1 QValueList <QString> lexique;
2 remplissage(& lexique); //une fonction qui met des mots dans la liste
3 QValueList <QString>::Iterator it;
4 int tailleTotale = 0;
5 for(it = lexique.begin() ; it != lexique.end() ; ++it)
6     tailleTotale += (*it).length();
```

La création d'une `référence` désignant le même élément que l'itérateur permet, dans ce cas, d'alléger un peu la notation :

```
for(it = lexique.begin() ; it != lexique.end() ; ++it)
{
    QString & enCours = *it; //donne un nom à l'objet désigné par l'itérateur
    tailleTotale += enCours.length(); //plus besoin de déréférencer it !
}
```

La fonction `fromLast()` et l'opérateur `--` permettent, le cas échéant, de parcourir une liste en commençant par la fin :

```
1 QValueList <double> lesValeurs; //une collection de nombres décimaux
2 remplissage(& lesValeurs); //une fonction qui met des valeurs dans la liste
3 QValueList <double>::Iterator unIt;
4 for(unIt = lesValeurs.fromLast() ; unIt != lesValeurs.end() ; --unIt)
5     *unIt -= 4;
```

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 `"référence"` à une liste `constante`.

Le parcours de la liste par la fonction exige alors le recours à un `ConstIterator`, qui interdit la modification de l'objet qu'il désigne :

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

Ajouter (encore) 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(QValueList <int> les Valeurs)
2 {
3     QValueList <int> unIt = lesValeurs.begin();
4     while(unIt != lesValeurs.end() && *unIt > 1)
5         ++unIt;
6     lesValeurs.insert(unIt, 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.

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).

La fonction `remove()` permet une élimination plus ciblée¹. Si on lui passe une valeur du type contenu dans la liste au titre de laquelle elle est appelée, cette fonction élimine tous les éléments de la liste qui sont égaux à cette valeur, et renvoie le nombre d'éléments éliminés :

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

Si le paramètre transmis à la fonction `remove()` est un itérateur, seul l'élément que celui-ci désigne sera éliminé de la liste. La fonction `remove()` renvoie alors un itérateur désignant l'élément qui suivait celui qui vient d'être éliminé. La fonction suivante, par exemple, retire d'une liste de doubles toutes les valeurs supérieures à un certain seuil :

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

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

Du fait de la nature de cette fonction, l'élément qu'on lui désigne n'existe plus une fois qu'elle a fait son travail. Il faut donc veiller à ce que le passage à l'élément suivant (qui permet de parcourir la liste) ne soit pas rendu impossible par cette invalidité.

Dans le fragment de code précédent, l'appel à `remove()` s'accompagne d'un oubli immédiat de la **valeur de l'itérateur qui désignait la victime**, puisque cette valeur est **remplacée** dans la variable `it` par celle renvoyée par `remove()`. Comme cette fonction renvoie une valeur désignant l'élément qui suivait la victime, il est nécessaire d'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).

Plutôt que d'utiliser la valeur renvoyée par `remove()`, on peut aussi postfixer l'opérateur `++` :

¹ Si les données contenues dans la liste sont des instances d'une classe, cette fonction exige que la classe en question dispose d'un opérateur `==`.

```

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

```

A la ligne 6, l'expression `it++` doit être évaluée avant l'appel de `remove()`, puisqu'elle détermine la valeur transmise à cette fonction. L'itérateur est donc incrémenté, ce qui lui fait désigner l'élément qui suit la victime. Comme l'opérateur `++` est postfixé, la valeur de l'expression `it++` reste toutefois celle qu'avait l'itérateur avant l'incrémement, c'est à dire une valeur désignant la victime. C'est donc bien cette valeur qui est transmise à `remove()`.

Obtenir des renseignements sur le contenu d'une liste

Deux `QValueList` 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.

La fonction `contains()` renvoie le nombre d'occurrences de la valeur qui lui est passée comme argument dans la liste au titre de laquelle elle est appelée².

Chercher une valeur dans une liste

La fonction `find()` renvoie un itérateur sur le premier élément de la liste dont la valeur est identique à l'argument qui lui est transmis². Si aucune valeur convenable ne figure dans la liste, `find()` renvoie la même valeur que `end()`.

La fonction suivante enlève de la liste de `double` qui lui est passée comme premier argument une seule occurrence de la valeur qui lui est passée comme second argument :

```

1 void enleveUneOccurrence(QValueList <double> & liste, double valeur)
2 {
3     QValueList <double>::Iterator it = liste.find(valeur);
4     if(it != liste.end())
5         liste.remove(it);
6 }

```

La fonction `find()` peut aussi accepter comme premier paramètre un itérateur désignant l'élément de la liste à partir duquel la recherche doit être effectuée. Cette possibilité est utile lorsque la liste contient plusieurs valeurs identiques et que toutes doivent être traitées :

```

1 void enleveToutesLesOccurrences(QValueList <double> & liste, double valeur)
2 {
3     QValueList <double>::Iterator it = liste.find(valeur);
4     while(it != liste.end())
5     {
6         it = liste.remove(it);
7         it = liste.find(it, valeur);
8     }
9 }

```

Après exécution de 6, l'itérateur désigne l'élément qui suivait celui qui vient d'être éliminé. La recherche effectuée par 7 ne repart donc pas du début de la liste, ce qui fait gagner du temps et serait indispensable si le traitement à appliquer n'était pas une suppression.

Si les fonctions membre de la classe `QValueList` permettent d'effectuer des recherches dans une liste sans avoir à programmer explicitement un parcours de la liste, il faut cependant se

² Si les données contenues dans la liste sont des instances d'une classe, cette fonction exige que la classe en question dispose d'un opérateur `==`.

souvenir que ces fonctions doivent procéder à un parcours séquentiel et éventuellement exhaustif de la liste. Cette contrainte compromet gravement les performances de certains programmes, et une caractéristique essentielle de la classe `QMap` est qu'elle s'en affranchit.

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 très rapide et ne s'accroît pas de façon perceptible lorsque le nombre de couples clé/valeur présents dans le conteneur augmente.

Ces caractéristiques font des `QMap` les structures de données idéales dans les 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.h"
```

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);
5 inventaire.insert("fossoyeurs", 4);
```

L'insertion d'un couple dont la clé est déjà utilisée par un couple 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 :

```
6 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 `QMap`, à condition, toutefois, que ces deux collections utilisent des clés et des valeurs de même type :

```
7 QMap <QString, int> uneAutre;
8 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.

³ Si les clés utilisées par la `QMap` sont des instances d'une classe, cette fonction exige que la classe en question dispose d'un opérateur `==`.

Parcourir une QMap

L'usage d'un `itérateur` d'un `type adapté` permet de parcourir une QMap tout aussi facilement qu'une QList (11). 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 `data()`, 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 parcourt une QMap en faisant la somme des valeurs (13) contenues dans les éléments dont la clé n'est pas "maison" (12) :

```

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

```

L'ordre dans lequel les éléments d'une QMap sont rencontrés lors du parcours est imprévisible.

Tout comme dans le cas des QList, 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 :

```

14 QMap<int, double> donnees;
15 int n;
16 for (n=0 ; n < 542 ; ++n)
17     donnees[n] = rand(); //un nombre tiré au hasard (cf annexe 2)
18 int somme = 0;
19 for (n=0 ; n < donnees.count() ; ++n)
20     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 contenues dans 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 :

```

21 inventaire.remove("raton laveur");

```

Si le paramètre transmis à la fonction `remove()` est un itérateur, c'est l'élément désigné par celui-ci qui sera éliminé de la liste. Le fragment de code suivant élimine de la QMap tous les éléments dont la valeur est égale à 2 :

```

22 itMap = inventaire.begin();
23 while(itMap != inventaire.end())
24     if(itMap.data() == 2)
25         inventaire.remove(itMap++);
26     else
27         ++itMap;

```

Contrairement à son homologue opérant sur les QList, la fonction `remove()` disponible dans la classe QMap ne renvoie pas l'adresse de l'élément suivant celui qui a été détruit. Il est donc indispensable d'utiliser l'**opérateur ++ postfixé** pour procéder simultanément au parcours du conteneur et à l'élimination de certains éléments.

Accéder à une valeur stockée dans une QMap

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

```
28 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 [] a 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 :

```
29 int nbMaisons = 0;  
30 if (inventaire.contains("maisons")  
31     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 argument⁴, ou la même valeur que `end()` si l'élément recherché n'existe pas :

```
32 int nbRuines = 0;  
33 if(inventaire.find("ruines") != inventaire.end())  
34     nbRuines = inventaire["ruines"];
```

4 - Bon, c'est gentil tout ça, mais ça fait déjà 8 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 des constructeurs par défaut et par copie et de l'opérateur d'affectation dont elle dispose normalement ;
 - 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 examiner 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.

⁴ Si les données contenues dans la liste sont des instances d'une classe, cette fonction exige que la classe en question dispose d'un opérateur ==.