



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

Apprendre C++ avec Qt : Leçon 17 Allocation dynamique encapsulée

1 - Confier l'allocation dynamique aux constructeurs.....	2
Première approche.....	2
Le problème du constructeur par copie.....	3
Le bon constructeur par copie.....	4
Le problème de l'opérateur d'affectation.....	4
La bonne fonction operator =().....	5
2 - Confier la libération de la mémoire au destructeur.....	5
3 - Opérateurs dont la signification se trouve remise en cause.....	6
Les opérateurs de comparaison.....	6
L'opérateur sizeof.....	7
Les opérateurs d'insertion et d'extraction.....	7
4 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?.....	8

La gestion de la mémoire allouée dynamiquement obéit à une logique d'appariement entre les appels à `new` (ou à `new[]`) et les appels à `delete` (ou à `delete[]`). Cet appariement est crucial car, comme nous l'avons vu au cours de la [Leçon 12](#) :

- si une zone de mémoire allouée dynamiquement n'est jamais libérée, le programme provoque une fuite de mémoire qui risque de perturber le fonctionnement du système ;
- les conséquences d'une tentative de libération d'une zone qui n'est pas (ou plus) allouée dynamiquement sont aussi imprévisibles que probablement catastrophiques.

La [Leçon 10](#) a fait apparaître un autre appariement qui, lui, possède l'avantage d'être toujours parfait : la création d'une instance s'accompagne de l'appel à un constructeur, alors que la disparition de cette instance provoquera l'appel du destructeur. Il est donc tentant de profiter de l'exécution automatique de ces fonctions pour leur faire effectuer les appels à `new` et à `delete`, ce qui garantirait la symétrie parfaite de ces opérations.

1 - Confier l'allocation dynamique aux constructeurs

Imaginons, par exemple, que nous soyons confrontés à un problème impliquant la création de nombreux faux tableaux de valeurs décimales (ou de tout autre type de valeurs). L'approche la plus rudimentaire conduit à avoir un grand nombre de séquences du genre

```
1 double * unTableau = new double[tailleRequise];
2 if (unTableau == NULL)
    //au secours...
```

Confier l'allocation dynamique au constructeur d'une [classe](#) permettrait d'alléger notre code en écrivant plutôt quelque chose comme

```
CTableau unTableau(tailleRequise);
```

Première approche

La création d'une classe `CTableau` permettant cet usage ne semble a priori pas très difficile :

```
1 class CTableau
2 {
3 public:
4     CTableau(int taille);
5     double & operator[](int index){return ptr[index];}
6 protected:
7     double * ptr;
8 };
```

La surcharge (5) de l'opérateur crochet permet d'utiliser les instances de `CTableau` comme s'il s'agissait de simples tableaux :

```
CTableau unTableau(100);
unTableau[25] = 12; //attribue une valeur au 26° élément
```

Pour un usage réel, il serait très tentant de doter la fonction `operator[]()` de vérifications les empêchant d'accéder à des "éléments" inexistants, de façon à ce qu'une tentative telle que

```
unTableau[150] = 15; //erreur, unTableau n'a que 100 éléments !
```

se traduise par un message d'erreur et l'interruption du programme. Ceci ne présente guère de difficultés, mais s'écarterait du sujet qui nous occupe ici.

Notre constructeur doit donc procéder à une allocation dynamique, et assurer au passage la gestion (5-6) du cas où la mémoire demandée ne peut être obtenue.

```
1 CTableau::CTableau(int t)
2 {
3     if(ptr = new double[t]) //oui, c'est bien l'opérateur d'affectation
4         return;
5     QMessageBox::critical(0, "Leçon 17 - CTableau", "Mémoire épuisée");
6     qApp->exit(-1); //met brutalement fin à l'exécution du programme
7 }
```

Le problème du constructeur par copie

La définition explicite d'un constructeur possédant un paramètre de type entier suffit à empêcher le compilateur de générer un constructeur par défaut. Il reste en revanche possible d'utiliser la version par défaut du constructeur par copie :

```
1 CTableau unTab(100); //création d'un tableau de 100 éléments
2 CTableau unAutreTab(unTab); //construction d'un tableau par copie du 1er
```

Nous savons ([Leçon 10](#)) que la version par défaut du constructeur par copie se borne à effectuer l'équivalent d'une copie membre à membre. Dans notre cas, cela revient grosso modo à utiliser une classe qui serait définie ainsi :

```
1 class CTableau
2 {
3 public:
4     CTableau(int taille);
5     CTableau(const CTableau & modele) : ptr(modele.ptr) {}
6     double & operator[] (int index){return ptr[index];}
7 protected:
8     double * ptr;
9 };
```

Le rôle du constructeur par copie se limite ici à l'initialisation de l'unique variable membre à l'aide de la valeur que ce membre contient dans l'instance qui sert de **modèle**.

Examinons, par exemple, ce qui se passe lors de l'exécution du fragment de code suivant :

```
1 CTableau premierTableau(100);
2 premierTableau[25] = 12;
3 CTableau unAutre(premierTableau); //utilisation du constructeur par copie
4 unAutre[25] = 15; //premierTableau[25] contient maintenant 15
```

Les deux premières lignes créent un `premierTableau` et placent la valeur 12 dans la 26^e case de celui-ci. La ligne 3 crée un `unAutre` tableau, par copie du premier. Ceci signifie que la variable `unAutre.ptr` contient la même valeur que la variable `premierTableau.ptr`. En d'autres termes, les deux pointeurs désignent une seule et unique zone de mémoire. En dépit des apparences, la ligne 4 modifie donc le "contenu" du `premierTableau`.

Ce problème peut sembler marginal : après tout, si le constructeur par copie ne produit pas des copies vraiment utilisables, on peut facilement se passer de l'initialisation d'une nouvelle instance par copie du contenu des membres d'une autre instance. Grave erreur, car, en fait :

Une classe dont le constructeur par copie est inutilisable est une classe quasi inutilisable.

En effet, le constructeur par copie est implicitement mobilisé dans des situations aussi fréquentes que le passage de paramètres ou le renvoi d'une valeur par une fonction. Dans l'état actuel de la classe `CTableau`, une fonction comme

```
1 void test(CTableau unTab)
2 { //le paramètre est initialisé par COPIE de la valeur passée
3     unTab[0] = 0;
4 }
```

est en mesure de modifier les caractères contenus dans la zone de mémoire réservée par la variable utilisée pour l'appeler, bien que son paramètre ne soit ni un "pointeur sur" ni une "référence à" cette variable :

```
1 CTableau tab(10);
2 tab[0] = 255;
3 test(tab); //tab[0] prend la valeur 0 !
```

Oltre le fait que cette propriété est en contradiction flagrante avec l'usage habituel, il faut bien se souvenir que la fonction `test()` ne peut en aucun cas modifier les valeurs contenues dans les

variables membre de l'instance dont la valeur lui est passée lors de l'appel. Il lui sera donc impossible de maintenir la cohérence d'un objet un peu plus complexe qu'un CTableau, et il est à craindre que des difficultés insurmontables surgissent rapidement.

Le bon constructeur par copie

Lorsque la construction comporte une allocation dynamique, tous les constructeurs disponibles doivent assurer ce service de façon cohérente.

Le constructeur par copie ne doit donc pas se contenter de recopier la valeur du pointeur contenu dans l'instance qui lui sert de modèle. Il lui faut demander l'allocation d'une nouvelle zone de mémoire, d'une taille égale à celle désignée par le pointeur contenu dans le modèle, puis ensuite recopier le contenu de la zone associée au modèle dans la zone nouvellement réservée.

La taille de la zone de mémoire dont l'adresse est contenue dans l'instance modèle ne peut pas être déterminée par le constructeur par copie et doit donc être stockée dans une **variable membre supplémentaire**. Par ailleurs, plutôt que de dupliquer le code effectuant l'allocation dynamique, il vaut mieux l'isoler dans une "fonction de service", qui sera appelée par tous les constructeurs :

```
1 void CTableau::allocation(int t)
2 {
3     if(ptr = new double[t]) //oui, c'est bien l'opérateur d'affectation
4         return;
5     QMessageBox::critical(0, "Leçon 17 - CTableau", "Mémoire épuisée");
6     qApp->exit(-1); //met brutalement fin à l'exécution du programme
7 }
```

La classe CTableau sera donc définie de la façon suivante :

```
1 class CTableau
2 {
3 public:
4     CTableau(int t) : taille(t) {allocation(t);}
5     CTableau(const CTableau & modele);
6     double & operator[](int index){return ptr[index];}
7 protected:
8     double * ptr;
9     int taille;
10    void allocation(int t); //fonction de service utilisée par les constructeurs
11 };
```

Le constructeur par copie, pour sa part, sera défini ainsi :

```
1 CTableau::CTableau(const CTableau & modele) : taille(modele.taille)
2 {
3     allocation(taille);
4     int i;
5     for(i=0 ; i < taille ; ++i)
6         ptr[i] = modele.ptr[i];
7 }
```

Après avoir **initialisé le membre taille**, le constructeur appelle la fonction d'allocation, puis **recopie les données** présentes dans le modèle dans la **zone de donnée qui lui est propre**.

Le problème de l'opérateur d'affectation

Lorsqu'une classe est créée, le compilateur peut, en fonction des besoins et des possibilités, générer automatiquement non seulement des versions des constructeurs par défaut et par copie, mais également une version de la fonction `operator =()`, ce qui permet de procéder à **l'affectation directe d'une valeur à une instance** :

```
1 const int TAILLE = 10;
2 CTableau unTableau(TAILLE);
3 int i;
```

```

4 for (i=0 ; i < TAILLE ; ++i)
5     unTableau[i] = i;
6 CTableau unAutre(TAILLE);
7 unAutre = unTableau; //affectation d'une valeur à une instance

```

Tout comme le constructeur par copie fourni automatiquement, la fonction `operator =()` dont nous disposons par défaut se borne à effectuer l'équivalent d'une affectation membre à membre.

Cette approche s'avère évidemment aussi insuffisante ici qu'elle l'était dans le cas du constructeur par copie, et pour les mêmes raisons : après exécution du code présenté ci-dessus, les variables `unTableau.ptr` et `unAutre.ptr` contiennent la même adresse, ce qui signifie que toute instruction modifiant l'un des éléments prétendument stockés dans l'un des tableaux modifie également l'élément prétendument stocké à la même position dans l'autre tableau. Par ailleurs, la valeur qui était contenue dans `unAutre.ptr` a été perdue, ce qui signifie que la zone de mémoire initialement allouée pour stocker les données confiées à l'instance `unAutre` n'est plus accessible (elle ne peut plus être utilisée, ni même restituée au système avant la fin de l'exécution du programme, et une fuite de mémoire est donc inévitable). Après avoir été forcés à définir un constructeur par copie, nous sommes donc contraints à surcharger l'opérateur d'affectation.

La bonne fonction `operator =()`

Fondamentalement, la fonction prenant en charge l'opérateur d'affectation doit simplement prendre les données fournies par le paramètre et les copier dans la zone de mémoire gérée par l'instance au titre de laquelle la fonction a été appelée. Avant de procéder à cette copie, il convient toutefois d'ajuster (3-8) la taille de cette zone de mémoire.

```

1 CTableau & CTableau::operator =(const CTableau & autre)
2 {
3     if(taille != autre.taille)
4     {
5         delete[] ptr; //serait une catastrophe en cas d'auto-affectation !
6         taille = autre.taille;
7         allocation(taille);
8     }
9     int i;
10    for(i=0 ; i < taille; ++i)
11        ptr[i] = autre.ptr[i];
12    return * this;
13 }

```

2 - Confier la libération de la mémoire au destructeur

Une fois les constructeurs correctement définis et les opérateurs concernés surchargés de la façon adéquate, nous sommes certains que chaque instance de la classe `CTableau` possède sa propre zone de mémoire allouée dynamiquement, dont elle est la seule à faire usage. Lorsque l'instance disparaît, il convient donc de **libérer cette zone** :

```

1 class CTableau
2 {
3 public:
4     CTableau(int t) : taille(t) {allocation(t);}
5     CTableau(const CTableau & modele);
6     ~CTableau() {delete[] ptr;}
7     double & operator[](int index){return ptr[index];}
8     CTableau & operator =(const CTableau & valeur);
9 protected:
10    double * ptr;
11    int taille;
12    void allocation(int t);
13 };

```

Le destructeur étant automatiquement appelé dès lors qu'une instance cesse d'exister, une dangereuse source d'erreurs se trouve éliminée. Confier l'allocation aux constructeurs et la libération au destructeur présente le grand avantage de rendre la gestion de l'allocation dynamique entièrement automatique et "invisible" pour le code utilisateur de la classe. Si, par exemple, une fonction a besoin d'une instance locale, son code est considérablement simplifié :

```
1 void uneFonction(int p)
2 {
3   CTableau unTab(p); //allocation dynamique "invisible"
4   //... code de la fonction : unTab[i] = unTab[i+3]; etc...
5 } //unTab disparaît : libération automatique de la mémoire par le destructeur
```

La fonction est non seulement déchargée de l'appel à `new[]`, de la prise en charge de l'éventualité d'un échec d'allocation et de l'obligation d'appeler `delete[]` lorsque son travail est terminé, elle est en outre dispensée de l'usage explicite d'un pointeur. On dit communément qu'une classe telle que `CTableau` **encapsule** une fonctionnalité (l'allocation dynamique, en l'occurrence) lorsqu'elle en assume la charge et ne laisse à ses utilisateurs que les bénéfices correspondants.

3 - Opérateurs dont la signification se trouve remise en cause

A partir du moment où une classe comporte un pointeur sur une zone de donnée allouée dynamiquement, il existe une divergence entre la sémantique de la classe (selon laquelle les données sont *contenues* dans une instance) et sa réalité technique (les données sont en fait stockées *à l'extérieur* de cette instance). Les auteurs anglophones parlent à ce propos de *remote ownership*, ce qui pourrait se traduire par "télépossession" ou même par "colonie". Cette situation exige un réexamen du sens de certains opérateurs.

Les opérateurs de comparaison

Un opérateur de test d'égalité¹ peut adopter deux stratégies : soit il considère que deux instances ne sont identiques que lorsque toutes les variables membre ont le même contenu dans chacune des instances, soit il admet que les pointeurs peuvent contenir des adresses différentes, à condition que les zones pointées aient elles-mêmes des contenus identiques.

On parle parfois de comparaison de surface (*shallow*) dans le premier cas, et de comparaison profonde (*deep*) dans le second. Ces termes sont également utilisés pour les opérations de copie (*shallow copy* et *deep copy*), que celles-ci soient effectuées par l'opérateur d'affectation ou par un constructeur.

Selon la nature la nature de la classe, l'une ou l'autre de ces stratégies peut sembler plus adaptée, mais il arrive aussi que les deux types de comparaisons doivent rester possibles. Il faut alors, bien entendu, définir deux fonctions différentes.

Dans le cas de notre exemple, il semble clair que les opérations de comparaison doivent être profondes. La fonction `operator ==()` pourrait donc être définie ainsi :

```
1 bool CTableau::operator ==(const CTableau & valeur) const
2 {
3   if (taille != valeur.taille)
4     return false;
5   int i;
6   for (i=0 ; i < taille ; ++i)
7     if (ptr[i] != valeur.ptr[i])
8       return false;
9   return true;
10 }
```

¹ Le même raisonnement peut être tenu pour les opérateurs `!=`, `<`, `>`, `<=` et `>=`, s'ils s'appliquent à la classe considérée.

L'opérateur sizeof

L'application de l'opérateur `sizeof` à une classe qui comporte un pointeur sur une zone de données produit une valeur qui correspond à la taille d'une instance. L'espace occupé par les données externes à l'instance n'est donc pas pris en compte. L'opérateur `sizeof` n'étant pas surchargeable, il n'est pas possible de modifier la façon dont le calcul de taille est effectué.

L'intérêt d'une telle modification serait de toute façon douteux, car le fait que les données décrivant effectivement l'état d'une instance sont dispersées en mémoire s'oppose à la plupart des opérations qui seraient intéressées par une taille "profonde" (une copie brute dans un fichier, par exemple, ne peut évidemment pas être effectuée en fournissant à la fonction `QDataStream::writeBytes()` l'adresse de l'instance et une taille obtenue en additionnant la taille de cette instance et celle d'une zone de données située ailleurs en mémoire).

La valeur renvoyée par `sizeof` (c'est à dire la taille "de surface") peut, en revanche, être utilisée tout à fait normalement pour déterminer le **nombre d'éléments** d'un **vrai tableau d'instances** :

```
1 CTableau tab[] = {CTableau(100), CTableau(10)};
2 int nbElements = sizeof(tab) / sizeof(tab[0]);
```

La création d'un vrai tableau d'instances de `CTableau` n'est possible qu'en fournissant une liste de valeurs destinées à initialiser les éléments. En effet, comme la classe `CTableau` ne comporte pas de constructeur par défaut, une instruction telle que

```
CTableau tab[2]; //ERREUR : no appropriate default constructor available
```

ne peut être compilée car elle ne fournit pas l'information relative à la taille des zones devant être allouée dynamiquement, information qui est nécessaire à la création de chacun des éléments du tableau. Les valeurs de type `CTableau` nécessaires sont simplement obtenues par **appel explicite** du constructeur. Remarquez que toutes les instances ainsi créées ne procèdent pas nécessairement à l'allocation dynamique de zones de données de la même taille, ce qui ne perturbe en rien le calcul du nombre d'éléments effectué ensuite, puisque ces zones n'interviennent pas dans la valeur calculée par `sizeof`.

Les opérateurs d'insertion et d'extraction

Lorsqu'une classe comporte un membre de type pointeur, il y a peu de chances que la conservation dans un fichier de l'adresse contenue dans cette variable présente un réel intérêt. Il est plus vraisemblablement utile de placer dans le fichier une copie de l'information stockée à cette adresse. C'est, en effet, cette information qu'il faudra retrouver lors de la lecture du fichier, si l'on souhaite pouvoir reconstituer un état identique à celui de l'instance sauvegardée.

Dans le cas de notre classe `CTableau`, l'opérateur d'insertion pourrait être :

```
1 QTextStream & operator << (QTextStream & out, const CTableau & uneInstance)
2 {
3     out << uneInstance.taille << "\n";
4     int i;
5     for(i=0 ; i < uneInstance.taille ; ++i)
6         out << uneInstance.ptr[i] << "\n";
7     return out;
8 }
```

Comme cette fonction globale **accède aux variables membre** de l'instance qui lui est passée comme paramètre, elle devra être déclarée amie par la classe `CTableau`.

Le cas de l'opérateur d'extraction est légèrement compliqué par la nécessité d'un ajustement de la taille de la zone de stockage des données analogue à celui effectué par l'opérateur d'affectation :

```
1 QTextStream & operator >> (QTextStream & source, CTableau & uneInstance)
2 {
3     source >> uneInstance.taille;
4     delete uneInstance.ptr;
5     uneInstance.allocation(uneInstance.taille);
6     int i;
```

```
7 for(i=0 ; i < uneInstance.taille ; ++i)
8     source >> uneInstance.ptr[i];
9 return source;
10 }
```

Il est peut-être encore plus utile de prévoir un constructeur extrayant les données d'un flux. La taille pouvant être lue avant l'allocation dynamique, il devient possible d'exploiter un fichier sans connaître à l'avance la taille profonde des CTableau qui y sont décrits :

```
1 CTableau::CTableau(QTextStream & source) //constructeur "par extraction"
2 {
3     source >> taille;
4     allocation(taille);
5     int i;
6     for(i=0 ; i < taille; ++i)
7         source >> ptr[i];
8 }
```

Pour des raisons de clarté, les opérations de lecture et d'écriture dans les fichiers ont été ici débarrassées des vérifications qui seraient nécessaires dans un véritable programme.

4 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) La gestion dynamique de la mémoire peut être simplifiée et rendue plus sûre en confiant l'allocation aux constructeurs d'une classe et la libération au destructeur correspondant.
- 2) L'adoption de cette méthode crée une divergence entre la sémantique de la classe (selon laquelle une instance contient les données qui lui sont confiées) et sa réalité (une instance ne contient en fait qu'un pointeur sur les données).
- 3) Les classes ainsi conçues pratiquent ce qu'on peut appeler la "télépossession" (*remote ownership*).
- 4) Les versions par défaut du constructeur par copie et de l'opérateur d'affectation ne sont pas adaptées aux classes pratiquant la télépossession. Pour ces classes, il faut IMPERATIVEMENT définir explicitement une version adaptée du constructeur par copie et surcharger l'opérateur d'affectation.
- 5) L'opérateur `sizeof` ne donne que la taille réelle d'une instance, sans prendre en compte ses éventuelles télépossessions.
- 6) Les opérateurs de comparaison, d'insertion et d'extraction doivent généralement être conçus pour opérer "profondément" sur les objets pratiquant la télépossession.