



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

Apprendre C++ avec Qt : Leçon 12

Allocation dynamique

1 - Notion de portée.....	2
2 - Notion de durée de stockage.....	2
3 - Variables locales statiques	3
Principe général.....	3
Le cas des fonctions membre.....	4
Variables membre statiques vs. variables locales statiques d'une fonction membre	5
4 - Allocation dynamique.....	5
Réserver de la mémoire avec new	5
Initialiser une zone mémoire réservée avec new.....	6
Libérer la mémoire avec delete	6
Les fuites de mémoire.....	8
A quoi sert réellement l'allocation dynamique ?	8
5 - Bon, c'est gentil tout ça, mais ça fait déjà 7 pages. Qu'est-ce que je dois vraiment en retenir ?	9

Les objets que nous avons manipulés jusqu'à présent sont des variables locales au bloc à l'intérieur duquel elles sont définies. Avant d'introduire la possibilité de créer des objets d'un autre type, prenons quelques instants pour préciser deux notions qui vont s'avérer importantes pour bien comprendre en quoi ces nouveaux objets diffèrent de ceux que nous connaissons déjà.

1 - Notion de portée

On appelle portée d'une variable (ou d'une fonction) la portion du programme dans laquelle il est possible d'accéder à la variable (ou à la fonction) en utilisant son nom.

Le terme anglais désignant la portée est *scope*.

Il faut remarquer que le fait que le nom d'une variable ne permette pas d'accéder à celle-ci n'implique pas nécessairement que la variable ait cessé d'exister. Une simple homonymie entre variables permet de créer une situation illustrant ce fait. Dans l'exemple suivant, nous avons simplement deux blocs de code, dont l'un est inclus dans l'autre. Nous savons que, dans ce cas, les variables locales du bloc externe sont disponibles à l'intérieur du bloc interne.

Oui, nous le savons. Comment pourrait-on utiliser les structures de contrôle (les `for`, `while` et autres `if`), si les blocs dont ces structures contrôlent l'exécution ne pouvaient accéder qu'à leurs propres variables locales ?

```
1 {  
2 int test = 4;      //définition d'une première variable nommée test  
3 {  
4 test = 5;         //changement de la valeur contenue dans la première variable  
5 int test = 3;      //définition d'une seconde variable nommée test  
6 //il est devenu impossible d'accéder à la variable qui contient 5  
7 } //la variable qui contient 3 cesse d'exister ici  
8 //il est à nouveau possible d'accéder à la variable qui contient 5  
9 }
```

Ce qui rend cette situation particulière, c'est que les blocs possèdent tous les deux une variable locale nommée `test`. Une fois définie, la **variable du bloc interne** masque la **variable homonyme** définie dans le bloc externe, ce qui signifie que celle-ci se retrouve hors de portée (on ne peut plus y accéder en utilisant son nom). Dès l'accolade marquant la fin du bloc interne (7), la variable locale de celui-ci cesse d'exister. Le phénomène de masquage disparaît donc, et la **variable locale du bloc externe** redevient accessible. Elle a même conservé son contenu, ce qui prouve bien qu'elle n'a jamais cessé d'exister.

Le fragment de code présenté ci-dessus n'est destiné qu'à mettre en évidence la différence entre portée et existence. Il ne constitue certainement pas un exemple stylistique qu'il serait judicieux de suivre. En effet, s'il est parfois légitime de définir une variable qui en masque une autre, il serait de très mauvais goût de le faire dans un bloc qui manipule également la variable masquée. Si un même nom peut éventuellement désigner des choses différentes à différents moments, il est préférable de conserver une correspondance entre "moments" et "blocs de code", faute de quoi la lisibilité du programme risque d'être compromise.

2 - Notion de durée de stockage

Lorsqu'on dit qu'une variable "cesse d'exister", cela signifie que la zone de mémoire dont l'état représentait le contenu de la variable cesse d'être réservée à cet usage. Toutes les variables cessent évidemment d'exister lorsque l'exécution du programme s'achève, et nous savons aussi que les variables locales cessent d'exister lorsque s'achève l'exécution du bloc de code à l'intérieur duquel elles sont définies.

Lorsque la variable qui cesse d'exister est une instance d'une classe, ses variables membre cessent également d'exister (elles ne sont que des sous-variables de l'instance).

Le fait qu'une variable cesse d'exister n'a pas forcément pour conséquence une modification immédiate de l'état de la zone de mémoire qui lui était réservée, mais plus rien ne garantit que cet état ne variera que d'une façon cohérente avec la signification qu'avait la variable. Il devient donc extrêmement périlleux d'interpréter cet état comme s'il correspondait encore au contenu de la variable, et le langage, très logiquement, interdit alors d'utiliser le nom de la variable.

Remarquons tout de suite que, si l'on accède à la zone de mémoire qui correspondait à la variable en utilisant un **pointeur** contenant son adresse, le langage N'EST PAS en mesure de proposer une sécurité analogue à celle offerte par l'interdiction de l'usage du nom de la variable. Cette situation est illustrée dans l'exemple suivant :

```
1 {  
2 int uneVariable;  
3 int *pointeur = & uneVariable;  
4 *pointeur = 5; //OK, on range 5 dans uneVariable, en fait  
5 {  
6 int uneVariableLocale;  
7 pointeur = & uneVariableLocale;  
8 *pointeur = 5; //OK, on range 5 dans uneVariableLocale, en fait  
9 } //uneVariableLocale cesse d'exister !  
10 *pointeur = 5; //DANGER : on modifie une zone de mémoire dont on ignore tout  
11 }
```

Ce danger est insidieux, car il est fort possible que le programme ne présente aucun symptôme immédiat. Tant que la zone de mémoire qui était attribuée à `uneVariableLocale` n'est pas affectée à un autre usage, tout semble en effet se passer normalement. Le problème peut n'apparaître que bien plus tard au cours du développement du programme et, qui plus est, il risque fort de se manifester par l'apparition de valeurs fantaisistes dans une variable qui n'a aucun rapport avec la ligne de code qui contient l'erreur. Les erreurs de ce type sont donc très difficiles à détecter et il faut à tout prix essayer d'éviter de les commettre.

Lorsqu'une variable cesse d'exister, tous les pointeurs qui contiennent son adresse cessent d'être valides et constituent autant de menaces graves pour l'intégrité du programme.

En réalité, les vrais problèmes n'apparaissent que dans des situations bien plus complexes que celle illustrée ci-dessus. Quoi qu'en disent certains esprits chagrins, tant que vous comprenez réellement ce que vous faites, l'usage de pointeurs n'est pas plus dangereux que n'importe quelle autre technique de programmation.

Et, si vous ne comprenez pas réellement ce que vous faites, toutes les techniques de programmation s'avèrent finalement trop dangereuses.

3 - Variables locales statiques

Une variable locale "ordinaire" cesse d'exister à la fin de l'exécution du bloc dans lequel elle est définie. Sa durée de stockage correspond donc exactement à sa portée : elle naît au moment de l'exécution de l'instruction qui la définit, meurt à la fin du bloc, et seules les instructions figurant entre ces deux points peuvent y accéder en utilisant son nom.

Si un objet portant un nom identique est défini ailleurs, il ne s'agit du même nom que d'un point de vue purement orthographique. Du point de vue de C++, ces deux noms sont aussi différents que s'ils n'avaient pas une seule lettre en commun.

Chaque fois qu'un bloc est exécuté, ses variables locales sont recrées et, éventuellement, ré-initialisées. Dans le cas d'une fonction, il arrive que cette caractéristique des variables locales ne permette pas d'obtenir l'effet recherché.

Principe général

Imaginons une fonction qui a, pour une raison quelconque, besoin de tenir compte du nombre de fois où elle a déjà été appelée. Une variable locale ordinaire ne peut servir à tenir ce compte, puisque son contenu est perdu à la fin de chacune des exécutions de la fonction. D'un autre côté, si cette fonction est appelée à partir de plusieurs parties différentes du programme, il peut s'avérer assez complexe de lui passer, lors de chaque appel, l'adresse d'une même variable dans laquelle elle pourrait tenir à jour le dénombrement des appels.

La difficulté provient du fait que les différentes parties du programme n'ont pas forcément, elles-mêmes, la possibilité de toutes accéder à une même variable. Cette approche aurait, en outre, l'inconvénient de créer une variable dont la portée dépasserait largement la zone du programme où elle est effectivement utile, qui est limitée à la fonction. Un tel dépassement est à éviter, car il augmente le risque que la variable soit modifiée par un programmeur n'ayant pas parfaitement conscience de sa signification et de la façon dont elle doit être manipulée.

Il existe un moyen simple pour résoudre ce problème : créer des variables locales dont la durée de stockage n'est pas limitée à l'exécution de la fonction. Pour créer ce type de variables, il suffit, lors de leur définition, de faire précéder leur type du mot `static`.

Les variables locales `static` naissent (et sont initialisées) lors de la première exécution du bloc où elles sont définies. Elles ont la portée habituelle mais ne meurent qu'à la fin du programme.

Le fragment de code suivant définit une fonction dont les appels successifs produisent progressivement la suite des nombres positifs impairs :

```
1 int generateurDImpairs()
2 {
3     static int impairSuivant = -1; //définition et initialisation
4     impairSuivant = impairSuivant + 2;
5     return impairSuivant;
6 }
```

Remarquez qu'une telle fonction ne pourrait pas être écrite s'il n'existait pas un processus d'initialisation distinct de celui de l'affectation. Remarquez également que, à l'usage, ce genre de fonction peut réserver quelques surprises :

```
int n;
for (n=0 ; n < 5 ; ++n)
    maFonction(generateurDImpairs());
```

Ce fragment de code semble invoquer 5 fois `maFonction()`, en lui passant successivement les valeurs 1, 3, 5, 7 et 9. Mais que se passe-t-il si quelqu'un qui n'est pas conscient de ce contexte d'utilisation introduit un appel à `generateurDImpairs()` dans le corps de `maFonction()` ?

Le cas des fonctions membre

L'utilisation de variables statiques locales à une fonction membre nécessite d'être conscient du fait qu'il n'existe pas un "exemplaire" de chaque fonction membre pour chaque instance de la classe. En clair, quelle que soit l'instance utilisée pour appeler la fonction membre, c'est le même jeu de variables locales statiques qui sera utilisé.

Dans bien des cas, l'usage d'une variable membre s'avère plus judicieux que celui d'une variable statique locale à une fonction membre, car il permet à chacune des instances de maintenir des valeurs différentes.

Une classe "fonctionoïde" dotée d'une variable membre permet, par exemple, de traiter correctement le problème de notre générateur de nombres impairs :

```
class generateurDImpairs
{
public:
    generateurDImpairs() : m_valeur(-1) {}
    int operator ()() {return m_valeur += 2; }
protected:
    int m_valeur;
};
```

L'usage de cette classe garantit que, quoi qu'entreprenne la fonction appelée, elle ne peut en aucun cas interférer avec le générateur utilisé par la fonction appelante (puisque'elle n'y a pas accès). Nous pouvons donc écrire en toute sécurité :

```
generateurDImpairs impairSuivant;
int n;
for (n=0 ; n < 5 ; ++n)
    maFonction(impairSuivant());
```

Il arrive que l'unicité des variables locales statiques d'une fonction membre ne pose pas problème, soit parce que c'est précisément l'effet souhaité, soit parce que la classe n'est pas destinée à être instanciée plusieurs fois simultanément¹ (ce qui est, par exemple, le cas de la plupart des classes décrivant un dialogue avec l'utilisateur...). Le recours à des variables locales statiques présente alors le double avantage de limiter le nombre de variables membre et de ne pas rendre l'information accessible aux autres fonctions membre.

¹ Si une classe ne supporte pas d'être instanciée plusieurs fois simultanément, il est de bonne politique de rendre ceci impossible (en pratiquant, par exemple, un comptage d'occurrences tel que celui décrit dans l'Annexe 3).

Variables membre statiques vs. variables locales statiques d'une fonction membre

Les lecteurs qui ont parcouru l'Annexe 3 savent que le mot `static` peut également qualifier des variables membres, ce qui peut prêter à confusion. En résumé :

Si une classe possède une **variable membre statique**, toutes les fonctions membre y ont accès (puisqu'il s'agit d'une variable membre) et elles accéderont toujours à la même variable, quelle que soit l'instance au titre de laquelle elles sont exécutées (puisque la variable est statique).

Si une fonction membre possède une **variable locale statique**, elle seule y aura accès (puisqu'il s'agit d'une variable locale) et elle accèdera toujours à la même variable, quelle que soit l'instance au titre de laquelle elle est exécutée.

4 - Allocation dynamique

Rendre une variable locale statique est donc un moyen d'augmenter sa durée de stockage, sans modifier pour autant sa portée. Ce moyen reste toutefois assez rudimentaire, car il n'offre de contrôle précis ni sur le moment de la naissance de la variable (c'est forcément lors de la première exécution du bloc) ni sur le moment de sa disparition (une variable statique ne disparaît qu'à la fin de l'exécution du programme). Lorsqu'une meilleure précision dans le contrôle de la durée de stockage s'avère nécessaire, c'est au processus d'allocation dynamique de la mémoire qu'il va nous falloir avoir recours.

La première chose qu'il faut bien comprendre à propos de l'allocation dynamique est que les objets ainsi créés restent anonymes.

Selon la définition que nous avons adoptée pour ce mot, ce ne sont donc pas des variables.

En l'absence de nom permettant de les désigner, ces objets ne peuvent donc être manipulés qu'à l'aide de pointeurs contenant leur adresse ou de références leur étant associées.

Réserver de la mémoire avec `new`

Jusqu'à présent, la seule méthode dont nous disposions pour rendre un pointeur valide était d'utiliser l'opérateur "adresse de" pour obtenir l'adresse d'une variable du type adéquat :

```
1 double uneVar = 3.14;  
2 double * pDouble = &uneVar;
```

L'allocation dynamique fournit un moyen permettant de réserver une zone de mémoire d'une taille correspondant au type de donnée que l'on souhaite y stocker.

Il n'y a aucune restriction sur le type de données pour lesquelles `new` peut réserver une zone de stockage. Les types natifs ne bénéficient d'aucun privilège de ce point de vue par rapport aux types que vous avez créés vous-même (en définissant une classe, par exemple).

L'opérateur qui effectue cette réservation est noté `new`, et il faut, bien entendu, lui préciser le type de donnée qui doit pouvoir être stocké dans l'emplacement réclamé. Si une zone de mémoire de la taille nécessaire est disponible, l'opérateur `new` la marque comme étant réservée et produit comme résultat l'adresse de cette zone. Si la réservation de mémoire échoue (lorsque, par exemple, toute la mémoire de l'ordinateur qui exécute le programme est déjà utilisée), l'opérateur `new` donne un résultat `NULL`.

C'est en tout cas la méthode qu'utilisait "traditionnellement" `new` pour signaler le problème. Si vous disposez d'un compilateur "moderne", il est possible qu'il utilise par défaut le mécanisme des exceptions (cf. Leçon 23). Vous pouvez alors lui demander explicitement de s'en tenir à sa première méthode en utilisant `new(nothrow)` en lieu et place de `new`.

Il convient donc, avant tout usage d'une adresse obtenue grâce à `new`, de vérifier que cette adresse est valide. Si ce n'est pas le cas, le programme doit adopter un comportement adapté.

Dans le cas de programmes simples, l'échec d'une demande d'allocation de mémoire doit, au minimum, conduire à un arrêt de l'exécution du programme, précédé d'un message avertissant l'utilisateur de la nature du problème. La façon correcte de mettre fin à l'exécution du programme ne peut pas être décrite ici, car elle dépend du système utilisé. Dans le cadre de projets plus ambitieux, il est parfois possible de mettre en place des stratégies palliatives, pour éviter d'avoir à abandonner prématurément l'exécution du programme.

Le fragment de code suivant propose un exemple de mise en œuvre de ce système :

```
1 double * pDouble = NULL;
2 pDouble = new double;
3 if (pDouble != NULL)
4 {
5     //on peut utiliser *pDouble comme s'il s'agissait du
    *pDouble = 3.14; //nom d'une variable de type double
6 }
7 else
8     //il faut prendre des mesures énergiques !
```

Dans cet exemple, une variable de type "pointeur sur double" est tout d'abord créée (1). Cette variable est initialisée avec la constante NULL, qui indique clairement que la variable ne contient pas, pour l'instant, l'adresse d'un double (et qu'il faut donc se garder de toute tentative de déréférencement).

Dans un second temps, l'opérateur new est utilisé pour réclamer l'adresse d'une zone mémoire susceptible de stocker une valeur de type double. L'adresse produite par new est rangée dans la variable pDouble (2).

L'opérateur new produit une adresse. C'est donc bien dans une variable de type pointeur qu'il convient de stocker ce résultat. De toutes façons, essayer de déréférencer pDouble avant de lui avoir donné un contenu valide ne pourrait conduire qu'à une catastrophe. Il serait donc doublement mal venu d'écrire :

```
double * pDouble = NULL;
* pDouble = new double; //une erreur grossière !
```

En effet, **déréférencer** le pointeur nous conduit à accéder à la zone de mémoire qu'il désigne. Or pDouble est de type "pointeur sur double", ce qui signifie que la zone qu'il désigne est destinée à stocker une valeur décimale, et non une adresse. De plus, comme pDouble n'est pas valide, la valeur produite par new serait copiée en mémoire à un endroit imprévisible et vraisemblablement utilisé à d'autres fins.

La suite du programme dépend de la valeur de pDouble. Si new a effectivement renvoyé une adresse valide, nous disposons d'une zone de mémoire qui peut contenir une valeur de type double et à laquelle nous accédons (5) en déréférençant pDouble. Dans le cas contraire, il ne faut surtout pas essayer de déréférencer pDouble !

Il est également possible (mais moins habituel) d'utiliser une **référence** pour désigner l'objet créé dynamiquement :

```
double & unDouble = *new double;
if (&unDouble != NULL)
{
    //on peut utiliser unDouble comme s'il s'agissait du
    unDouble = 3.14; //nom d'une variable de type double
}
else
    //il faut prendre des mesures énergiques !
```

Remarquez que, pour "initialiser" la référence, il faut bien entendu **déréférencer** l'adresse renvoyée par new. Inversement, pour tester la validité de la référence, c'est **l'adresse de** l'objet qu'elle désigne qui doit être vérifiée. En contrepartie de ces efforts, nous obtenons la possibilité de **désigner l'objet créé** sans avoir à utiliser le moindre opérateur.

Initialiser une zone mémoire réservée avec new

La demande de réservation d'une zone mémoire peut être assortie d'une demande d'initialisation de cette zone. La syntaxe employée dans ce cas est l'une de celles possibles pour initialiser les variables : il suffit de mentionner entre parenthèses la **valeur souhaitée** :

```
1 int *pEntier = NULL;
2 pEntier = new int(12); //initialisation de la zone avec la valeur 12
```

Libérer la mémoire avec delete

Si elle n'est pas explicitement libérée, la mémoire réservée à l'aide de l'opérateur new reste réservée jusqu'à la fin de l'exécution du programme. L'opérateur qui permet cette libération se nomme delete, et il faut, bien entendu, lui indiquer quel est le bloc de mémoire qui doit être

remis à la disposition du système. Assez naturellement, on désigne le bloc de mémoire à libérer en indiquant son adresse. Si l'on reprend l'exemple précédent, la libération de la mémoire se présenterait de la façon suivante :

```

1 double * pDouble = NULL;
2 pDouble = new double;
3 if (pDouble != NULL)
4 {
5     *pDouble = 3.14; //on peut utiliser *pDouble comme s'il s'agissait du nom
6                       //d'une variable de type double.
7     delete pDouble; //Lorsqu'on n'en a plus besoin, on libère la mémoire.
8 }
9 else
10    //il faut prendre des mesures énergiques !

```

La notation employée ne doit pas vous conduire à un contresens concernant l'effet de

```
delete pDouble;
```

Contrairement à ce qu'on pourrait croire, l'exécution de cette instruction n'a aucun effet sur `pDouble` (qui continue à exister), ni même sur le contenu de cette variable (qui peut éventuellement continuer à être l'adresse du bloc de mémoire qui vient d'être libéré). Le seul aspect de la variable `pDouble` qui soit certainement affecté par l'exécution de l'instruction en question est sa validité en tant que pointeur. Après cette instruction, il devient évidemment dangereux de déréférencer `pDouble`, puisque ceci revient à accéder à une zone de mémoire dont nous ignorons désormais quel est l'usage qui en est fait par le système.

Cette situation est très proche de celle signalée page 3, à propos du pointeur contenant l'adresse d'une variable locale à un bloc venant de se refermer. La disparition de la variable locale libère la mémoire qui lui était allouée, exactement comme `delete` libère le bloc de mémoire dont l'adresse lui est passée. Dans les deux cas, il nous reste un pointeur contenant l'adresse d'une zone de mémoire dont l'usage nous est désormais inconnu, c'est à dire un pointeur invalide.

Etant donné que `delete` a besoin de l'adresse du bloc à libérer, cet opérateur ne peut être utilisé que dans la mesure où il existe une variable de type pointeur contenant l'adresse en question. Il n'est pas pour autant nécessaire que le pointeur utilisé à cette occasion soit le même que celui ayant initialement reçu l'adresse produite par `new`. Il arrive assez fréquemment que ce ne soit pas le cas, comme dans l'exemple suivant, où une fonction utilisatrice() sous-traite à `alloueInt()` la tâche consistant à allouer la mémoire et à mettre fin au programme en cas d'échec de l'allocation.

```

1 int * alloueInt()
2 {
3     int * pEntier = new int;
4     if (pEntier == NULL)
5     {
6         //ici doivent figurer des instructions destinées à mettre fin au programme
7     }
8     return pEntier;
9 }
10 //*****
11 void utilisatrice()
12 {
13     //appels de la fonction sous-traitante
14     int * unPointeur = alloueInt();
15     int * unAutre = alloueInt();
16     //ici peut figurer du code utilisant *unPointeur et *unAutre
17     //libération de la mémoire
18     delete unPointeur;
19     delete unAutre;
20 }

```

Dans cet exemple, la seule utilisation de `new` figure (3) dans la fonction `alloueInt()`, et elle réserve une nouvelle zone de mémoire à chaque appel de la fonction. Ces zones mémoire ne sont pas destinées à être utilisées par `alloueInt()`, mais par la fonction qui l'appelle. La fonction `alloueInt()` s'achève donc (7) en renvoyant comme résultat l'adresse du bloc qui

vient d'être réservé (sauf, bien entendu, si l'allocation échoue).

Remarquez bien que la fonction retourne la valeur du pointeur, et non la valeur contenue dans la zone de mémoire dont l'adresse est dans ce pointeur. L'instruction `return` est donc appliquée simplement sur la variable `pEntier`, sans opération de déréférencement.

L'adresse renvoyée par `alloueInt()` est stockée par la fonction utilisatrice() dans l'une de ses variables locales (11-12). Lorsque la fonction utilisatrice s'achève, elle libère tous les blocs de mémoire alloués dynamiquement en appliquant l'opérateur `delete` à chacun de ses pointeurs (15-16). L'équilibre entre allocation et libération est donc obtenu ici avec un seul `new` (3) contre deux `delete` (15 et 16), chacune de ces trois instructions impliquant un pointeur différent (`pEntier`, `unPointeur` et `unAutre`).

Le fait qu'appliquer `delete` à un pointeur affecte la disponibilité de la zone de mémoire désignée et non le pointeur lui-même ne signifie pas seulement que la zone de mémoire peut être libérée en la désignant à l'aide d'un autre pointeur que celui ayant recueilli son adresse lors de l'allocation. Il signifie aussi que, si plusieurs pointeurs contiennent la même adresse, un seul `delete` (portant sur n'importe lequel d'entre eux) les rendra tous invalides.

Lorsqu'un bloc de mémoire est libéré à l'aide de `delete`, TOUS les pointeurs qui contiennent son adresse deviennent invalides.

Les fuites de mémoire

Si un programme procède à de nombreuses allocations dynamiques sans prendre soin de libérer les zones de mémoire devenues inutiles, la quantité de mémoire disponible finit fatalement par être épuisée, ce qui se traduit par l'échec d'une des demandes d'allocation.

Même si l'usage que fait votre programme de la mémoire est dérisoire vis à vis de la capacité mémoire totale de la machine que vous utilisez, laisser la fin du programme libérer pour vous la mémoire que vous avez réquisitionnée à l'aide de `new` est une pratique de sauvagerie.

Que va-t-il se passer si votre programme est exécuté sur un ordinateur moins richement doté en mémoire ? Et si de très nombreux programmes différents sont simultanément exécutés ? Et si plusieurs exemplaires de votre programme sont exécutés en même temps ? Et si quelqu'un d'autre (ou vous-même, dans six mois) essaie de généraliser votre programme et lui donne une ampleur que vous n'aviez pas prévue ? Même si rien de tout cela ne se produit, il est clair que, dans un programme bien conçu, il est assez facile de libérer proprement tous les blocs alloués avec `new`. Si votre programme ne le fait pas, cela en dit long sur la qualité générale de sa conception et suggère que d'autres erreurs pourraient bien y être tapies.

A chaque allocation d'un bloc de mémoire à l'aide de l'opérateur `new` doit correspondre une libération du bloc en question à l'aide de l'opérateur `delete`.

Lorsqu'un programme ne respecte pas cette règle, on dit qu'il présente une "fuite de mémoire" (*memory leak*, en anglais), et il s'agit d'un indice à peu près aussi rassurant qu'une flaque d'huile sous la voiture d'occasion que vous vous apprêtez à acheter.

A quoi sert réellement l'allocation dynamique ?

D'une façon générale, l'allocation dynamique s'avère indispensable dès qu'un nombre important ou imprévisible de données doivent être manipulées.

La plupart du temps, vous n'utiliserez pas directement les opérateurs `new` et `delete` mais des classes conteneur (`QValueList` ou `QMap`, par exemple) qui assurent à moindre frais une gestion correcte de l'allocation dynamique.

Il reste néanmoins deux cas où le recours explicite à l'allocation dynamique s'imposera :

- lorsque certains widgets proposés à l'utilisateur seront mis en place par le programme lui-même, au cours de son exécution (et non prépositionnés à l'aide de Qt Designer).

Les widgets générés dynamiquement sont habituellement confiés à un objet Qt préexistant, qui en prend la responsabilité : la destruction du widget est alors assurée automatiquement en temps voulu, et aucun appel explicite à `delete` ne doit être fait.

- lorsque vous utiliserez des structures de données autres que celles prises en charge par les conteneurs Qt (cf. Leçons 17, 18, 19, 20 et 21) ;

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

- 1) La portée d'une variable est la portion du programme où le nom de la variable peut être utilisé.
- 2) En général, la durée de stockage d'une variable correspond à sa portée : de la définition de la variable jusqu'à la fin du bloc où cette définition figure.
- 3) Lorsque, à l'intérieur de la portée d'une variable, on définit une variable homonyme, cette nouvelle variable masque la première (qui se trouve donc hors de portée, jusqu'à la disparition de la seconde variable).
- 4) Rendre une variable statique prolonge sa durée de stockage jusqu'à la fin de l'exécution du programme, sans modifier sa portée.
- 5) On peut obtenir une zone de stockage convenant à une donnée d'un type quelconque en utilisant l'opérateur `new`.
- 6) L'opérateur `new` produit l'adresse de la zone de mémoire réservée, et cette adresse peut être stockée dans un pointeur du type adéquat.
- 7) Si `new` ne parvient pas à réserver la zone de mémoire demandée, l'adresse qu'il produit est `NULL`.
- 8) Les blocs de mémoire réservés à l'aide de `new` doivent être libérés à l'aide de `delete`.
- 9) Déréférencer un pointeur contenant l'adresse d'une zone de mémoire qui n'est pas (ou plus) consacrée au stockage d'une donnée d'un type correspondant à celui du pointeur est rarement une bonne idée.