Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 17
Membres statiques

Normalement, les membres d'une classe n'ont d'existence que dans le contexte d'une instance de cette classe. Ainsi, si une classe `CExemple` comporte une variable de type `int` nommée `m_entier` et une fonction membre nommée `laFonction()`, il nous faut disposer d'une instance de `CExemple` pour disposer d'une sous-variable `m_entier` susceptible de contenir une valeur et pour pouvoir appeler `laFonction()`. Il existe toutefois des membres pour lesquels cette règle générale ne s'applique pas, ce qui leur confère, comme nous allons le voir, des propriétés tout à fait particulières.

1 - Variables membre statiques

Une variable membre statique est une variable qui existe toujours en un seul exemplaire, quel que soit le nombre d'instances de la classe. Cette caractéristique peut être exploitée de deux façons différentes : on peut considérer qu'il s'agit d'une variable *commune* à toutes les instances, qui permet donc à celles-ci de communiquer entre elles, ou on peut considérer qu'il s'agit d'une variable appartenant à la classe elle-même, et sans véritable lien avec les instances[1]. Quoi qu'il en soit, l'existence des variables membre statiques n'est pas liée à l'instanciation de la classe, ce qui complique légèrement leur création en exigeant une dissociation de leur déclaration et de leur définition.

Déclaration

Les variables membre statiques sont déclarées, comme les autres, dans la définition de la classe. Leur seule singularité est ici la présence du mot `static` qui indique leur statut.

```
1  class CAvecStatiques //définition DE LA CLASSE
2  {
3  public:
4     int m_entier;                    //DECLARATION d'un membre "ordinaire"
5     static int m_entierStatique;     //DECLARATION d'un membre statique
6  };
```

Dans la plupart des cas, la définition d'un type (et, en particulier, d'une classe) est faite dans un fichier .h, qui fera l'objet d'une directive `#include` dans tous les fichiers où la connaissance de cette définition pourrait être rendue nécessaire par le fait qu'un objet du type en question y est mentionné.

Définition

La définition d'un type (une classe, en l'occurrence) ne crée aucune variable. Les variables membre "ordinaires" ne sont effectivement crées que lorsque la classe est instanciée. En d'autres termes, leur définition est implicite, car elle est "contenue" dans la définition d'une instance de la classe. Il ne peut en aller de même dans le cas des variables membre statiques, puisque leur création ne doit pas être tributaire de l'instanciation (et ne doit pas être réitérée en cas d'instanciations multiples). Il est donc nécessaire de créer les variables membre statiques en les définissant explicitement.

Comme nous le savons, si un même objet peut être redéclaré sans inconvénients[2], il n'est en revanche pas question de *redéfinir* un objet qui existe déjà. Le fait que les fichiers .h sont amenés à faire l'objet de directives `#include` dans de multiples fichiers relevant du même projet interdit donc d'y faire figurer des définitions. Par conséquent,

Le lieu naturel de la définition des variables membre statiques est le fichier .cpp qui contient la définition des fonctions membre, et non le fichier .h définissant la classe elle-même.

La définition d'une variable membre statique exige la mention de son nom complet, mais le mot `static` ne doit pas être rappelé à cette occasion. Dans le cas de notre exemple, cette définition prendrait donc la forme suivante :

```
int CAvecStatiques::m_entierStatique; //définition sans mention du mot static
```

[1] C'est cette interprétation que suggère le langage Smalltalk, dans lequel l'équivalent des variables membre statiques s'appelle, justement, des "*variables de classe*", par opposition aux autres, qui sont des "*variables d'instance*".
[2] A la seule condition que toutes ces déclarations soient identiques.

Initialisation

La façon dont sont créées les variables membre statiques simplifie considérablement leur initialisation. Celle-ci s'effectue en effet sans intervention d'aucun constructeur (puisqu'elle doit être indépendante de l'instanciation de la classe), ce qui permet en particulier d'attribuer une valeur à une constante sans devoir recourir à la liste d'initialisations d'un constructeur. Si notre classe est complétée ainsi :

```
1  class CAvecStatiques
2  {
3  public:
4     int m_entier;
5     static int m_entierStatique;
6     static const int m_laConstanteStatique;
7  };
```

l'initialisation des membres statiques s'effectue sans aucun problème :

```
1  int CAvecStatiques::m_entierStatique = 12;
2  const int CAvecStatiques::m_laConstanteStatique = 15;
```

Utilisation

Il existe plusieurs façons d'accéder à une variable membre statique. L'une de ces méthodes est d'utiliser une instance de la classe pour accéder à la variable membre, exactement comme s'il ne s'agissait pas d'une variable statique :

```
1  CAvecStatiques uneInstance;
2  uneInstance.m_entierStatique = 18;
```

Cette méthode est cependant un peu paradoxale, puisque c'est la même variable `m_entierStatique` qui est concernée, quelle que soit l'instance utilisée pour y accéder[3]. Le recours à une instance pour accéder à une variable qui n'a aucun lien particulier avec elle donne une apparence trompeuse à l'instruction, et un lecteur distrait risque fort de mal interpréter cette notation.

> Cette façon de procéder recèle même un danger encore plus pernicieux : l'expression utilisée à gauche du sélecteur de membre n'est qu'un moyen indirect de mentionner la classe concernée. Dans certains cas, l'évaluation de cette expression peut être incomplète, et ses éventuels effets peuvent donc ne pas être réalisés.

Dans le cas d'une fonction membre (non statique) de la classe `CAvecStatiques`, l'utilisation du nom d'une variable membre sans précision de l'instance concernée revient à utiliser implicitement l'instance au titre de laquelle la fonction est exécutée. Si une fonction membre de la classe `CAvecStatiques` comporte une ligne telle que :

```
m_entierStatique = 0;
```

nous avons vu (Leçon 14) que cette ligne est en fait équivalente à :

```
this->m_entierStatique = 0;
```

ce qui revient à accéder à la variable membre statique au titre d'une instance particulière de la classe. Même si cette façon de procéder est correcte, sa lisibilité est douteuse car le rappel du caractère statique de la variable n'est pas syntaxique mais dépend uniquement du choix judicieux du nom qui lui a été attribué.

Il est préférable de toujours désigner les variables membre statiques par leur nom complet, ce qui élimine tout risque de malentendu quant à la nature de ces variables :

```
CAvecStatiques::m_entierStatique = 18; //affectation d'une valeur
```

> Remarquez comme, dans cette notation, l'indépendance de la variable statique vis à vis de toute instance de la classe `CAvecStatique` saute immédiatement aux yeux...

[3] Et pour cause : il n'existe qu'une seule variable `m_entierStatique`, quel que soit le nombre d'instances !

Un exemple

L'exemple illustrant traditionnellement l'utilisation des variables membre statiques est celui du dénombrement (et, éventuellement, de l'immatriculation) des instances d'une classe[4]. Dans certaines situations (notamment lors d'opérations de débugage), il peut être intéressant de disposer d'un compteur indiquant en permanence combien d'instances d'une classe donnée sont "en vie", et d'un moyen d'identifier sans ambiguïté les instances disponibles[5]. Ce genre de mécanisme est très facile à mettre en place de façon parfaitement fiable : il suffit de veiller à ce que TOUS les constructeurs[6] de la classe concernée tiennent à jour les variables membre statiques servant au dénombrement et à l'immatriculation.

Dans sa version minimale, une classe pratiquant cet exercice doit comporter :
- deux variables membre statiques : l'une sert à dénombrer les instances existant à un moment donné (elle est initialisée à 0, elle est incrémentée par tous les constructeurs et elle est décrémentée par le destructeur), alors que l'autre sert à attribuer un numéro unique à chaque instance créée (elle est initialisée à zéro et est incrémentée par tous les constructeurs).
- une constante permettant à chaque instance de stocker le numéro d'identification qui lui est attribué lors de sa naissance.

> Remarquez le contraste entre les deux membres statiques (qui n'existent qu'en un seul exemplaire et dont les valeurs changent au cours de l'exécution du programme) et le membre constant (dont un exemplaire est créé pour chaque instance, avec une valeur chaque fois différente).

Par ailleurs, une telle classe doit définir explicitement au moins deux constructeurs (le constructeur par défaut et le constructeur par copie), ainsi que son destructeur. Nous obtenons donc la définition suivante :

```
1  class CSurveillee
2  {
3  public:
4     CSurveillee();                                  //constructeur par défaut
5     CSurveillee(const CSurveillee & modele);        //constructeur par copie
6     ~CSurveillee();                                 //destructeur
7  protected:
8     static int nbInstances;
9     static int numeroDeLaProchaineInstance;
10    const int m_numero;
11 };
```

Les membres statiques doivent faire l'objet d'une définition, qui fournit l'occasion de les initialiser :

```
//définition et initialisation des variables membre statiques
1  int CSurveillee::nbInstances = 0;
2  int CSurveillee::numeroDeLaProchaineInstance = 0;
```

Le constructeur par défaut doit utiliser sa liste d'initialisation pour donner une valeur au membre constant qui identifie l'instance créée. Bien entendu, ce constructeur doit aussi tenir à jour le compteur d'occurrences, et faire en sorte que la prochaine instance qui va être crée reçoive un numéro d'identification différent de celui qui vient d'être utilisé.

```
//constructeur par défaut
1  CSurveillee::CSurveillee()
                    : m_numero(CSurveillee::numeroDeLaProchaineInstance)
2  {
3  ++ CSurveillee::nbInstances;
4  ++ CSurveillee::numeroDeLaProchaineInstance;
5  }
```

---

[4] On rencontre, bien entendu, de nombreux autres usages des variables membre statiques. Cet exemple a surtout l'avantage de n'exiger qu'un contexte minimal pour devenir significatif.
[5] Cette identification peut permettre de retrouver quelle fonction est responsable de la création d'une instance particulière, par exemple.
[6] Attention à ne pas oublier le constructeur par copie éventuellement généré automatiquement par le compilateur...

Le constructeur par copie doit effectuer les mêmes opérations, mais il doit aussi procéder au report des valeurs des variables membre qui, dans un exemple complet, contiendraient l'information stockée dans une variable de type `CSurveillee`.

> Lors d'une construction par copie d'un objet de type `CSurveillee`, le membre constant `m_numero` ne doit, bien entendu, pas être initialisé avec la valeur qu'il a dans l'instance qui sert de modèle, ce que ferait la version du constructeur par copie générée automatiquement par le compilateur. Ce simple fait suffirait à imposer une définition explicite du constructeur par copie, si celle-ci n'était pas de toutes façons exigée du fait des modifications qui doivent être effectuées sur le contenu des variables statiques.
>
> La même remarque peut être faite à propos de l'opérateur d'affectation : s'il est défini, il ne doit pas transférer la valeur de la constante `m_numero` de l'instance modèle vers l'instance cible (ce qui serait de toutes façons impossible puisque, justement, il s'agit d'un membre *constant*). La classe `CSurveillee` ne définit pas cet opérateur et, la présence d'un membre constant en interdisant la génération automatique, elle restera donc dépourvue d'opérateur d'affectation.

```
//constructeur par copie
CSurveillee::CSurveillee(const CSurveillee &modele)
                  : m_numero(CSurveillee::numeroDeLaProchaineInstance)
{
++ CSurveillee::nbInstances;
++ CSurveillee::numeroDeLaProchaineInstance;
//dans un exemple complet, il faudrait aussi reporter les valeurs des
//différents membre du modele dans nos propres variables membre
}
```

La tâche incombant au destructeur est, par comparaison, très simple : il lui suffit de signaler la disparition d'une instance, et il peut profiter de l'occasion pour s'assurer que le nombre d'instances reste positif (si ce n'est pas le cas, le programme comporte une erreur : soit `CSurveillee::nbInstances` n'est pas initialisée correctement, soit l'un des constructeurs oublie de l'incrémenter).

```
//destructeur
CSurveillee::~CSurveillee()
{
-- CSurveillee::nbInstances;
assert (CSurveillee::nbInstances >= 0);
}
```

2 - Fonctions membre statiques

Les fonctions membre peuvent, elles aussi, être rendues statiques. Les conséquences de ce statut sont toutefois légèrement différentes de celles que nous venons d'observer à propos des variables membre. En effet, les fonctions membre "ordinaires" (ie. non statiques) n'existent pas en autant d'exemplaires qu'il y a d'instances de la classe[7] et doivent, de toutes façons, être définies explicitement. Les fonctions membre statiques ne peuvent donc se différencier des fonctions "ordinaires" ni de l'un ni de l'autre de ces points de vue.

L'originalité fondamentale d'une fonction membre statique réside dans le fait qu'il n'est pas nécessaire de disposer d'une instance pour appeler cette fonction. Comme dans le cas des variables membre, l'usage du nom complet d'une fonction membre permet de l'appeler, même en l'absence de toute instance de la classe concernée.

> La similitude avec le cas des variables membre statiques va même plus loin : il est également possible d'appeler une fonction membre statique en utilisant une instance de la classe et un opérateur de sélection. Comme dans le cas des variables membre statiques, cette façon de procéder présente le défaut de ne rien révéler du fait que le membre sélectionné est statique, et que l'instance utilisée ne joue ici qu'un rôle purement syntaxique.

[7] Souvenez-vous, à ce propos, d'une particularité des variables locales statiques appartenant à des fonctions membre que nous avons déjà eu l'occasion de souligner : si ces variables sont bien capables de retrouver, au début d'une exécution de la fonction, la valeur qu'elles avaient à la fin de l'exécution précédente, elles sont en revanche tout à fait incapables de stocker des valeurs différentes selon l'instance utilisée pour appeler la fonction. Ceci montre bien qu'il n'existe pas un exemplaire de chaque fonction membre (et, par conséquent, un exemplaire de chacune des variables locales de celles-ci) pour chaque instance de la classe.

Déclaration

La déclaration d'une fonction membre statique se singularise uniquement par la présence du mot `static` en tête de la déclaration. Nous pouvons donc compléter ainsi la définition de notre classe comportant des membres statiques :

```
1  class CAvecStatiques
2  {
3  public:
4     int m_entier;
5     static int m_entierStatique;
6     static const int m_laConstanteStatique;
7     static void fonctionMembreStatique();
8  };
```

Définition

Comme dans le cas de la définition d'une variable membre statique, le mot static ne doit pas figurer dans la définition d'une fonction membre statique.

> Le caractère statique d'une fonction membre est donc indiscernable à partir de la définition de cette fonction. Seul l'examen de la déclaration de la fonction le révèle.

Le fait qu'une fonction membre statique n'est pas invoquée au titre d'une instance présente bien entendu une conséquence majeure : les fonctions statiques ne disposent pas du paramètre caché `this`, et ne peuvent donc pas accéder implicitement aux variables membre d'une instance particulière (de quelle instance pourrait-il bien s'agir, d'ailleurs ?). Les traitements effectués par une fonction membre statiques sont donc limités aux variables membre statiques et aux objets auxquels leurs paramètres visibles leur donnent accès.

3 - Usages particuliers des variables membre statiques

La nature particulière des variables membre statiques permet d'en faire des usages qui ne seraient pas possibles dans le cas de variables membre "ordinaires".

Valeurs par défaut des paramètres de fonctions membre

La valeur par défaut d'un paramètre doit être indiquée lors de la déclaration de la fonction concernée, c'est à dire, dans le cas d'une fonction membre, lors de la définition de la classe. Lorsque cette valeur peut être donnée littéralement, l'opération ne présente guère de difficultés :

```
1  #include "uneClasse.h"
2  class CExemple_1
3  {
4  public:
5     void m_fonctionUn(int parametre = 12);
6     void m_fonctionDeux(CUneClasse parametre = CUneClasse(-12, "coucou",true));
7  }
```

> Dans cet exemple, on suppose que la classe `CUneClasse` (dont la définition figure dans le fichier uneClasse.h) comporte un constructeur attendant trois paramètres, constructeur dont l'appel explicite nous permet de simuler l'expression littérale d'une valeur de type `CUneClasse`.

Lorsque la valeur qui doit être utilisée par défaut ne peut pas être spécifiée littéralement[8], il devient nécessaire d'utiliser une variable. Il est alors indispensable que le nom de cette variable soit dans la portée de la déclaration de la fonction qui souhaite l'utiliser comme valeur par défaut pour l'un de ses paramètres.

Une première solution "envisageable" serait d'avoir recours à une variable globale. Cette approche présente toutefois deux inconvénients sérieux : la création d'une variable globale

[8] Ce qui est notamment le cas lorsque que le type du paramètre est une classe qui, à la différence de la classe `CUneClasse`, ne dispose pas d'un constructeur se prêtant à l'instanciation "à la volée".

n'est jamais une chose souhaitable, et l'utilisation d'une telle variable dans la définition d'une classe implique quelques complications syntaxiques.

> Pour être utilisable, la variable globale doit en effet être *déclarée* dans le fichier .h où se trouve la définition de la classe qui comporte une fonction membre dont l'un des paramètres prend par défaut la valeur de cette variable. Comme cette variable ne peut pas être *définie* dans un fichier .h (dont les inclusions répétées provoqueraient alors des erreurs de compilation), il faut faire précéder la mention du type et du nom de la variable du mot `extern`, ce qui permet de déclarer une variable sans la définir en même temps.

Une solution largement préférable est de donner à l'instance en question un statut de membre statique de la classe[9], et non de variable globale.

```
1  #include "test.h"
2  class CExemple_2
3  {
4  public:
5     void m_laFonction(CTest leParametre = CExemple_2::parDefaut);
6     static CTest parDefaut;
7  };
```

Il suffit ensuite de veiller à ce que la variable `CExemple_2::parDefaut` soit dans l'état souhaité au moment où `m_laFonction()` est appelée.

Variables membres statiques dont le type est la classe elle-même

Une variable non statique membre d'une classe ne peut avoir pour type la classe elle-même. En effet, si c'était le cas, la création d'une instance impliquerait la création d'une autre instance, qui elle-même impliquerait la création d'une troisième instance, et ainsi de suite, à l'infini...

Ce problème ne se pose pas dans le cas d'une variable membre statique, car la création d'une instance ne donne en aucun cas lieu à la création d'une variable correspondant à ce membre. De ce fait, une variable statique peut avoir pour type la classe elle-même, sans pour autant rendre impossible le processus d'instanciation. La définition suivante est donc admissible :

```
1  class CAvecInstanceMembre
2  {
3  public :
4     static CAvecInstanceMembre uneInstance;
5  };
```

Cette possibilité peut être combinée avec l'usage d'un membre statique comme valeur par défaut d'un paramètre d'une fonction membre. Il devient ainsi possible, par exemple, de doter le paramètre du constructeur par copie d'une valeur par défaut, ce qui attribue à ce constructeur un second rôle : celui de constructeur par défaut.

```
1  class CExemple_3
2  {
3  public:
      //constructeur par copie et par défaut
4     CExemple_3 (const CExemple_3 & modele = modeleParDefaut);
5  protected:
      //modèle par défaut pour le constructeur par copie
6     static CExemple_3 modeleParDefaut;
   //autre membres…
   };
```

> Dans ce genre de situation, il faut vraisemblablement éviter que le membre statique qui sert de modèle par défaut soit lui-même construit par le constructeur par défaut. Ceci conduirait en effet à utiliser comme modèle l'instance en cours de construction, ce qui a peu de chance de donner un résultat très intéressant. Cette façon de procéder n'a toutefois rien d'illégal, et il reste possible de modifier ensuite le membre statique, de façon à ce que les exécutions ultérieures du constructeur par défaut utilisent un modèle dont le contenu est cohérent.

[9] Un membre non statique ne conviendrait pas, car son existence supposerait que la classe soit instanciée, ce qui n'est évidemment pas possible au moment où l'on définit cette classe...

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

1) Les variables membre statiques sont des variables dont il n'existe pas un exemplaire dans chaque instance, mais un seul exemplaire qui peut être vu comme appartenant à la classe elle-même.
2) Les variables membre statiques sont déclarées (comme les autres membres) dans la définition de la classe.
3) A la différence des variables membre non statiques, les variables membre statiques doivent faire l'objet d'une définition explicite, ce qui offre une occasion de les initialiser facilement.
4) Il est préférable de toujours désigner les variables membre statiques par leur nom complet, même lorsque le compilateur les reconnaîtrait à partir de leur nom abrégé.
5) Les fonctions membre statiques peuvent être invoquées sans référence à une instance particulière.
6) Comme les fonctions membre statiques ne sont pas invoquées au titre d'une instance, elles n'ont aucun accès privilégié aux variables membre d'une instance particulière. Les seules variables membre sur lesquelles elles peuvent opérer sont donc les variables membre statiques (et, bien entendu, les variables membres d'objets locaux ou rendus accessibles par des paramètres).