Claude Delannoy

Programmer en langage C++

8e édition

Avec une intro aux design patterns
et une annexe sur la norme C++11

© Groupe Eyrolles, 1993-2011.
© Groupe Eyrolles, 2014, pour la nouvelle présentation, ISBN : 978-2-212-14008-8.

EYROLLES

11

Classes et objets

Avec ce chapitre, nous abordons véritablement les possibilités de P.O.O. de C++. Comme nous l'avons dit dans le premier chapitre, celles-ci reposent entièrement sur le concept de classe. Une classe est la généralisation de la notion de type défini par l'utilisateur[1], dans lequel se trouvent associées à la fois des données (membres données) et des méthodes (fonctions membres). En P.O.O. « pure », les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes. En C++, en revanche, vous pourrez n'encapsuler qu'une partie des données d'une classe (même si cette démarche reste généralement déconseillée). Vous pourrez même ajouter des méthodes au type structure (mot clé *struct*) que nous avons déjà rencontré ; dans ce cas, il n'existera aucune possibilité d'encapsulation. Ce type sera rarement employé sous cette forme généralisée mais comme, sur un plan conceptuel, il correspond à un cas particulier de la classe, nous l'étudierons tout d'abord, ce qui nous permettra dans un premier temps de nous limiter à la façon de mettre en œuvre l'association des données et des méthodes. Nous ne verrons qu'ensuite comment s'exprime l'encapsulation au sein d'une classe (mot clé *class*).

Comme une classe (ou une structure) n'est qu'un simple type défini par l'utilisateur, les objets possèdent les mêmes caractéristiques que les variables ordinaires, en particulier en ce qui concerne leurs différentes classes d'allocation (statique, automatique, dynamique). Cependant, pour rester simple et nous consacrer au concept de classe, nous ne considérerons dans ce chapitre que des objets automatiques (déclarés au sein d'une fonction quelconque),

1. Les types définis par l'utilisateur que nous avons rencontrés jusqu'ici sont : les structures, les unions et les énumérations.

ce qui correspond au cas le plus naturel. Ce n'est qu'au chapitre 13 que nous aborderons les autres classes d'allocation des objets.

Par ailleurs, nous introduirons ici les notions très importantes de constructeur et de destructeur (il n'y a guère d'objets intéressants qui n'y fassent pas appel). Là encore, compte tenu de la richesse de cette notion et de son interférence avec d'autres (comme les classes d'allocation), il vous faudra attendre la fin du chapitre 13 pour en connaître toutes les possibilités. Nous étudierons ensuite ce qu'on nomme les membres données statiques, ainsi que la manière de les intialiser. Enfin, ce premier des trois chapitres consacrés aux classes nous permettra de voir comment exploiter une classe en C++ en recourant aux possibilités de compilation séparée.

1 Les structures généralisées

Considérons une déclaration classique de structure telle que :

```
struct point
{ int x ;
  int y ;
}
```

C++ nous permet de lui associer des méthodes (fonctions membres). Supposons, par exemple, que nous souhaitions introduire trois fonctions :

- *initialise* pour attribuer des valeurs aux « coordonnées » d'un point ;
- *deplace* pour modifier les coordonnées d'un point ;
- *affiche* pour afficher un point : ici, nous nous contenterons, par souci de simplicité, d'afficher les coordonnées du point.

Voyons comment y parvenir, en distinguant la déclaration de ces fonctions membres de leur définition.

1.1 Déclaration des fonctions membres d'une structure

Voici comment nous pourrions *déclarer* notre structure *point* :

```
struct point
{              /* déclaration "classique" des données */
    int x ;
    int y ;
          /* déclaration des fonctions membre (méthodes) */
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Déclaration d'une structure comportant des méthodes

Outre la déclaration classique des champs de données apparaissent les déclarations (en-têtes) de nos trois fonctions. Notez bien que la définition de ces fonctions ne figure pas à ce niveau de simple déclaration : elle sera réalisée par ailleurs, comme nous le verrons un peu plus loin.

Ici, nous avons prévu que la fonction membre *initialise* recevra en arguments deux valeurs de type *int*. À ce niveau, rien n'indique l'usage qui sera fait de ces deux valeurs. Ici, bien entendu, nous avons écrit l'en-tête de *initialise* en ayant à l'esprit l'idée qu'elle affecterait aux membres *x* et *y* les valeurs reçues en arguments. Les mêmes remarques s'appliquent aux deux autres fonctions membres.

Vous vous attendiez peut-être à trouver, pour chaque fonction membre, un argument supplémentaire précisant la structure de type *point* sur laquelle elle doit opérer. Nous verrons comment cette information sera automatiquement fournie à la fonction membre lors de son appel.

1.2 Définition des fonctions membres d'une structure

Elle se fait par une définition (presque) classique de fonction. Voici ce que pourrait être la définition de *initialise* :

```
void point::initialise (int abs, int ord)
{ x = abs ;
  y = ord ;
}
```

Dans l'en-tête, le nom de la fonction est :

```
point::initialise
```

Le symbole :: correspond à ce que l'on nomme l'opérateur de « résolution de portée », lequel sert à modifier la portée d'un identificateur. Ici, il signifie que l'identificateur *initialise* concerné est celui défini dans *point*. En l'absence de ce « préfixe » (*point::*), nous définirions effectivement une fonction nommée *initialise*, mais celle-ci ne serait plus associée à *point* ; il s'agirait d'une fonction « ordinaire » nommée *initialise*, et non plus de la fonction membre *initialise* de la structure *point*.

Si nous examinons maintenant le corps de la fonction *initialise*, nous trouvons une affectation :

```
x = abs ;
```

Le symbole *abs* désigne, classiquement, la valeur reçue en premier argument. Mais *x*, quant à lui, n'est ni un argument ni une variable locale. En fait, *x* désigne le membre *x* correspondant au type *point* (cette association étant réalisée par le *point::* de l'en-tête). Quelle sera précisément la structure de type *point* concernée ? Là encore, nous verrons comment cette information sera transmise automatiquement à la fonction *initialise* lors de son appel.

Nous n'insistons pas sur la définition des deux autres fonctions membres ; vous trouverez ci-dessous l'ensemble des définitions des trois fonctions.

```
/* ----- Définition des fonctions membres du type point ---- */
#include <iostream>
using namespace std ;
```

```
void point::initialise (int abs, int ord)
{  x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{  x += dx ; y += dy ;
}
void point::affiche ()
{  cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Définition des fonctions membres

Les instructions ci-dessus ne peuvent pas être compilées seules. Elles nécessitent l'incorporation des instructions de déclaration correspondantes présentées au paragraphe 1.1. Celles-ci peuvent figurer dans le même fichier ou, mieux, faire l'objet d'un fichier en-tête séparé.

1.3 Utilisation d'une structure généralisée

Disposant du type *point* tel qu'il vient d'être déclaré au paragraphe 1.1 et défini au paragraphe 1.2, nous pouvons déclarer autant de structures de ce type que nous le souhaitons. Par exemple :

```
point a, b ;
```
[1]

déclare deux structures nommées *a* et *b*, chacune possédant des membres *x* et *y* et disposant des trois méthodes *initialise*, *deplace* et *affiche*. À ce propos, nous pouvons d'ores et déjà remarquer que si chaque structure dispose en propre de chacun de ses membres, il n'en va pas de même des fonctions membres : celles-ci ne sont générées[2] qu'une seule fois (le contraire conduirait manifestement à un gaspillage de mémoire !).

L'accès aux membres *x* et *y* de nos structures *a* et *b* pourrait se dérouler comme nous avons appris à le faire avec les structures usuelles ; ainsi pourrions-nous écrire :

```
a.x = 5 ;
```

Ce faisant, nous accéderions directement aux données, sans passer par l'intermédiaire des méthodes. Certes, nous ne respecterions pas le principe d'encapsulation, mais dans ce cas précis (de structure et pas encore de classe), ce serait accepté en C++[3].

On procède de la même façon pour l'appel d'une fonction membre. Ainsi :

```
a.initialise (5,2) ;
```

1. Ou *struct point a, b* ; le mot *struct* est facultatif en C++.

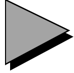
2. Exception faite des fonctions en ligne (les fonctions en ligne ordinaires ont déjà été présentées au paragraphe 14 du chapitre 7 ; les fonctions membres en ligne seront abordées au paragraphe 3 du chapitre 12).

3. Ici, justement, les fonctions membres prévues pour notre structure *point* permettent de respecter le principe d'encapsulation.

signifie : appeler la fonction membre *initialise* **pour la structure a**, en lui transmettant en arguments les valeurs 5 et 2. Si l'on fait abstraction du préfixe *a*., cet appel est analogue à un appel classique de fonction. Bien entendu, c'est justement ce préfixe qui va préciser à la fonction membre quelle est la structure sur laquelle elle doit opérer. Ainsi, l'instruction :

```
x = abs ;
```

de *point::initialise* placera dans le champ *x* de la structure *a* la valeur reçue pour *abs* (c'est-à-dire 5).



Remarques

1 Un appel tel que *a.initialise (5,2) ;* pourrait être remplacé par :

```
a.x = 5 ; a.y = 2 ;
```

Nous verrons précisément qu'il n'en ira plus de même dans le cas d'une (vraie) classe, pour peu qu'on y ait convenablement encapsulé les données.

2 En jargon P.O.O., on dit également que *a.initialise (5, 2)* constitue l'**envoi d'un message** (*initialise*, accompagné des informations 5 et 2) à l'objet *a*.

1.4 Exemple récapitulatif

Voici un programme reprenant la déclaration du type *point*, la définition de ses fonctions membres et un exemple d'utilisation dans la fonction *main* :

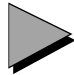
```
#include <iostream>
using namespace std ;
       /* ------------ Déclaration du type point ------------- */
struct point
{              /* déclaration "classique" des données */
    int x ;
    int y ;
          /* déclaration des fonctions membres (méthodes) */
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;

     /* ----- Définition des fonctions membres du type point ---- */
void point::initialise (int abs, int ord)
{   x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{   x += dx ; y += dy ;
}
void point::affiche ()
{   cout << "Je suis en " << x << " " << y << "\n" ;
}
```

```
int main()
{  point a, b ;
   a.initialise (5, 2) ; a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
   b.initialise (1,-1) ; b.affiche () ;
}
```

```
Je suis en 5 2
Je suis en 3 6
Je suis en 1 -1
```

Exemple de définition et d'utilisation du type point



Remarques

1 La syntaxe même de l'appel d'une fonction membre fait que celle-ci reçoit obligatoirement un argument implicite du type de la structure correspondante. Une fonction membre ne peut pas être appelée comme une fonction ordinaire. Par exemple, cette instruction :

```
initialise (3,1) ;
```

sera rejetée à la compilation (à moins qu'il n'existe, par ailleurs, une fonction ordinaire nommée *initialise*).

2 Dans la déclaration d'une structure, il est permis (mais généralement peu conseillé) d'introduire les données et les fonctions dans un ordre quelconque (nous avons systématiquement placé les données avant les fonctions).

3 Dans notre exemple de programme complet, nous avons introduit :

– la déclaration du type *point* ;

– la définition des fonctions membres ;

– la fonction (*main*) utilisant le type *point*.

Mais, bien entendu, il serait possible de *compiler séparément* le type *point* ; c'est d'ailleurs ainsi que l'on pourra « réutiliser » un composant logiciel. Nous y reviendrons au paragraphe 6.

4 Il reste possible de déclarer des structures généralisées anonymes, mais cela est très peu utilisé.

5 Seules les structures généralisées les plus simples peuvent être initialisées lors de leur déclaration par un initialiseur de la forme {....}. Par exemple, cela ne sera plus possible pour une structure munie d'un constructeur. En pratique, cette possibilité, essentiellement destinée à assurer une compatibilité avec le langage C, reste peu usitée (même si C++11 en élargit le champ).

2 Notion de classe

Comme nous l'avons déjà dit, en C++ la structure est un cas particulier de la classe. Plus précisément, une classe sera une structure dans laquelle seulement certains membres et/ou fonctions membres seront « publics », c'est-à-dire accessibles « de l'extérieur », les autres membres étant dits « privés ».

La déclaration d'une classe est voisine de celle d'une structure. En effet, il suffit :

- de remplacer le mot clé *struct* par le mot clé *class* ;
- de préciser quels sont les membres publics (fonctions ou données) et les membres privés en utilisant les mots clés *public* et *private*.

Par exemple, faisons de notre précédente structure *point* une classe dans laquelle tous les membres données sont privés, et toutes les fonctions membres sont publiques. Sa déclaration serait simplement la suivante :

```
/* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
  private :         /* facultatif (voir remarque 4) */
    int x ;
    int y ;
               /* déclaration des membres publics */
  public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Déclaration d'une classe

Ici, les membres nommés *x* et *y* sont privés, tandis que les fonctions membres nommées *initialise*, *deplace* et *affiche* sont publiques.

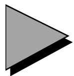
En ce qui concerne la définition des fonctions membres d'une classe, elle se fait exactement de la même manière que celle des fonctions membres d'une structure (qu'il s'agisse de fonctions publiques ou privées). En particulier, ces fonctions membres ont accès à l'ensemble des membres (publics ou privés) de la classe.

L'utilisation d'une classe se fait également comme celle d'une structure. À titre indicatif, voici ce que devient le programme du paragraphe 1.4 lorsque l'on remplace la structure *point* par la classe *point* telle que nous venons de la définir :

```
#include <iostream>
using namespace std ;
```

```
      /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
  private :
    int x ;
    int y ;
             /* déclaration des membres publics */
  public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
     /* ----- Définition des fonctions membres de la classe point ---- */
void point::initialise (int abs, int ord)
{ x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{ x = x + dx ; y = y + dy ;
}
void point::affiche ()
{ cout << "Je suis en " << x << " " << y << "\n" ;
}
        /* -------- Utilisation de la classe point -------- */
int main()
{  point a, b ;
   a.initialise (5, 2) ; a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
   b.initialise (1,-1) ; b.affiche () ;
}
```

*Exemple de définition et d'utilisation d'une classe (*point*)*



Remarques

1 Dans le jargon de la P.O.O., on dit que *a* et *b* sont des **instances** de la classe *point*, ou encore que ce sont des **objets** de type *point* ; c'est généralement ce dernier terme que nous utiliserons.

2 Dans notre exemple, tous les membres données de *point* sont privés, ce qui correspond à une encapsulation complète des données. Ainsi, une tentative d'utilisation directe (ici au sein de la fonction *main*) du membre *a* :

```
a.x = 5
```

conduirait à un diagnostic de compilation (bien entendu, cette instruction serait acceptée si nous avions fait de *x* un membre public).

En général, on cherchera à respecter le principe d'encapsulation des données, quitte à prévoir des fonctions membres appropriées pour y accéder.

3 Dans notre exemple, toutes les fonctions membres étaient publiques. Il est tout à fait possible d'en rendre certaines privées. Dans ce cas, de telles fonctions ne seront plus accessibles de l'« extérieur » de la classe. Elles ne pourront être appelées que par d'autres fonctions membres.

4 Les mots-clés *public* et *private* peuvent apparaître à plusieurs reprises dans la définition d'une classe, comme dans cet exemple :

```
class X
{     private :
      ...
      public :
      ...
      private :
      ...
} ;
```

Si aucun de ces deux mots n'apparaît au début de la définition, tout se passe comme si *private* y avait été placé. C'est pourquoi la présence de ce mot n'était pas indispensable dans la définition de notre classe *point*.

Si aucun de ces deux mots n'apparaît dans la définition d'une classe, tous ses membres seront privés, donc inaccessibles. Cela sera rarement utile.

5 Si l'on rend publics tous les membres d'une classe, on obtient l'équivalent d'une structure. Ainsi, ces deux déclarations définissent le même type *point* :

```
struct point                          class point
{  int x ;                            { public :
   int y ;                                int x ;
   void initialise (...) ;                int y ;
    .....                                 void initialise (...) ;
} ;                                         ....
                                      } ;
```

6 Par la suite, en l'absence de précisions supplémentaires, nous utiliserons le mot **classe** pour désigner indifféremment une « vraie » classe (*class*) ou une structure (*struct*), voire une union (*union*) dont nous parlerons un peu plus loin[1]. De même, nous utiliserons le mot **objet** pour désigner des instances de ces différents types.

7 En toute rigueur, il existe un troisième mot, *protected* (protégé), qui s'utilise de la même manière que les deux autres ; il sert à définir un statut intermédiaire entre public et privé, lequel n'intervient que dans le cas de classes dérivées. Nous en reparlerons au chapitre 19.

8 On peut définir des classes anonymes, comme on pouvait définir des structures anonymes.

1. La situation de loin la plus répandue restant celle du type *class*.

3 Affectation d'objets

Nous avons déjà vu comment affecter à une structure (usuelle) la valeur d'une autre structure de même type. Ainsi, avec les déclarations suivantes :

```
struct point
{
    int x ;
    int y ;
} ;
point a, b ;
```

vous pouvez tout à fait écrire :

```
b = a ;
```

Cette instruction recopie l'ensemble des valeurs des champs de *a* dans ceux de *b*. Elle joue le même rôle que :

```
b.x = a.x ;
b.y = a.y ;
```

Comme on peut s'y attendre, cette possibilité s'étend aux structures généralisées présentées précedemment, avec la même signification que pour les structures usuelles. Mais elle s'étend aussi aux (vrais) objets de même type. Elle correspond tout naturellement à une **recopie des valeurs des membres données**[1], **que ceux-ci soient publics ou non**. Ainsi, avec ces déclarations (notez qu'ici nous avons prévu, artificiellement, *x* privé et *y* public) :

```
class point
{
    int x ;
   public :
     int y ;
      ....
} ;
point a, b ;
```

l'instruction :

```
b = a ;
```

provoquera la recopie des valeurs des membres *x* et *y* de *a* dans les membres correspondants de *b*.

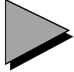
Contrairement à ce qui a été dit pour les structures, il n'est plus possible ici de remplacer cette instruction par :

```
b.x = a.x ;
b.y = a.y ;
```

1. Les fonctions membres n'ont aucune raison d'être concernées.

En effet, si la deuxième affectation est légale, puisque ici *y* est public, la première ne l'est pas, car *x* est privé[1]. On notera bien que :

> L'affectation *a* = *b* est toujours légale, quel que soit le statut (public ou privé) des membres données. On peut considérer qu'elle ne viole pas le principe d'encapsulation, dans la mesure où les données privées de *b* (les copies de celles de *a*, après affectation) restent toujours inaccessibles de manière directe.



Remarque

Le rôle de l'opérateur = tel que nous venons de le définir (recopie des membres données) peut paraître naturel ici. En fait, il ne l'est que pour des cas simples. Nous verrons des circonstances où cette banale recopie s'avérera insuffisante. Ce sera notamment le cas dès qu'un objet comportera des pointeurs sur des emplacements dynamiques : la recopie en question ne concernera pas cette partie dynamique de l'objet, elle sera « superficielle ». Nous reviendrons ultérieurement sur ce point fondamental, qui ne trouvera de solution satisfaisante que dans la surdéfinition (pour la classe concernée) de l'opérateur = (ou, éventuellement, dans l'interdiction de son utilisation).

En Java

En C++, on peut dire que la « sémantique » d'affectation d'objets correspond à une recopie de valeur. En Java, il s'agit simplement d'une recopie de référence : après affectation, on se retrouve alors en présence de deux références sur un même objet.

4 Notions de constructeur et de destructeur

4.1 Introduction

A priori, les objets[2] suivent les règles habituelles concernant leur initialisation par défaut : seuls les objets statiques voient leurs données initialisées à zéro. En général, il est donc nécessaire de faire appel à une fonction membre pour attribuer des valeurs aux données d'un objet. C'est ce que nous avons fait pour notre type *point* avec la fonction *initialise*.

Une telle démarche oblige toutefois à compter sur l'utilisateur de l'objet pour effectuer l'appel voulu au bon moment. En outre, si le risque ne porte ici que sur des valeurs non définies, il n'en va plus de même dans le cas où, avant même d'être utilisé, un objet doit effectuer un certain nombre d'opérations nécessaires à son bon fonctionnement, par exemple : alloca-

1. Sauf si l'affectation *b.x = a.x* était écrite au sein d'une fonction membre de la classe *point*.

2. Au sens large du terme.

tion dynamique de mémoire[1], vérification d'existence de fichier ou ouverture, connexion à un site web... L'absence de procédure d'initialisation peut alors devenir catastrophique.

C++ offre un mécanisme très performant pour traiter ces problèmes : le **constructeur**. Il s'agit d'une fonction membre (définie comme les autres fonctions membres) qui sera appelée automatiquement à chaque création d'un objet. Ceci aura lieu quelle que soit la classe d'allocation de l'objet : statique, automatique ou dynamique. Notez que les objets automatiques auxquels nous nous limitons ici sont créés par une déclaration. Ceux de classe dynamique seront créés par *new* (nous y reviendrons au chapitre 13).

Un objet pourra aussi posséder un **destructeur**, c'est-à-dire une fonction membre appelée automatiquement au moment de la destruction de l'objet. Dans le cas des objets automatiques, la destruction de l'objet a lieu lorsque l'on quitte le bloc ou la fonction où il a été déclaré.

Par convention, le constructeur se reconnaît à ce qu'il porte le même nom que la classe. Quant au destructeur, il porte le même nom que la classe, précédé d'un tilde (~).

4.2 Exemple de classe comportant un constructeur

Considérons la classe *point* précédente et transformons simplement notre fonction membre *initialise* en un constructeur en la renommant *point* (dans sa déclaration et dans sa définition). La déclaration de notre nouvelle classe *point* se présente alors ainsi :

```
class point
{             /* déclaration des membres privés */
    int x ;
    int y ;
  public :     /* déclaration des membres publics */
    point (int, int) ;          // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;
```

*Déclaration d'une classe (*point*) munie d'un constructeur*

Comment utiliser cette classe ? A priori, vous pourriez penser que la déclaration suivante convient toujours :

```
point a ;
```

1. Ne confondez pas un objet dynamique avec un objet (par exemple automatique) qui s'alloue dynamiquement de la mémoire. Une situation de ce type sera étudiée au prochain chapitre.

En fait, à partir du moment où un constructeur est défini, il doit pouvoir être appelé (automatiquement) lors de la création de l'objet *a*. Ici, notre constructeur a besoin de deux arguments. Ceux-ci doivent obligatoirement être fournis dans notre déclaration, par exemple :

```
point a(1,3) ;
```

Cette contrainte est en fait un excellent garde-fou :

> À partir du moment où une classe possède un constructeur, il n'est plus possible de créer un objet sans fournir les arguments requis par son constructeur (sauf si ce dernier ne possède aucun argument !).

À titre d'exemple, voici comment pourrait être adapté le programme du paragraphe 2 pour qu'il utilise maintenant notre nouvelle classe *point* :

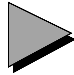
```
#include <iostream>
using namespace std ;
       /* ------------ Déclaration de la classe point ------------- */
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
              /* déclaration des membres publics */
 public :
    point (int, int) ;           // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;

    /* ----- Définition des fonctions membre de la classe point ---- */
point::point (int abs, int ord)
{    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{    x = x + dx ; y = y + dy ;
}
void point::affiche ()
{    cout << "Je suis en " << x << " " << y << "\n" ;
}

        /* -------- Utilisation de la classe point -------- */
int main()
{  point a(5,2) ;
   a.affiche () ;
   a.deplace (-2, 4) ;  a.affiche () ;
   point b(1,-1) ;
   b.affiche () ;
}
```

```
Je suis en 5 2
Je suis en 3 6
Je suis en 1 -1
```

*Exemple d'utilisation d'une classe (*point*) munie d'un constructeur*



Remarques

1 Supposons que l'on définisse une classe *point* disposant d'un constructeur sans argument. Dans ce cas, la déclaration d'objets de type *point* continuera de s'écrire de la même manière que si la classe ne disposait pas de constructeur :

```
point a ;     // déclaration utilisable avec un constructeur sans argument
```

Certes, la tentation est grande d'écrire, par analogie avec l'utilisation d'un constructeur comportant des arguments :

```
point a() ;   // incorrect
```

En fait, cela représenterait la déclaration d'une fonction nommée *a*, ne recevant aucun argument, et renvoyant un résultat de type *point*. En soi, ce ne serait pas une erreur, mais il est évident que toute tentative d'utiliser le symbole *a* comme un objet conduirait à une erreur...

2 Nous verrons dans le prochain chapitre que, comme toute fonction (membre ou ordinaire) un constructeur peut être surdéfini ou posséder des arguments par défaut.

3 Lorsqu'une classe ne définit aucun constructeur, tout se passe en fait comme si elle disposait d'un « constructeur par défaut » ne faisant rien. On peut alors dire que lorsqu'une classe n'a pas défini de constructeur, la création des objets correspondants se fait en utilisant ce constructeur par défaut. Nous retrouverons d'ailleurs le même phénomène dans le cas du « constructeur de recopie », avec cette différence toutefois que le constructeur par défaut aura alors une action précise.

4.3 Construction et destruction des objets

Nous vous proposons ci-dessous un petit programme mettant en évidence les moments où sont appelés respectivement le constructeur et le destructeur d'une classe. Nous y définissons une classe nommée *test* ne comportant que ces deux fonctions membres ; celles-ci affichent un message nous fournissant ainsi une trace de leur appel. En outre, le membre donnée *num* initialisé par le constructeur nous permet d'identifier l'objet concerné (dans la mesure où nous nous sommes arrangés pour qu'aucun des objets créés ne contienne la même valeur). Nous créons des objets automatiques[1] de type *test* à deux endroits différents : dans la fonction *main* d'une part, dans une fonction *fct* appelée par *main* d'autre part.

1. Rappelons qu'ici nous nous limitons à ce cas.

```
#include <iostream>
using namespace std ;
class test
{ public :
   int num ;
   test (int) ;           // déclaration constructeur
   ~test () ;             // déclaration destructeur
} ;
test::test (int n)       // définition constructeur
{  num = n ;
   cout << "++ Appel constructeur - num = " << num << "\n" ;
}
test::~test ()           // définition destructeur
{  cout << "-- Appel destructeur  - num = " << num << "\n" ;
}
int main()
{  void fct (int) ;
   test a(1) ;
   for (int i=1 ; i<=2 ; i++) fct(i) ;
}
void fct (int p)
{   test x(2*p) ;      // notez l'expression (non constante) : 2*p
}
```

```
++ Appel constructeur - num = 1
++ Appel constructeur - num = 2
-- Appel destructeur  - num = 2
++ Appel constructeur - num = 4
-- Appel destructeur  - num = 4
-- Appel destructeur  - num = 1
```

Construction et destruction des objets

4.4 Rôles du constructeur et du destructeur

Dans les exemples précédents, le rôle du constructeur se limitait à une initialisation de l'objet à l'aide des valeurs qu'il avait reçues en arguments. Mais le travail réalisé par le constructeur peut être beaucoup plus élaboré. Voici un programme exploitant une classe nommée *hasard*, dans laquelle le constructeur fabrique dix valeurs entières aléatoires qu'il range dans le membre donnée *val* (ces valeurs sont comprises entre zéro et la valeur qui lui est fournie en argument) :

```
#include <iostream>
#include <cstdlib>        // pour la fonction rand
using namespace std ;
class hasard
{  int val[10] ;
 public :
   hasard (int) ;
   void affiche () ;
} ;
hasard::hasard (int max)  // constructeur : il tire 10 valeurs au hasard
                          // rappel : rand fournit un entier entre 0 et RAND_MAX
{  int i ;
   for (i=0 ; i<10 ; i++) val[i] = double (rand()) / RAND_MAX * max ;
}
void hasard::affiche ()          // pour afficher les 10 valeurs
{  int i ;
   for (i=0 ; i<10 ; i++) cout << val[i] << " " ;
   cout << "\n" ;
}

int main()
{  hasard suite1 (5) ;
   suite1.affiche () ;
   hasard suite2 (12) ;
   suite2.affiche () ;
}
```

```
0 2 0 4 2 2 1 4 4 3
2 10 8 6 3 0 1 4 1 1
```

Un constructeur de valeurs aléatoires

En pratique, on préférera d'ailleurs disposer d'une classe dans laquelle le nombre de valeurs (ici fixé à *10*) pourra être fourni en argument du constructeur. Dans ce cas, il est préférable que l'espace (variable) soit alloué dynamiquement au lieu d'être surdimensionné. Il est alors tout naturel de faire effectuer cette allocation dynamique par le constructeur lui-même. Les données de la classe *hasard* se limiteront ainsi à :

```
class hasard
    {
      int nbval   // nombre de valeurs
      int * val   // pointeur sur un tableau de valeurs
      ...
    } ;
```

Bien sûr, il faudra prévoir que le constructeur reçoive en argument, outre la valeur maximale, le nombre de valeurs souhaitées.

Par ailleurs, à partir du moment où un emplacement a été alloué dynamiquement, il faut se soucier de sa libération lorsqu'il sera devenu inutile. Là encore, il paraît tout naturel de confier ce travail au destructeur de la classe.

Voici comment nous pourrions adapter en ce sens l'exemple précédent.

```
#include <iostream>
#include <cstdlib>      // pour la fonction rand
using namespace std ;
class hasard
{  int nbval ;           // nombre de valeurs
   int * val ;           // pointeur sur les valeurs
 public :
   hasard (int, int) ;   // constructeur
   ~hasard () ;          // destructeur
   void affiche () ;
} ;
hasard::hasard (int nb, int max)
{  int i ;
   val = new int [nbval = nb] ;
   for (i=0 ; i<nb ; i++) val[i] = double (rand()) / RAND_MAX * max ;
}
hasard::~hasard ()
{  delete val ;
}
void hasard::affiche ()          // pour afficher les nbavl valeurs
{  int i ;
   for (i=0 ; i<nbval ; i++) cout << val[i] << " " ;
   cout << "\n" ;
}
int main()
{  hasard suite1 (10, 5) ;       // 10 valeurs entre 0 et 5
   suite1.affiche () ;
   hasard suite2 (6, 12) ;       // 6 valeurs entre 0 et 12
   suite2.affiche () ;
}
```

```
0 2 0 4 2 2 1 4 4 3
2 10 8 6 3 0
```

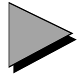
Exemple de classe dont le constructeur effectue une allocation dynamique de mémoire

Dans le constructeur, l'instruction :

```
val = new [nbval = nb] ;
```

joue le même rôle que :

```
nbval = nb ;
val = new [nbval] ;
```



Remarques

1 Ne confondez pas une allocation dynamique effectuée au sein d'une fonction membre d'un objet (souvent le constructeur) avec une allocation dynamique d'un objet, dont nous parlerons plus tard.

2 Lorsqu'un constructeur se contente d'attribuer des valeurs initiales aux données d'un objet, le destructeur est rarement indispensable. En revanche, il le devient dès que, comme dans notre exemple, l'objet est amené (par le biais de son constructeur ou d'autres fonctions membres) à allouer dynamiquement de la mémoire.

3 Comme nous l'avons déjà mentionné, dès qu'une classe contient, comme dans notre dernier exemple, des pointeurs sur des emplacements alloués dynamiquement, l'affectation entre objets de même type ne concerne pas ces parties dynamiques ; généralement, cela pose problème et la solution passe par la surdéfinition de l'opérateur =. Autrement dit, la classe *hasard* définie dans le dernier exemple ne permettrait pas de traiter correctement l'affectation d'objets de ce type.

4.5 Quelques règles

Un constructeur peut comporter un nombre quelconque d'arguments, éventuellement aucun. Par définition, un constructeur ne renvoie pas de valeur ; aucun type ne peut figurer devant son nom (dans ce cas précis, la présence de *void* est une erreur). Il n'est pas prévu de mécanisme permettant à un constructeur d'en appeler un autre (C++11 offre une solution dans ce sens).

Par définition, un destructeur ne peut pas disposer d'arguments et ne renvoie pas de valeur. Là encore, aucun type ne peut figurer devant son nom (et la présence de *void* est une erreur).

En théorie, constructeurs et destructeurs peuvent être publics ou privés. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre publics.

On notera que, si un destructeur est privé, il ne pourra plus être appelé directement, ce qui n'est généralement pas grave, dans la mesure où cela est rarement utile.

En revanche, la privatisation d'un constructeur a de lourdes conséquences puisqu'il ne sera plus utilisable, sauf par des fonctions membres de la classe elle-même.

Informations complémentaires

Voici quelques circonstances où un constructeur privé peut se justifier :

– la classe concernée ne sera pas utilisée telle quelle car elle est destinée à donner naissance, par héritage, à des classes dérivées qui, quant à elles, pourront disposer d'un constructeur public ; nous reviendrons plus tard sur cette situation dite de « classe abstraite » ;

– la classe dispose d'autres constructeurs (nous verrons bientôt qu'un constructeur peut être surdéfini), dont au moins un est public ;

– on cherche à mettre en œuvre un motif de conception[1] particulier : le « singleton » ; il s'agit de faire en sorte qu'une même classe ne puisse donner naissance qu'à un seul objet et que toute tentative de création d'un nouvel objet se contente de renvoyer la référence de cet unique objet. Dans ce cas, on peut prévoir un constructeur privé (de corps vide) dont la présence fait qu'il est impossible de créer explicitement des objets du type (du moins si ce constructeur n'est pas surdéfini). La création d'objets se fait alors par appel d'une fonction membre statique qui réalise elle-même les allocations nécessaires, c'est-à-dire le travail d'un constructeur habituel, et qui, en outre, s'assure de l'unicité de l'objet.

En Java

Le constructeur possède les mêmes propriétés qu'en C++ et une classe peut ne pas comporter de constructeur. Mais, en Java, les membres données sont toujours initialisés par défaut (valeur « nulle ») et ils peuvent également être initialisés lors de leur déclaration (la même valeur étant alors attribuée à tous les objets du type). Ces deux possilités (initialisation par défaut et initialisation explicite) n'existent pas en C++, comme nous le verrons plus tard, de sorte qu'il est pratiquement toujours nécessaire de prévoir un constructeur, même dans des situations d'initialisation simple (C++11 permet les initialisations explicites).

5 Les membres données statiques

5.1 Le qualificatif *static* pour un membre donnée

A priori, lorsque dans un même programme on crée différents objets d'une même classe, chaque objet possède ses propres membres données. Par exemple, si nous avons défini une classe *exple1* par :

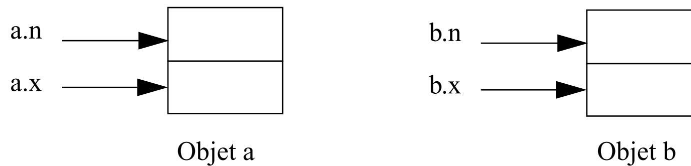
```
class exple1
{   int n ;
    float x ;
     .....
} ;
```

une déclaration telle que :

```
exple1 a, b ;
```

1. *Design pattern*, en anglais.

conduit à une situation que l'on peut schématiser ainsi :



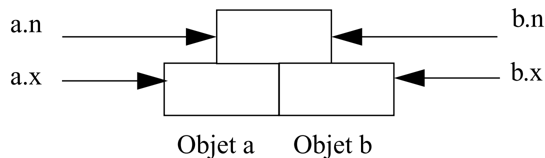
Une façon (parmi d'autres) de permettre à plusieurs objets de partager des données consiste à déclarer avec le qualificatif *static* les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe. Par exemple, si nous définissons une classe *exple2* par :

```
class exple2
{     static int n ;
      float x ;
      ...
} ;
```

la déclaration :

```
exple2 a, b ;
```

conduit à une situation que l'on peut schématiser ainsi :



On peut dire que les membres données statiques sont des sortes de variables globales dont la portée est limitée à la classe.

5.2 Initialisation des membres données statiques

Par leur nature même, les membres données statiques n'existent qu'en un seul exemplaire, indépendamment des objets de la classe (même si aucun objet de la classe n'a encore été créé). Dans ces conditions, leur initialisation ne peut plus être faite par le constructeur de la classe.

On pourrait penser qu'il est possible d'initialiser un membre statique lors de sa déclaration, comme dans :

```
class exple2
{  static int n = 2 ;     // erreur
   .....
} ;
```

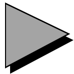
En fait, cela n'est pas permis car, compte tenu des possibilités de compilation séparée, le membre statique risquerait de se voir réserver différents emplacements[1] dans différents modules objets.

Un membre statique doit donc être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que :

```
int exple2::n = 5 ;
```

Cette démarche est utilisable aussi bien pour les membres statiques privés que publics.

Par ailleurs, contrairement à ce qui se produit pour une variable ordinaire, un membre statique n'est pas initialisé par défaut à zéro.



Remarque

Les membres statiques **constants** peuvent être initialisés comme nous venons de le voir, mais également au moment de leur déclaration :

```
class exple3
{ static const int n=5 ;   // initialisation OK
  .....
}
```

Ce sont les seuls membres susceptibles d'être initialisés lors de leur déclaration. C++11 étend cette possibilité aux membres non statiques.

5.3 Exemple

Voici un exemple de programme exploitant cette possibilité dans une classe nommée *cpte_obj*, afin de connaître, à tout moment, le nombre d'objets existants. Pour ce faire, nous avons déclaré avec l'attribut statique le membre *ctr*. Sa valeur est incrémentée de 1 à chaque appel du constructeur et décrémentée de 1 à chaque appel du destructeur.

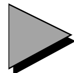
```
#include <iostream>
using namespace std ;
class cpte_obj
{  static int ctr ;             // compteur du nombre d'objets créés
 public :
   cpte_obj () ;
   ~cpte_obj () ;
} ;
int cpte_obj::ctr = 0 ;  // initialisation du membre statique ctr
cpte_obj::cpte_obj ()          // constructeur
{  cout << "++ construction : il y a maintenant   " << ++ctr << " objets\n" ;
}
```

1. On trouvait le même phénomène pour les variables globales en langage C : elles pouvaient être déclarées plusieurs fois, mais elles ne devaient être définies qu'une seule fois.

```
cpte_obj::~cpte_obj ()          // destructeur
{   cout << "-- destruction  : il reste maintenant " << --ctr << " objets\n" ;
}
int main()
{  void fct () ;
   cpte_obj a ;
   fct () ;
   cpte_obj b ;
}
void fct ()
{  cpte_obj u, v ;
}
```

```
++ construction : il y a maintenant   1 objets
++ construction : il y a maintenant   2 objets
++ construction : il y a maintenant   3 objets
-- destruction  : il reste maintenant 2 objets
-- destruction  : il reste maintenant 1 objets
++ construction : il y a maintenant   2 objets
-- destruction  : il reste maintenant 1 objets
-- destruction  : il reste maintenant 0 objets
```

Exemple d'utilisation de membre statique



Remarque

Nous avons déjà vu que ce même mot-clé *stati*c etait utilisé dans ces situations :

- pour attribuer la classe d'allocation statique à une variable locale ;
- pour cacher une variable globale dans un fichier source (comme nous l'avons vu au paragraphe 12.4 du chapitre 7).

Nous venons de lui en découvrir une troisième, pour demander qu'un membre donnée soit indépendant d'une quelconque instance de la classe. Nous verrons au prochain chapitre qu'il pourra s'appliquer aux fonctions membres avec la même signification.

En Java

Les membres données statiques existent également en Java, et on utilise le mot clé *static* pour leur déclaration (c'est d'ailleurs la seule signification de ce mot-clé). Comme en C++, ils peuvent être initialisés lors de leur déclaration ; mais ils peuvent aussi l'être par le biais d'un *bloc d'initialisation* qui contient alors des instructions exécutables, ce que ne permet pas C++.

6 Exploitation d’une classe

6.1 La classe comme composant logiciel

Jusqu’ici, nous avions regroupé au sein d’un même programme trois sortes d’instructions destinées à :

- la déclaration de la classe ;
- la définition de la classe ;
- l’utilisation de la classe.

En pratique, on aura souvent intérêt à découpler la classe de son utilisation. C’est tout naturellement ce qui se produira avec une classe d’intérêt général utilisée comme un composant séparé des différentes applications.

On sera alors généralement amené à isoler les seules instructions de déclaration de la classe dans un fichier en-tête (extension *.h*) qu’il suffira d’inclure (par *#include*) pour compiler l’application.

Par exemple, le concepteur de la classe *point* du paragraphe 4.2 pourra créer le fichier en-tête suivant :

```
class point
{              /* déclaration des membres privés */
    int x ;
    int y ;
  public :     /* déclaration des membres publics */
    point (int, int) ;          // constructeur
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Fichier en-tête pour la classe point

Si ce fichier se nomme *point.h*, le concepteur fabriquera alors un module objet, en compilant la définition de la classe *point* :

```
#include <iostream>
#include "point.h"   // pour introduire les déclarations de la classe point
using namespace std ;
     /* ----- Définition des fonctions membre de la classe point ---- */
point::point (int abs, int ord)
{    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{    x = x + dx ; y = y + dy ;
}
```

```
void point::affiche ()
{    cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Fichier à compiler pour obtenir le module objet de la classe point

Pour faire appel à la classe *point* au sein d'un programme, l'utilisateur procédera alors ainsi :

- Il inclura la déclaration de la classe *point* dans le fichier source contenant son programme par une directive telle que :

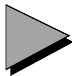
  ```
  #include "point.h"
  ```

  Rappelons que la directive *#include* possède deux syntaxes très voisines : l'une utilise la forme <.....> pour les fichiers en-tête standards, l'autre la forme ″.....″ pour les fichiers en-tête fournis par l'utilisateur.

- Il incorporera le module objet correspondant, au moment de l'édition de liens de son propre programme. En principe, à ce niveau, la plupart des éditeurs de liens n'introduisent que les fonctions réellement utilisées, de sorte qu'il ne faut pas craindre de prévoir trop de méthodes pour une classe.

Parfois, on trouvera plusieurs classes différentes au sein d'un même module objet et d'un même fichier en-tête, de façon comparable à ce qui se produit avec les fonctions de la bibliothèque standard[1]. Là encore, en général, seules les fonctions réellement utilisées seront incorporées à l'édition de liens, de sorte qu'il est toujours possible d'effectuer des regroupements de classes possédant quelques affinités.

Signalons que bon nombre d'environnements disposent d'outils[2] permettant de prendre automatiquement en compte les « dépendances » existant entre les différents fichiers sources et les différents fichiers objets concernés ; dans ce cas, lors d'une modification, quelle qu'elle soit, seules les compilations nécessaires sont effectuées.



Remarque

Comme une fonction ordinaire, une fonction membre peut être déclarée sans qu'on n'en fournisse de définition. Si le programme fait appel à cette fonction membre, ce n'est qu'à l'édition de liens qu'on s'apercevra de son absence. En revanche, si le programme n'utilise pas cette fonction membre, l'édition de liens se déroulera normalement car il n'introduit que les fonctions effectivement appelées.

1. Avec cette différence que, dans le cas des fonctions standards, on n'a pas à spécifier les modules objets concernés au moment de l'édition de liens.

2. On parle souvent de *projet*, de *fichier projet*, de fichier *make*...

6.2 Protection contre les inclusions multiples

Plus tard, nous verrons qu'il existe différentes circonstances pouvant amener l'utilisateur d'une classe à inclure plusieurs fois un même fichier en-tête lors de la compilation d'un même fichier source (sans même qu'il n'en ait conscience !). Ce sera notamment le cas dans les situations d'objets membres et de classes dérivées.

Dans ces conditions, on risque d'aboutir à des erreurs de compilation, liées tout simplement à la redéfinition de la classe concernée.

En général, on réglera ce problème en protégeant systématiquement tout fichier en-tête des inclusions multiples par une technique de compilation conditionnelle (présentée au paragraphe 3 du chapitre 31), comme dans :

```
#ifndef POINT_H
#define POINT_H
// déclaration de la classe point
#endif
```

Le symbole défini pour chaque fichier en-tête sera choisi de façon à éviter tout risque de doublons. Ici, nous avons choisi le nom de la classe (en majuscules), suffixé par _H.

6.3 Cas des membres données statiques

Nous avons vu (paragraphe 5.2) qu'un membre donnée statique doit toujours être initialisé explicitement. Dès qu'on est amené à considérer une classe comme un composant séparé, le problème se pose alors de savoir dans quel fichier source placer une telle initialisation : fichier en-tête, fichier définition de la classe, fichier utilisateur (dans notre exemple du paragraphe 5.3, ce problème ne se posait pas car nous n'avions qu'un seul fichier source).

On pourrait penser que le fichier en-tête est un excellent candidat pour cette initialisation, dès lors qu'il est protégé contre les inclusions multiples. En fait, il n'en est rien ; en effet, si l'utilisateur compile séparément plusieurs fichiers source utilisant la même classe, plusieurs emplacements seront générés pour le même membre statique et, en principe, l'édition de liens détectera cette erreur.

Comme par ailleurs il n'est guère raisonnable de laisser l'utilisateur initialiser lui-même un membre statique, on voit qu'en définitive :

> Il est conseillé de prévoir l'initialisation des membres données statiques dans le fichier contenant la définition de la classe.

6.4 Modification d'une classe

6.4.1 Notion d'interface et d'implémentation

Une bonne conception orientée objet s'appuie sur la notion de « contrat » qui consiste à considérer qu'une classe est caractérisée par un ensemble de services définis par :

- les en-têtes de ses fonctions membres publiques ; cet ensemble se nomme souvent « l'interface » de la classe ;
- le comportement de ces fonctions membres.

L'utilisateur de la classe n'a rien d'autre à connaître. Il peut donc ignorer totalement ce que l'on nomme souvent « l'implémentation » de la classe : corps des méthodes publiques, membres données (on se place dans un contexte d'encapsulation totale), fonctions membres privées.

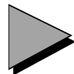
Le contrat définit ce que fait la classe ; son implémentation précise comment elle le fait. L'interface montre comment utiliser les méthodes publiques. Elle constitue l'intégralité de l'information qu'a à connaître l'utilisateur, pour peu que les données soient convenablement encapsulées.

Lorsqu'une classe, considérée comme un composant logiciel, a besoin d'être modifiée, il faut distinguer deux situations très différentes suivant que les modifications atteignent ou non son interface.

6.4.2 Modification d'une classe sans modification de son interface

De telles modifications n'ont alors aucune répercussion sur la manière d'utiliser la classe. Il peut, par exemple, s'agir de transformations de structures de données encapsulées (privées), de modifications d'algorithmes de traitement, d'améliorations de performances...

Dans ce cas, **les programmes utilisant la classe** n'ont pas à être modifiés. Néanmoins, il **doivent être recompilés avec le nouveau fichier en-tête correspondant**[1]. On procédera ensuite à une édition de liens en incorporant le nouveau module objet.



Remarque

En général, on a tendance à confondre la notion d'interface avec celle, plus générale, de contrat, qui fait intervenir une information supplémentaire peu formelle, concernant ce que font réellement les méthodes. En toute rigueur, on pourrait imaginer une modification de contrat, sans modification d'interface, qui puisse compromettre la bonne utilisation de la classe. Ce serait le cas si, dans une de nos classes *Point*, la méthode *deplace* affectait aux coordonéées les valeurs reçues ou, encore, si la méthode *affiche* remettait à 0 les coordonnées d'un point.

6.4.3 Modification d'une classe avec modification de son interface

Ici, il est clair que les programmes utilisant la classe risquent de nécessiter des modifications. Cette situation devra bien sûr être évitée dans la mesure du possible. Elle doit être considérée comme une faute de conception de la classe. Nous verrons d'ailleurs que ces problèmes pour-

1. Une telle limitation n'existe pas dans tous les langages de P.O.O. En C++, elle se justifie par le besoin qu'a le compilateur de connaître la taille des objets (statiques ou automatiques) pour leur allouer un emplacement.

ront souvent être résolus par l'utilisation du mécanisme d'héritage qui permet d'adapter une classe (censée être au point) sans la remettre en cause.

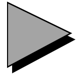
7 Les classes en général

Nous apportons ici quelques compléments d'information sur des situations peu usuelles.

7.1 Les autres sortes de classes en C++

Nous avons déjà eu l'occasion de dire que C++ qualifiait de « classes » les types définis par *struct* et *class*. La caractéristique d'une classe, au sens large que lui donne C++[1], est d'associer, au sein d'un même type, des membres données et des fonctions membres.

Pour C++, les **unions sont aussi des classes**. Ce type peut donc disposer de fonctions membres. Notez bien que, comme pour le type *struct,* les données correspondantes ne peuvent pas se voir attribuer un statut particulier : elles sont, de fait, publiques.



Remarque

C++ emploie souvent le mot *classe* pour désigner indifféremment un type *class*, *struct* ou *union*. De même, on parle souvent d'*objet* pour désigner des variables de l'un de ces trois types. Cet « abus de langage » semble assez licite, dans la mesure où ces trois types jouissent pratiquement des mêmes propriétés, notamment au niveau de l'héritage ; toutefois, seul le type *class* permet l'encapsulation des données. Lorsqu'il sera nécessaire d'être plus précis, nous parlerons de « vraie classe » pour désigner le type *class*.

7.2 Ce qu'on peut trouver dans la déclaration d'une classe

En dehors des déclarations de fonctions membres, la plupart des instructions figurant dans une déclaration de classe seront des déclarations de membres données d'un type quelconque. Néanmoins, on peut également y rencontrer des déclarations de type, y compris d'autres types classes ; dans ce cas, leur portée est limitée à la classe (mais on peut recourir à l'opérateur de résolution de portée ::), comme dans cet exemple :

```
class A
{  public :
     class B { ..... } ;      // classe B déclarée dans la classe A
} ;
int main()
{ A a ;
  A::B b ;        // déclaration d'un objet b du type de la classe B de A
}
```

1. Et non la P.O.O. d'une manière générale, qui associe l'encapsulation des données à la notion de classe.

En pratique, cette situation se rencontre peu souvent.

Par ailleurs, il n'est pas possible (sauf avec C++11) d'initialiser un membre donnée lors de sa déclaration :

```
class X
{ int n = 0 ;    //  interdit
    .....
} ;
```

En revanche, la déclaration de membres données constants[1] est autorisée, comme dans :

```
class exple
{  int n ;           // membre donnée usuel
   const int p ;     // membre donnée constant - initialisation impossible
    .....            // à ce niveau - constructeur explicite obligatoire
} ;
```

Dans ce cas, on notera bien que chaque objet du type *exple* possédera un membre *p*. C'est ce qui explique qu'il ne soit pas possible d'initialiser le membre constant au moment de sa déclaration[2]. Pour y parvenir, la seule solution consistera à utiliser une syntaxe particulière du constructeur (qui devient donc obligatoire), telle qu'elle sera présentée au paragraphe 6 du chapitre 13 (relatif aux objets membres).

En Java

Java autorise l'initialisation de membres dans la déclaration de la classe. La notion de membre constant existe également et elle utilise l'attribut *final*.

7.3 Emplacement de la déclaration d'une classe

La plupart du temps, les classes seront déclarées à un niveau global. Néanmoins, il est permis de déclarer des classes locales à une fonction. Dans ce cas, leur portée est naturellement exclusivement limitée à cette fonction, sans possibilité, cette fois, de reourir à un quelconque opérateur de résolution de portée.

1. Ne confondez pas la notion de membre donnée constant (chaque objet en possède un ; sa valeur ne peut pas être modifiée) et la notion de membre donnée statique (tous les objets d'une même classe partagent le même ; sa valeur peut changer).

2. Sauf, comme on l'a vu au paragraphe 5.2, s'il s'agit d'un membre statique constant ; dans ce cas, ce membre est unique pour tous les objets de la classe.