Claude Delannoy

Programmer en langage C++

8e édition

 ISBN : 978-2-212-12976-2

EYROLLES

Avant-propos

1 Historique de C++

Très tôt, les concepts de la programmation orientée objet (en abrégé P.O.O.) ont donné naissance à de nouveaux langages dits « orientés objets » tels que Smalltalk, Simula, Eiffel ou, plus récemment, Java. Le langage C++, quant à lui, a été conçu suivant une démarche hybride. En effet, Bjarne Stroustrup, son créateur, a cherché à adjoindre à un langage structuré existant (le C), un certain nombre de spécificités lui permettant d'appliquer les concepts de P.O.O. Dans une certaine mesure, il a permis à des programmeurs C d'effectuer une transition en douceur de la programmation structurée vers la P.O.O. De sa conception jusqu'à sa normalisation, le langage C++ a quelque peu évolué. Initialement, un certain nombre de publications de AT&T ont servi de référence du langage. Les dernières en date sont : la version 2.0 en 1989, les versions 2.1 et 3 en 1991. C'est cette dernière qui a servi de base au travail du comité ANSI qui, sans la remettre en cause, l'a enrichie de quelques extensions et surtout de composants standard originaux se présentant sous forme de fonctions et de classes génériques qu'on désigne souvent par le sigle S.T.L (*Standard Template Library*). La norme définitive de C++ a été publiée par l'ANSI en juillet 1998. Quelques modifications mineures ont été apportées en 2003, mais elles ne concernent pas l'utilisateur. En revanche, un nouveau standard, dit C++0x est pratiquement défini et devrait être publié en 2011 (le standard actuel étant noté indifféremment C++98 ou C++03).

2 Objectifs et structure de l'ouvrage

Cet ouvrage est destiné à tous ceux qui souhaitent maîtriser la programmation en C++. Il s'adresse à la fois aux étudiants, aux développeurs et aux enseignants en informatique. Il ne requiert aucune connaissance en P.O.O, ni en langage C[1] ; en revanche, il suppose que le lecteur possède déjà une expérience de la programmation structurée, c'est-à-dire qu'il est habitué aux notions de variables, de types, d'affectation, de structures de contrôle, de fonctions, etc., qui sont communes à la plupart des langages en usage aujourd'hui (Cobol, C/C++, C#, Visual Basic, Delphi, Perl, Python, JavaScript, Java, PHP...).

L'ouvrage est conçu sous la forme d'un cours progressif. Généralement, chaque notion fondamentale est illustrée d'un programme simple mais complet (et assorti d'un exemple d'exécution) montrant comment la mettre en œuvre dans un contexte réel. Cet exemple peut également servir à une prise de connaissance intuitive ou à une révision rapide de la notion en question, à une expérimentation directe dans votre propre environnement de travail ou encore de point de départ à une expérimentation personnelle.

Nous y étudions l'ensemble des possibilités de programmation structurée du C++, avant d'aborder les concepts orientés objet. Cette démarche se justifie pour les raisons suivantes :

- La P.O.O. s'appuie sur la plupart des concepts de programmation structurée : variables, types, affectation, structure de contrôle, etc. Seul le concept de fonction se trouve légèrement adapté dans celui de « méthode ».
- Le C++ permet de définir des « fonctions ordinaires » au même titre qu'un langage non orienté objet, ce qui n'est théoriquement pas le cas d'un pur langage objet où il n'existe que des méthodes s'appliquant obligatoirement à des objets[2]. Ces fonctions ordinaires (indépendantes d'un objet) sont même indispensables en C++ dans certaines circonstances telles que la surdéfinition d'opérateurs. Ne pas utiliser de telles fonctions, sous prétexte qu'elles ne correspondent pas à une « pure programmation objet », reviendrait à se priver de certaines possibilités du langage.
- Sur un plan pédagogique, il est plus facile de présenter la notion de classe quand sont assimilées les autres notions fondamentales sur lesquelles elle s'appuie.

Les aspects orientés objet sont ensuite abordés de façon progressive, mais sans pour autant nuire à l'exhaustivité de l'ouvrage. Nous y traitons, non seulement les purs concepts de P.O.O. (classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique), mais aussi les aspects très spécifiques au langage (surdéfinition d'opérateurs, fonctions amies, flots, gestion d'exceptions). Nous pensons ainsi permettre au lecteur de devenir parfaitement opérationnel dans la conception, le développement et la mise au point

1. Un autre ouvrage du même auteur, *C++ pour les programmeurs C*, s'adresse spécifiquement aux connaisseurs du langage C.

2. En fait, certains langages, dont Java, permettent de définir des « méthodes de classe », indépendantes d'un quelconque objet, jouant finalement pratiquement le même rôle que ces fonctions ordinaires.

de ses propres classes. C'est ainsi, par exemple, que nous avons largement insisté sur le rôle du constructeur de recopie, ainsi que sur la redéfinition de l'opérateur d'affectation, éléments qui conduisent à la notion de « classe canonique ». Toujours dans le même esprit, nous avons pris soin de bien développer les notions avancées mais indispensables que sont la ligature dynamique et les classes abstraites, lesquelles débouchent sur la notion la plus puissante du langage (et de la P.O.O.) qu'est le polymorphisme. De même, la S.T.L. a été étudiée en détail, après avoir pris soin d'exposer préalablement d'une part les notions de classes et de fonctions génériques, d'autre part celles de conteneur, d'itérateur et d'algorithmes qui conditionnent la bonne utilisation de la plupart de ses composants.

3 L'ouvrage, C, C++ et Java

L'ouvrage est entièrement fondé sur la norme ANSI/ISO du langage C++. Compte tenu de la popularité du langage Java, nous avons introduit de nombreuses remarques titrées « En Java ». Elles mettent l'accent sur les différences majeures existant entre Java et C++. Elles seront utiles non seulement au programmeur Java qui apprend ici le C++, mais également au lecteur qui, après la maîtrise du C++, souhaitera aborder l'étude de Java. En outre, quelques remarques titrées « En C » viennent signaler les différences les plus importantes existant entre C et C++. Elles serviront surtout au programmeur C++ souhaitant réutiliser du code écrit en C.

Enfin, la présente édition a été dotée d'un nouveau chapitre d'introduction aux design patterns, ainsi que d'une annexe présentant les nouveautés de la norme C++0x.

Table des matières

7

Les fonctions

Dès qu'un programme dépasse quelques pages de texte, il est pratique de pouvoir le décomposer en des parties relativement indépendantes dont on pourra comprendre facilement le rôle, sans avoir à examiner l'ensemble du code.

La programmation procédurale permet un premier pas dans ce sens, grâce à la notion de fonction que nous allons aborder dans ce chapitre : il s'agit d'un bloc d'instructions qu'on peut utiliser à loisir dans un programme en citant son nom et, éventuellement, en lui fournissant des « paramètres ».

La P.O.O. constituera une seconde étape dans ce processus de décomposition. Chaque classe, définie de façon indépendante, associera des données et des méthodes ; nous verrons que ces méthodes seront rédigées de façon comparable à des fonctions, de sorte que nous serons alors amenés à utiliser l'essentiel de ce que nous aurons étudié ici.

On notera qu'en C++ (comme en C ou en Java), la fonction possède un rôle plus général que la « fonction mathématique ». En effet, une fonction mathématique :

- possède des arguments dont on fournit la valeur lors de l'appel (par exemple, *x* dans *sqrt(x)* ou *5.2* dans *sqrt(5.2)* ;
- fournit un résultat (scalaire) désigné simplement par son appel : *sqrt(x)* désigne le résultat fourni par la fonction ; on peut l'utiliser directement dans une expression arithmétique comme *y + 2*sqrt(x)*.

Or, en C++, si une fonction peut effectivement, comme *sqrt*, jouer le rôle d'une fonction mathématique, elle pourra aussi :

- modifier les valeurs de certains des arguments qu'on lui a transmis ;

- réaliser une action (autre qu'un simple calcul), par exemple : lire des valeurs, afficher des valeurs, ouvrir un fichier, établir une connexion...
- fournir un résultat d'un type non scalaire (structures, objets...) ;
- fournir une valeur qu'on n'utilisera pas ;
- ne pas fournir de valeur du tout.

Nous commencerons par vous présenter la notion de fonction sur un exemple, et nous donnerons quelques règles générales concernant l'écriture des fonctions, leur utilisation et leur déclaration. Nous verrons ensuite que, par défaut, les arguments sont transmis par valeur et nous apprendrons à demander explicitement une transmission par référence. Nous parlerons succinctement des variables globales, surtout pour en déconseiller l'utilisation. Nous ferons ensuite le point sur la classe d'allocation et l'initialisation des variables locales. Nous apprendrons à définir des valeurs par défaut pour certains arguments d'une fonction. Puis nous étudierons l'importante notion de surdéfinition qui permet de définir plusieurs fonctions de même nom, mais ayant des arguments différents. Nous donnerons alors quelques éléments concernant les possibilités de compilation séparée de C++. Enfin, nous verrons comment définir des « fonctions en ligne ».

1 Exemple de définition et d'utilisation d'une fonction

Pour vous montrer comment définir et utiliser une fonction en C++, nous commencerons par un exemple simple correspondant en fait à une fonction mathématique, c'est-à-dire recevant des arguments et fournissant une valeur.

```
#include <iostream>
using namespace std ;
float fexple (float, int, int) ; // declaration de fonction fexple
          /***** le programme principal (fonction main) *****/
int main ()
{  float x = 1.5 ;
   float y, z ;
   int n = 3, p = 5, q = 10 ;

         /* appel de fexple avec les arguments x, n et p */
   y = fexple (x, n, p) ;
   cout << "valeur de y : " << y << "\n" ;

         /* appel de fexple avec les arguments x+0.5, q et n-1 */
   z = fexple (x+0.5, q, n-1) ;
   cout << "valeur de z : " << z << "\n" ;
}
```

```
        /*************** la fonction fexple ****************/
float fexple (float x, int b, int c)
{  float val ;          // declaration d'une variable "locale" à fexple
   val = x * x + b * x + c ;
   return val ;
}
```

```
valeur de y : 11.75
valeur de z : 26
```

Exemple de définition et d'utilisation d'une fonction

Nous y trouvons tout d'abord, de façon désormais classique, un programme principal formé d'un bloc. Mais, cette fois, à sa suite, apparaît la **définition d'une fonction**. Celle-ci possède une structure voisine de la fonction *main*, à savoir un en-tête et un corps délimité par des accolades ({ et }). Mais l'en-tête est plus élaboré que celui de la fonction *main*. On y trouve le nom de la fonction (*fexple*), une liste d'arguments (nom + type), ainsi que le type de la valeur qui sera fournie par la fonction (on la nomme indifféremment « résultat », « valeur de la fonction », « valeur de retour »...) :

```
float          fexple         (float x,        int b,        int c)
  |              |              |              |             |
type de la    nom de la       premier        deuxième      troisième
"valeur       fonction       argument       argument      argument
de retour"                  (type float)    (type int)    (type int)
```

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration :

```
float val ;
```

Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type *float* nommée *val*. On dit que *val* est une variable locale à la fonction *fexple*, de même que les variables telles que *n*, *p*, *y*... sont des variables locales à la fonction *main* (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de portée qui s'y attache.

L'instruction suivante de notre fonction *fexple* est une affectation classique (faisant toutefois intervenir les valeurs des arguments *x*, *n* et *p*).

Enfin, l'instruction *return val* précise la valeur que fournira la fonction à la fin de son travail.

En définitive, on peut dire que *fexple* est une fonction telle que *fexple (x, b, c)* fournit la valeur de l'expression $x^2 + bx + c$. Notez bien l'aspect arbitraire du nom des arguments ; on obtiendrait la même définition de fonction avec, par exemple :

```
float fexple (float z, int coef, int n)
{
   float val ; // déclaration d'une variable "locale" à fexple
   val = z * z + coef * z + n ;
   return val ;
}
```

Notez qu'avant la fonction *main*, on trouve une déclaration :

```
float fexple (float, int, int) ;
```

Elle sert à prévenir le compilateur que *fexple* est une fonction, et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration, ainsi que sur les endroits où elle peut figurer.

Quant à l'utilisation de notre fonction *fexple* au sein de la fonction *main*, elle est classique et comparable à celle d'une fonction prédéfinie telle que *sqrt*. Ici, nous nous sommes contentés d'appeler notre fonction à deux reprises avec des arguments différents.

2 Quelques règles

2.1 Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des « arguments muets », ou encore « arguments formels » ou « paramètres formels » (de l'anglais : *formal parameter)*. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des « arguments effectifs » (ou encore « paramètres effectifs »). Comme le laisse deviner l'exemple précédent, on peut utiliser n'importe quelle expression comme argument effectif ; au bout du compte, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel. Notez qu'une telle « liberté » n'aurait aucun sens dans le cas des paramètres formels : il serait impossible d'écrire un en-tête de *fexple* sous la forme *float fexple (float, a+b, ...)*, pas plus qu'en mathématiques vous ne définiriez une fonction f par $f(x+y)=5$!

2.2 L'instruction *return*

Voici quelques règles générales concernant cette instruction.

- L'instruction *return* peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction *fexple* précédente de cette manière :

```
float fexple (float x, int b, int c)
{
   return (x * x + b * x + c) ;
}
```

- L'instruction *return* peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :

```
double absom (double u, double v)
{
    double s ;
    s = a + b ;
    if (s>0)    return (s) ;
         else   return (-s)
}
```

  Notez bien que non seulement l'instruction *return* définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (en l'occurrence, ici, la fonction *main*). Nous verrons qu'une fonction peut ne fournir aucune valeur : elle peut alors disposer de plusieurs instructions *return* **sans expression**, interrompant simplement l'exécution de la fonction ; mais elle peut aussi, dans ce cas, ne comporter aucune instruction *return*, le retour étant alors mis en place automatiquement par le compilateur à la fin de la fonction.

- Si le type de l'expression figurant dans *return* est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion : les conversions légales sont celles qui sont autorisées par affectation (avec les mêmes risques de conversions dégradantes, par exemple de *double* en *int*).

  Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

2.3 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé *void*. Par exemple, voici l'en-tête d'une fonction recevant un argument de type *int* et ne fournissant aucune valeur :

```
void sansval (int n)
```

et voici quelle serait sa déclaration :

```
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction *return*. Certains compilateurs ne détecteront toutefois pas l'erreur.

Quand une fonction ne reçoit aucun argument, on se contentera de ne rien mentionner dans la liste d'arguments. Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type *float* (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :

```
float tirage ()
```

Sa déclaration serait très voisine (elle ne diffère que par la présence du point-virgule !) :

```
float tirage () ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni argument ni valeur de retour. Dans ce cas, son en-tête sera de la forme :

```
void message ()
```

et sa déclaration sera :

```
void message () ;
```

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction *affiche_carres* qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments, et une fonction *erreur* qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

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

void affiche_carres (int, int) ; // prototype de affiche_carres
void erreur () ;                 // prototype de erreur
int main ()
{  int debut = 5, fin = 10 ;
      .....
   affiche_carres (debut, fin) ;
      .....
   if (...) erreur () ;
}

void affiche_carres (int d, int f)
{  int i ;
   for (i=d ; i<=f ; i++)
      cout << i << " a pour carré " << i*i << "\n" ;
}
void erreur ()
{  cout << "*** erreur ***\n" ;
}
```

Remarque

En toute rigueur, l'en-tête de la fonction *main* montre l'existence d'une valeur de retour de type *int*. Effectivement, il est possible d'introduire, dans cette fonction *main*, une ou plusieurs instructions *return* accompagnées d'une expression de type *int*. Cette valeur est susceptible d'être utilisée par l'environnement de programmation. Il est convenu que la valeur 0 indique un bon déroulement du programme. Si aucune instruction *return* ne figure dans *main*, tout se passe comme si on trouvait *return 0* à la fin de son exécution.

En C

Le langage C est beaucoup plus tolérant (à tort) que C++ dans les déclarations de fonctions ; on peut omettre le type des arguments (quels que soient leurs types) ou celui de la valeur de retour (s'il s'agit d'un *int*). Mais les règles employées par C++ restent valides (et même conseillées) en C. Une seule incompatibilité existe dans le cas des fonctions sans argument : C utilise le mot *void*, là où C++ demande une liste vide.

3 Les fonctions et leurs déclarations

3.1 Les différentes façons de déclarer une fonction

Dans notre exemple du paragraphe 1, nous avions fourni la définition de la fonction *fexple* **après** celle de la fonction *main*. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float, int, int) ; // déclaration de la fonction fexple
float fexple (float x, int b, int c)
{
    ....
}
int main ()
{      .....
   y = fexple (x, n, p) ;
       .....
}
```

En toute rigueur, dans ce cas, la déclaration de la fonction *fexple* est facultative car, lorsqu'il traduit la fonction *main*, le compilateur connaît déjà la fonction *fexple*. Néanmoins, nous vous déconseillons d'omettre la déclaration de *fexple* dans ce cas ; en effet, il est tout à fait possible qu'ultérieurement vous soyez amené à modifier votre programme source ou même à l'éclater en plusieurs fichiers source comme l'autorisent les possibilités de compilation séparée de C++.

La déclaration d'une fonction porte le nom de **prototype**. Il est possible, dans un prototype, de faire figurer des noms d'arguments, lesquels sont alors totalement arbitraires ; cette possibilité a pour seul intérêt de pouvoir écrire des prototypes qui sont identiques à l'en-tête de la fonction (au point-virgule près), ce qui peut en faciliter la création automatique. Dans notre exemple du paragraphe 1, notre fonction *fexple* aurait pu être **déclarée** ainsi :

```
float fexple (float x, int b, int c) ;
```

3.2 Où placer la déclaration d'une fonction

Dans notre exemple du paragraphe 1, nous avions placé la déclaration de la fonction *fexple* avant la définition de la fonction (*main*) qui y faisait appel. Nous avions donc affaire à une déclaration globale dont la portée s'étendait à l'ensemble du fichier source, ce qui pourrait

permettre, le cas échéant, d'utiliser *fexple* dans d'autres fonctions du même fichier. Il s'agit là de la démarche la plus utilisée, notamment dans les programmes conséquents. Mais, on peut théoriquement recourrir également à des déclarations locales à une fonction. Ainsi, notre exemple du paragraphe 1 aurait pu utiliser ce canevas :

```
int main()
{ float fexple (float, int, int) ;   // déclaration de fexple - portée limitée au main
  .....
}
```

Par ailleurs, nous verrons, au paragraphe 12.1 que, dans les programmes de quelque importance, les déclarations de fonctions sont en fait placées dans des fichiers en-têtes, dont l'inclusion se fait alors préférentiellement à un niveau global.

3.3 Contrôles et conversions induites par le prototype

La déclaration d'une fonction peut être utilisée par le compilateur, de deux façons complètement différentes.

1 Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.

2 Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction *fexple* du paragraphe 1, un appel tel que :

```
fexple (n+1, 2*x, p)
```

sera traduit par :

- l'évaluation de la valeur de l'expression *n+1* (en *int*) et sa conversion en *float* ;
- l'évaluation de la valeur de l'expression *2*x* (en *float*) et sa conversion en *int* (conversion dégradante).

4 Transmission des arguments par valeur

Jusqu'ici, nous nous sommes contentés de dire que les valeurs des arguments étaient transmis à la fonction au moment de son appel. Nous vous proposons de voir ici ce que cela signifie exactement, et les limitations qui en découlent.

Voyez cet exemple :

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

```
void echange (int a, int b) ;
int main ()
{   int n=10, p=20 ;
   cout << "avant appel   : " << n << " " << p << "\n" ;
   echange (n, p) ;
   cout << "apres appel   : " << n << " " << p << "\n" ;
}
void echange (int a, int b)
{
   int c ;
   cout << "début echange : "<< a << " " <<  b << "\n"  ;
   c = a ;
   a = b ;
   b = c ;
   cout << "fin echange   : " << a << " " << b << "\n"  ;
}
```

```
avant appel   : 10 20
début echange : 10 20
fin echange   : 20 10
apres appel   : 10 20
```

Conséquences de la transmission par valeur des arguments

La fonction *echange* reçoit deux valeurs correspondant à ses deux arguments muets *a* et *b*. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs *n* et *p*.

En effet, lors de l'appel de *echange*, il y a eu transmission de la valeur des expressions *n* et *p*. On peut dire que ces valeurs ont été recopiées localement dans la fonction *echange* dans des emplacements nommés *a* et *b*. C'est effectivement sur ces copies qu'a travaillé la fonction *echange*, de sorte que les valeurs des variables *n* et *p* n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

En fait, ce mode de transmission par valeur n'est que le mode utilisé par défaut par C++. Comme nous allons le voir bientôt, le choix explicite d'une transmission par référence permettra de réaliser correctement notre fonction *echange*.

Remarques

1 C'est bien parce que la transmission des arguments se fait « par valeur » que les arguments effectifs peuvent prendre la forme d'une expression quelconque. Et, d'ailleurs, nous verrons qu'avec la transmission par référence, les arguments effectifs ne pourront plus être des expressions, mais simplement des *lvalue*.

2 La norme n'impose aucun ordre pour l'évaluation des différents arguments d'une fonction lors de son appel. En général, ceci est de peu d'importance, excepté dans une situation (fortement déconseillée !) telle que :

```
int i = 10 ;
   ...
f (i++, i) ;  // i++ peut se trouver caclulé avant i - l'appel sera : f (10, 11)
              //                          ou après i - l'appel sera : f (10, 10)
```

3 En toute rigueur, la valeur de retour (lorsqu'elle existe) est elle aussi transmise par valeur, c'est-à-dire qu'elle fait l'objet d'une recopie de la fonction appelée dans la fonction appelante. Ce point peut sembler anodin, mais nous verrons plus tard qu'il existe des circonstances où il s'avère fondamental et où, là encore, il faudra recourir à une transmission par référence.

5 Transmission par référence

Nous venons de voir que, par défaut, les arguments d'une fonction sont transmis par valeur. Comme nous l'avons constaté avec la fonction *echange*, ce mode de transmission ne permet pas à une fonction de modifier la valeur d'un argument. Or, C++ dispose de la notion de **référence**, laquelle correspond à celle d'adresse : considérer la référence d'une variable revient à considérer son adresse, et non plus sa valeur. Nous commencerons par voir comment utiliser cette notion de référence pour la transmission d'arguments, ce qui constitue de loin son application principale. Par la suite (au paragraphe 13.2), nous ferons un point plus détaillé sur cette notion de référence, en particulier sur son utilisation pour la valeur de retour.

5.1 Exemple de transmission d'argument par référence

Le programme ci-dessous montre comment utiliser une transmission par référence dans notre précédente fonction *echange* :

```
#include <iostream>
using namespace std ;
void echange (int &, int &) ;
int main ()
{  int n=10, p=20 ;
   cout << "avant appel :   " << n << " " << p << "\n" ;
   echange (n, p) ;        // attention, ici pas de &n, &p
   cout << "apres appel :   " << n << " " << p << "\n" ;
}
void echange (int & a, int & b)
{  int c ;
   cout << "debut echange : " << a << " " << b << "\n" ;
   c = a ; a = b ; b = c ;
   cout << "fin echange   : " << a << " " << b << "\n" ;
}
```

```
avant appel :   10 20
début echange : 10 20
```

```
fin echange   : 20 10
après appel   : 20 10
```

Utilisation de la transmission d'argument par référence en C++

Dans l'instruction :

```
void echange (int & a, int & b) ;
```

la notation *int & a* signifie que a est une information de type *int* transmise par référence. Notez bien que, dans la fonction *echange*, on utilise simplement le symbole *a* pour désigner cette variable dont la fonction aura reçu effectivement l'adresse.

En Java

La notion de référence existe en Java, mais elle est entièrement transparente au programmeur. Plus précisément, les variables d'un type de base sont transmises par valeur, tandis que les objets sont transmis par référence. Il reste cependant possible de créer explicitement une copie d'un objet en utilisant une méthode appropriée dite de *clonage*.

5.2 Propriétés de la transmission par référence d'un argument

La transmission par référence d'un argument évite une recopie d'information, d'où un gain de temps d'exécution qui, s'il n'est guère sensible dans le cas de variables scalaires, pourra devenir apréciable dans le cas de gros agrégats ou de gros objets. En revanche, il entraîne un certain nombre de conséquences qui n'existaient pas dans le cas de la transmission par valeur.

5.2.1 Appel de la fonction

Dans l'appel d'une fonction recevant un argument par référence, l'argument effectif correspondant doit obligatoirement être une *lvalue* (une expression n'a pas d'adresse, une constante n'est pas modifiable). En outre, comme il s'agit de transmettre l'adresse de cette *lvalue*, il n'est pas possible d'en prévoir une quelconque conversion (même non dégradante), puisqu'il faudrait pour cela créer une nouvelle valeur, à une nouvelle adresse :

```
void f (int &) ;   // f reçoit la référence à un entier
  .....
const int n=15 ;
int q ;
f(q) ;       // OK
f(2*q+3) ;   // erreur : 2*q+3 n'est pas une lvalue
f(3) ;       // erreur : 3 n'est pas modifiable
f(n) ;       // erreur : n n'est pas modifiable
  .....
float x ;
f(x) ;       // erreur : x n'est pas de type int
```

La transmission par référence impose donc à un argument effectif d'être une lvalue du type prévu pour l'argument muet. Il existe cependant une exception dans le cas des arguments muets constants comme nous allons le voir maintenant.

5.2.2 Cas d'un argument muet constant

Il est possible de prévoir qu'un argument transmis par référence soit constant, comme dans cet en-tête :

```
void fct1 (const int & n) ;
```

Dans ce cas, la fonction *fct1* s'attend à recevoir l'adresse d'une constante et elle ne devra pas, dans sa définition, modifier la valeur de *n* ; dans le cas contraire, on obtiendra une erreur de compilation.

Cette fois, les appels suivants seront corrects :

```
const int c = 15 ;
   .....
fct1 (3) ;    // correct ici
fct1 (c) ;    // correct ici
```

L'acceptation de ces instructions se justifie par le fait que *fct* a prévu de recevoir une référence à quelque chose de constant ; le risque de modification évoqué précédemment n'existe donc plus.

Qui plus est, un appel tel que *fct1 (exp)* (*exp* désignant une expression quelconque) sera accepté quel que soit le type de *exp*. En effet, dans ce cas, il y aura création d'une variable temporaire (de type *int*) qui recevra le résultat de la conversion de *exp* en *int*. Par exemple :

```
void fct1 (const int &) ;
float x ;
   .....
fct1 (x) ;   // correct : f reçoit la référence à une variable temporaire
             //  contenant le résultat de la conversion de x en int
```

En définitive, l'utilisation de *const* pour un argument muet transmis par référence est lourde de conséquences. Certes, comme on s'y attend, cela amène le compilateur à verifier la constance de l'argument concerné au sein de la fonction. Mais, de surcroît, on autorise la création d'une copie de l'argument effectif (précédée d'une conversion) dès lors que ce dernier est constant et d'un type différent de celui attendu[1].

Cette remarque prendra encore plus d'acuité dans le cas où l'argument en question sera un objet volumineux.

5.2.3 Induction de risques indirects

Le choix du mode de transmission par référence est fait au moment de l'écriture de la fonction concernée. L'utilisateur de la fonction n'a plus à s'en soucier ensuite, si ce n'est au

1. Dans le cas d'une constante du même type, la norme laisse l'implémentation libre d'en faire ou non une copie. Généralement, la copie n'est faite que pour les constantes d'un type scalaire.

niveau de la déclaration du prototype de la fonction (d'ailleurs, ce prototype proviendra en général d'un fichier en-tête).

En contrepartie, l'emploi de la transmission par référence accroît les risques d'« effets de bord » non désirés. En effet, lorsqu'il appelle une fonction, l'utilisateur ne sait plus s'il transmet, au bout du compte, la valeur ou l'adresse d'un argument (la même notation pouvant désigner l'une ou l'autre des deux possibilités). Il risque donc de modifier une variable dont il pensait n'avoir transmis qu'une copie de la valeur.

Remarque

Nous verrons (au paragraphe 5 du chapitre 8) qu'il est également possible de « simuler » une transmission par référence, par le biais de pointeurs. Dans ce cas, l'utilisateur de la fonction devra transmettre explicitement des adresses : les risques évoqués précédemment disparaissent, en contrepartie d'une programmation plus délicate et plus risquée de la fonction elle-même.

6 Les variables globales

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.

En théorie, en C++, plusieurs fonctions (dont, bien entendu le programme principal *main*) peuvent partager des variables communes qu'on qualifie alors de **globales**. Il s'agit cependant là d'une **pratique risquée qu'il faudra éviter au maximum**. Nous vous la présenterons cependant ici car :

- vous risquez de rencontrer du code y recourant ;
- la notion de variable globale permet de mieux comprendre la différence entre classe d'allocation statique et classe d'allocation dynamique, laquelle prendra toute son importance dans un contexte objet ;
- dans une classe, les champs de données auront un comportement « global » pour les (seules) méthodes de cette classe.

6.1 Exemple d'utilisation de variables globales

Voyez l'exemple de programme ci après :

```
#include <iostream>
using namespace std ;
int i ;
void optimist (void) ;
```

```
int main ()
{   for (i=1 ; i<=5 ; i++) optimist() ;
}
void optimist(void)
{   cout << "il fait beau " << i << " fois\n" ;
}
```

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

Exemple d'utilisation de variable globale

La variable *i* a été déclarée en dehors de la fonction *main*. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à *i* des valeurs qui se trouvent utilisées par la fonction *optimist*.

Notez qu'ici la fonction *optmist* se contente d'utiliser la valeur de *i* mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsqu'on souhaite qu'une fonction modifie la valeur d'une variable, il est beaucoup plus judicieux d'en transmettre l'adresse en argument (soit par référence, comme nous avons appris à le faire, soit par pointeur, comme on le verra plus tard). Dans ce cas, l'appel de la fonction indique clairement quelles sont les seules variables susceptibles d'être modifiées.

6.2 La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
int main ()
{ ....
}
int n ;
float x ;
void fct1 (...)
{ ....
}
void fct2 (...)
{ ....
}
```

Les variables *n* et *x* sont accessibles aux fonctions *fct1* et *fct2*, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En pratique, s'il faut absolument recourir à des variables globales (par exemple, dans des codes critiques en temps d'exécution), on s'arrangera pour privilégier la lisibilité des codes en regroupant en début de programme[1] les déclarations de toutes ces variables globales.

6.3 La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.

De plus, ces variables se voient **initialisées à zéro**[2], avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

7 Les variables locales

À l'exception de l'exemple du paragraphe précédent, les variables que nous avions rencontrées jusqu'ici n'étaient pas des variables globales. Plus précisément, elles étaient définies au sein d'une fonction (qui pouvait être *main*). De telles variables sont dites **locales** à la fonction dans laquelle elles sont déclarées.

7.1 La portée des variables locales

Les variables locales ne sont connues du compilateur qu'à l'intérieur de la fonction où elles sont déclarées. **Leur portée est donc limitée à cette fonction.** Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions. Voyez cet exemple :

```
int n ;
int main ()
{  int p ;
   ....
}
void fct1 ()
{  int p ;
   int n ;
}
```

1. Ou dans un fichier en-tête séparé.

2. Cette notion de « zéro » sera précisée pour les pointeurs et pour les agrégats (tableaux, structures, objets...).

La variable *p* de *main* n'a aucun rapport avec la variable *p* de *fct1*. De même, la variable *n* de *fct1* n'a aucun rapport avec la variable globale *n*. En toute rigueur, si l'on souhaite utiliser dans *fct1* la variable globale *n*, on utilise l'opérateur dit « de résolution de portée » (::) en la nommant *::n*.

7.2 Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent.

Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant.

On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur ce mode de gestion de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 8) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

Informations complémentaires

Généralement, on utilise pour les variables automatiques une « pile » de type FIFO (*First In, First Out*) simulée dans une zone de mémoire contiguë. Lors de l'appel d'une fonction, on alloue de l'espace sur la pile pour :

- la valeur de retour ;
- les valeurs des arguments ou leurs références ;
- les différentes variables locales à la fonction.

Lors de la sortie de la fonction, ces différents emplacements sont libérés par la fonction elle-même, hormis celui de la valeur de retour qui sera libéré par la fonction appelante, après qu'elle l'aura utilisé.

La gestion de la pile se fait à l'aide d'un pointeur désignant le premier emplacement disponible. La libération d'un emplacement se fait par une simple modification de la valeur de ce pointeur ; l'emplacement libéré garde généralement sa valeur, de sorte que si, par une erreur de programmation, on y accède avant qu'il ait été alloué à une autre variable, on peut croire, à tort, qu'une variable locale est « rémanente »...

7.3 Les variables locales statiques

Il est possible de demander d'attribuer un emplacement permanent à une variable locale et de conserver ainsi sa valeur d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **static**. En voici un exemple :

```
#include <iostream>
using namespace std ;
void fct() ;
int main ()
{  int n ;
   for ( n=1 ; n<=5 ; n++)
      fct() ;
}
void fct()
{  static int i ;
   i++ ;
   cout << "appel numéro : " << i << "\n" ;
}
```

```
appel numéro : 1
appel numéro : 2
appel numéro : 3
appel numéro : 4
appel numéro : 5
```

Exemple d'utilisation de variable locale statique

La variable locale *i* a été déclarée de classe « statique ». On constate bien que sa valeur progresse de *1* à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**. Notez que nous aurions pu initialiser explicitement *i*, par exemple :

```
static int i = 3 ;
```

Dans ce cas, nos appels auraient porté des numéros allant de 4 à 8.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée *i* qui n'aurait alors aucun rapport avec la variable *i* de *fct.*

Remarque

Si *i* n'avait pas été déclarée avec l'attribut *static*, il se serait agit d'une variable locale usuelle, non rémanente et, de surcroît, non initialisée. Sa valeur aurait donc été aléatoire. De plus, ici, c'est toujours le même emplacement qui se serait trouvé alloué à *i* sur la pile,

de sorte qu'on afficherait toujours la même valeur, donnant l'illusion d'une certaine rémanence de *i* (qui toutefois, ici, ne serait pas incrémentée comme souhaité !).

7.4 Variables locales à un bloc

Comme nous l'avions déjà évoqué succinctement, C++ vous permet de déclarer des variables locales à un bloc. Leur portée est alors tout naturellement limitée à ce bloc ; leur emplacement est alloué à l'entrée dans le bloc et il disparaît à la sortie. Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc englobant.

```
void f()
{ int n ;        // n est accessible de tout le bloc constituant f
    .....
  for (...)
  { int p ;     // p n'est connue que dans le bloc de for
    int n ;     // n masque la variable n de portée "englobante"
      .....     // attention, on ne peut pas utiliser ::n ici qui
             .....     // désignerait une variable globale (inexistante ici)
  }
    .....
  { int p ;     // p n'est connue que dans ce bloc ; elle est allouée ici
    .....       // et n'a aucun rapport avec la variable p ci-dessus
  }             // et elle sera désallouée ici
    .....
}
```

Notez qu'on peut créer artificiellement un bloc, indépendamment d'une quelconque instruction structurée comme *if*, *for*. C'est le cas du deuxième bloc interne à notre fonction *f* ci-dessus.

D'autre part, nous avons déjà vu qu'il était possible d'effectuer une déclaration dans une instruction *for*, par exemple :

```
for (int i=2, j=4 ; ... ; ...)
{ // i et j sont considérées comme deux variables locales à ce bloc
}
```

On notera que, même si l'instruction *for* ne contient aucun bloc explicite, comme dans :

```
for (int i=1, j=1 ; i<4 ; i++) cout << i+j ;
```

les variables *i* et *j* ne seront plus connues par la suite, exactement comme si l'on avait écrit

```
for (int i=1, j=1 ; i<4 ; i++) { cout << i+j ; }
```

Informations complémentaires

En toute rigueur, il existe une classe d'allocation un peu particulière, à savoir la classe « registre » : toute variable entrant a priori dans la classe automatique peut être déclarée explicitement avec le qualificatif *register*. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un « registre » de la machine pour y ranger la variable : cela

peut amener quelques gains de temps d'exécution. Bien entendu, cette possibilité ne peut s'appliquer qu'aux variables d'un type simple.

7.5 Le cas des fonctions récursives

C++ autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects :

- récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle-même ;
- récursivité croisée : l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux fonctions).

Voici un exemple fort classique (d'ailleurs inefficace sur le plan du temps d'exécution) d'une fonction calculant une factorielle de manière récursive :

```
long fac (int n)
{
   if (n>1) return (fac(n-1)*n) ;
       else return(1) ;
}
```

Fonction récursive de calcul de factioriuelle

Il faut bien voir que chaque appel de *fac* entraîne une allocation d'espace pour les variables locales et pour son argument *n* (apparemment, *fct* ne comporte aucune variable locale ; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de *fac*, à l'intérieur de *fac*, provoque une telle allocation, sans que les emplacements précédents soient libérés.

Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction *return* que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

8 Initialisation des variables

Nous avons vu qu'il était possible d'initialiser explicitement une variable lors de sa déclaration. Ici, nous allons faire le point sur ces possibilités, lesquelles dépendent en fait de la classe d'allocation de la variable concernée.

8.1 Les variables de classe statique

Il s'agit des variables globales, ainsi que des variables locales déclarées avec l'attribut *static*. Ces variables sont permanentes. Elles sont initialisées une seule fois avant le début de l'exécution du programme. Elles peuvent être initialisées explicitement lors de leur déclaration, à l'aide de constantes ou d'**expressions constantes** (calculables par le compilateur) d'un **type compatible par affectation** avec celui de la variable, comme dans cet exemple (on notera que les conversions dégradantes du type *long* --> *float* sont acceptées, mais peu conseillées) :

```
void f (...)
{ const int NB = 5 ;
  static int limit = 2 *NB + 1 ;  // 2*NB+1 est une expression constante
  static short CTOT = 25 ;        // 25 de type int est converti en short int
  static float XMAX = 5 ;      // 5 de type int est converti en float
  static long YTOT = 9.7 ;     // 9.7 de type float est converti en long (déconseillé)
    .....
```

En l'absence d'initialisation explicite, ces variables seront initialisées à zéro.

8.2 Les variables de classe automatique

Il s'agit des variables locales à une fonction ou à un bloc. Ces variables ne sont **pas initialisées par défaut**. En revanche, comme les variables de classe statique, elles peuvent être initialisées explicitement lors de leur déclaration. Dans ce cas, la valeur initiale peut être fournie sous la forme d'une **expression quelconque (d'un type compatible par affectation)**, pour peu que sa valeur soit définie au moment de l'entrée dans la fonction correspondante (il peut s'agir de la fonction *main* !). N'oubliez pas que ces variables automatiques se trouvent alors initialisées à chaque appel de la fonction dans laquelle elles sont définies. En voici un cas d'école :

```
#include <iostream>
using namespace std ;
int n ;
void fct (int r) ;
int main ()
{  int p ;
  for (p=1 ; p<=5 ; p++)
    { n = 2*p ;
      fct(p) ;
    }
}
void fct(int r)
{
  int q=n, s=r*n ;
  cout << r << " " << q << " " << s << "\n" ;
}
```

```
1 2 2
2 4 8
3 6 18
4 8 32
5 10 50
```

Initialisation de variables de classe automatique

9 Les arguments par défaut

9.1 Exemples

Jusqu'ici, nos appels de fonction renfermaient autant d'arguments que la fonction en attendait effectivement. C++ permet de s'affranchir en partie de cette règle, grâce à un mécanisme d'attribution de valeurs par défaut à des arguments non fournis lors de l'appel.

Exemple 1

Considérez l'exemple suivant :

```
#include <iostream>
using namespace std ;
void fct (int, int=12) ; // prototype avec une valeur par défaut
int main ()
{  int n=10, p=20 ;
   fct (n, p) ;             // appel "normal"
   fct (n) ;                // appel avec un seul argument
                            // fct()  serait, ici, rejeté */
}
void fct (int a, int b)     // en-tête "habituelle"
{
   cout << "premier argument : " << a << "\n" ;
   cout << "second argument  : " << b << "\n" ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
```

Exemple de définition de valeur par défaut pour un argument

La déclaration de *fct*, ici dans la fonction *main*, est réalisée par le prototype :

```
void fct (int, int = 12) ;
```

La déclaration du second argument apparaît sous la forme :

```
int = 12
```

Celle-ci précise au compilateur que, en cas d'absence de ce second argument dans un éventuel appel de *fct*, il lui faudra « faire comme si » l'appel avait été effectué avec cette valeur.

Les deux appels de *fct* illustrent le phénomène. Notez qu'un appel tel que :

```
fct ( )
```

serait rejeté à la compilation puisque ici il n'était pas prévu de valeur par défaut pour le premier argument de *fct*.

Exemple 2

Voici un second exemple, dans lequel nous avons prévu des valeurs par défaut pour tous les arguments de *fct* :

```
#include <iostream>
using namespace std ;
void fct (int=0, int=12) ; // prototype avec deux valeurs par défaut
int main ()
{  int n=10, p=20 ;
   fct (n, p) ;               // appel "normal"
   fct (n) ;                  // appel avec un seul argument
   fct () ;                   // appel sans argument
}
void fct (int a, int b)     // en-tête "habituelle"
{  cout << "premier argument : " << a << "\n" ;
   cout << "second argument  : " << b << "\n" ;
}
```

```
premier argument : 10
second argument  : 20
premier argument : 10
second argument  : 12
premier argument : 0
second argument  : 12
```

Exemple de définition de valeurs par défaut pour plusieurs arguments

9.2 Les propriétés des arguments par défaut

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.

Par exemple, une déclaration telle que :

```
float fexple (int = 5, long, int = 3) ;
```

est interdite. En fait, une telle interdiction relève du pur bon sens. En effet, si cette déclaration était acceptée, l'appel suivant :

```
fexple (10, 20) ;
```

pourrait être interprété aussi bien comme :

```
fexple (5, 10, 20) ;
```

que comme :

```
fexple (10, 20, 3) ;
```

Notez bien que le mécanisme proposé par C++ revient à **fixer les valeurs par défaut dans la déclaration de la fonction et non dans sa définition**. Autrement dit, ce n'est pas le « concepteur » de la fonction qui décide des valeurs par défaut, mais l'utilisateur. Une conséquence immédiate de cette particularité est que les arguments soumis à ce mécanisme et les valeurs correspondantes peuvent varier d'une utilisation à une autre ; en pratique toutefois, ce point ne sera guère exploité, ne serait-ce que parce que les déclarations de fonctions sont en général « figées » une fois pour toutes, dans un fichier en-tête.

Nous verrons que les arguments par défaut se révéleront particulièrement précieux lorsqu'il s'agira de fabriquer ce que l'on nomme le « constructeur d'une classe ».

Remarque

Les valeurs par défaut ne sont pas nécessairement des expressions constantes. Elles ne peuvent toutefois pas faire intervenir de variables locales[1].

En Java

Les arguments par défaut n'existent pas en Java.

10 Surdéfinition de fonctions

D'une manière générale, on parle de « surdéfinition »[2] lorsqu'un même symbole possède plusieurs significations différentes, le choix de l'une des significations se faisant en fonction du contexte. C'est ainsi que la plupart des langages évolués utilisent la surdéfinition d'un certain nombre d'opérateurs. Par exemple, dans une expression telle que :

```
a + b
```

la signification du + dépend du type des opérandes *a* et *b* ; suivant les cas, il pourra s'agir d'une addition d'entiers ou d'une addition de flottants. De même, le symbole * peut désigner, suivant le contexte, une multiplication d'entiers, de flottants (ou, commme nous le verrons lorsque nous étudierons les pointeurs, une indirection).

Un des grands atouts de C++ est de permettre la surdéfinition de la plupart des opérateurs (lorsqu'ils sont associés à la notion de classe). Lorsque nous étudierons cet aspect, nous verrons qu'il repose en fait sur la surdéfinition de fonctions. C'est cette dernière possibilité que nous proposons d'étudier ici pour elle-même.

---

1. Ni la valeur *this* pour les fonctions membres (*this* sera étudié au chapitre 11).

2. De *overloading*, parfois traduit par « surcharge ».

Pour pouvoir employer plusieurs fonctions de même nom, il faut bien sûr un critère (autre que le nom) permettant de choisir la bonne fonction. En C++, ce choix est basé (comme pour les opérateurs cités précédemment en exemple) sur le type des arguments. Nous commencerons par vous présenter un exemple complet montrant comment mettre en œuvre la surdéfinition de fonctions. Nous examinerons ensuite différentes situations d'appel d'une fonction surdéfinie avant d'étudier les règles détaillées qui président au choix de la « bonne fonction ».

10.1 Mise en œuvre de la surdéfinition de fonctions

Nous allons définir et utiliser deux fonctions nommées *sosie*. La première possédera un argument de type *int*, la seconde un argument de type *double*, ce qui les différencie bien l'une de l'autre. Pour que l'exécution du programme montre clairement la fonction effectivement appelée, nous introduisons dans chacune une instruction d'affichage appropriée. Dans le programme d'essai, nous nous contentons d'appeler successivement la fonction surdéfinie *sosie*, une première fois avec un argument de type *int*, une seconde fois avec un argument de type *double*.

```
#include <iostream>
using namespace std ;
void sosie (int) ;              // les prototypes
void sosie (double) ;
int main ()                     // le programme de test
{  int n=5 ;
   double x=2.5 ;
   sosie (n) ;
   sosie (x) ;
}
void sosie (int a)              // la première fonction
{  cout << "sosie numero I   a = " << a << "\n" ;
}
void sosie (double a)           // la deuxième fonction
{   cout << "sosie numero II  a = " << a << "\n" ;
}
```

```
sosie numero I   a = 5
sosie numero II  a = 2.5
```

Exemple de surdéfinition de la fonction sosie

Vous constatez que le compilateur a bien mis en place l'appel de la « bonne fonction » *sosie*, au vu de la liste d'arguments (ici réduite à un seul).

10.2 Exemples de choix d'une fonction surdéfinie

Notre précédent exemple était simple, dans la mesure où nous appelions toujours la fonction *sosie* avec un argument ayant **exactement** l'un des types prévus dans les prototypes (*int* ou *double*). On peut se demander ce qui se produirait si nous l'appelions par exemple avec un argument de type *char* ou *long*, ou si l'on avait affaire à des fonctions comportant plusieurs arguments...

Avant de présenter les règles de détermination d'une fonction surdéfinie, examinons tout d'abord quelques situations assez intuitives.

Exemple 1

```
void sosie (int) ;       // sosie I
void sosie (double) ;    // sosie II
char c ; float y ;
   .....
sosie(c) ;   // appelle sosie I, après conversion de c en int
sosie(y) ;   // appelle sosie II, après conversion de y en double
sosie('d') ; // appelle sosie I, après conversion de 'd' en int
```

Exemple 2

```
void essai (int, double) ;    // essai I
void essai (double, int) ;    // essai II
int n, p ; double z ; char c ;
   .....
essai(n,z) ; // appelle essai I
essai(c,z) ; // appelle essai I, après conversion de c en int
essai(n,p) ; // erreur de compilation,
```

Compte tenu de son ambiguïté, le dernier appel conduit à une erreur de compilation. En effet, deux possibilités existent ici : convertir *p* en *double* sans modifier *n* et appeler *essai I* ou, au contraire, convertir *n* en *double* sans modifier *p* et appeler *essai II*.

Exemple 3

```
void test (int n=0, double x=0) ;     // test I
void test (double y=0, int p=0) ;     // test II
int n ; double z ;
   .....
test(n,z) ; // appelle test I
test(z,n) ; // appelle test II
test(n) ;   // appelle test I
test(z) ;   // appelle test II
test() ;    // erreur de compilation, compte tenu de l'ambiguïté.
```

Exemple 4

Avec ces déclarations :

```
void truc (int) ;         // truc I
void truc (const int) ;   // truc II
```

vous obtiendrez une erreur de compilation. En effet, C++ n'a pas prévu de distinguer *int* de *const int*. Cela se justifie par le fait que, les deux fonctions *truc* recevant une copie de l'information à traiter, il n'y a aucun risque de modifier la valeur originale. Notez bien qu'ici l'erreur tient à la seule présence des déclarations de *truc*, indépendamment d'un appel quelconque.

Exemple 5

En revanche, considérez maintenant ces déclarations :

```
void chose (int &) ;        // chose I
void chose (const int &) ;  // chose II
int n = 3 ;
const int p = 5 ;
   .....
chose (n) ;  // appelle chose I
chose (p) ;  // appelle chose II
```

Cette fois, la distinction entre *int &* et *const int &* est justifiée. En effet, on peut très bien imaginer que *chose I* modifie la valeur de la *lvalue* dont elle reçoit la référence, tandis que *chose II* n'en fait rien.

Exemple 6

L'exemple précédent a montré comment on pouvait distinguer deux fonctions agissant, l'une sur une référence, l'autre sur une référence constante. Mais l'utilisation de références possède des conséquences plus subtiles, comme le montrent ces exemples (revoyez éventuellement le paragraphe 5.2.2) :

```
void chose (int &) ;       // chose I
void chose (const int &)   // chose II
int n :
float x ;
   .....
chose (n) ; // appelle chose I
chose (2) ; // appelle chose II, après copie éventuelle de 2 dans un entier[1]
            // temporaire dont la référence sera transmise à chose
chose (x) ; // appelle chose II, après conversion de la valeur de x en un
            // entier temporaire dont la référence sera transmise à chose
```

Remarques

1 En dehors de la situation examinée dans l'exemple 5, on notera que le mode de transmission (référence ou valeur) n'intervient pas dans le choix d'une fonction surdéfinie. Par exemple, les déclarations suivantes conduiraient à une erreur de compilation due à leur ambiguïté (indépendamment de tout appel de *chose*) :

```
void chose (int &) ;
void chose (int) ;
```

1. Comme l'autorise la norme, l'implémentation est libre de faire ou non une copie dans ce cas.

2 Nous venons de voir comment *int &* se distingue de *const int &*. Lorsque nous étudierons les pointeurs, nous verrons (paragraphe 9 du chapitre 8) qu'il existe une distinction comparable entre un pointeur sur une variable (*int* *) et un pointeur sur une constante (*const int* *).

En Java

La surdéfinition des fonctions existe en Java. Mais les règles de recherche de la bonne fonction sont beaucoup plus simples qu'en C++, car il existe peu de possibilités de conversions implicites.

10.3 Règles de recherche d'une fonction surdéfinie

Pour l'instant, nous vous présenterons plutôt la philosophie générale, ce qui sera suffisant pour l'étude des chapitres suivants. Au cours de cet ouvrage, nous serons amenés à vous apporter des informations complémentaires. De plus, l'ensemble de toutes ces règles sont reprises en Annexe A.

10.3.1 Cas des fonctions à un argument

Le compilateur recherche la « meilleure correspondance » possible. Bien entendu, pour pouvoir définir ce qu'est cette meilleure correspondance, il faut qu'il dispose d'un critère d'évaluation. Pour ce faire, il est prévu différents niveaux de correspondance :

1) ***Correspondance exacte*** : on distingue bien les uns des autres les différents types de base, en tenant compte de leur éventuel attribut de signe[1] ; de plus, comme on l'a vu dans les exemples précédents, l'attribut *const* peut intervenir dans le cas de références (il en ira de même pour les pointeurs).

2) ***Correspondance avec promotions numériques***, c'est-à-dire essentiellement :

char et *short* –> *int*

float –> *double*

Rappelons qu'un argument transmis par référence ne peut être soumis à aucune conversion, sauf lorsqu'il s'agit de la référence à une constante.

3) ***Conversions dites standard*** : il s'agit des conversions légales en C++, c'est-à-dire de celles qui peuvent être imposées par une affectation (sans opérateur de *cast*) ; cette fois, il peut s'agir de conversions dégradantes puisque, notamment, toute conversion d'un type numérique en un autre type numérique est acceptée.

1. Attention : en C++, *char* est différent de *signed char* et de *unsigned char*.

4) ***D'autres niveaux*** sont prévus ; en particulier on pourra faire intervenir ce que l'on nomme des « conversions définies par l'utilisateur » (C.D.U.), qui ne seront étudiées qu'au chapitre 16.

   Là encore, un argument transmis par référence ne pourra être soumis à aucune conversion, sauf s'il s'agit d'une référence à une constante.

La recherche s'arrête au premier niveau ayant permis de trouver une correspondance, qui doit alors être unique. Si plusieurs fonctions conviennent au même niveau de correspondance, il y a erreur de compilation due à l'ambiguïté rencontrée. Bien entendu, si aucune fonction ne convient à aucun niveau, il y a aussi erreur de compilation.

10.3.2 Cas des fonctions à plusieurs arguments

L'idée générale est qu'il doit se dégager une fonction « meilleure » que toutes les autres. Pour ce faire, le compilateur sélectionne, **pour chaque argument**, la ou les fonctions qui réalisent la meilleure correspondance (au sens de la hiérarchie définie ci-dessus). Parmi l'ensemble des fonctions ainsi sélectionnées, il choisit celle (si elle existe et si elle est unique) qui réalise, pour chaque argument, une correspondance au moins égale à celle de toutes les autres fonctions[1].

Si plusieurs fonctions conviennent, là encore, on aura une erreur de compilation due à l'ambiguïté rencontrée. De même, si aucune fonction ne convient, il y aura erreur de compilation.

Notez que les fonctions comportant un ou plusieurs arguments par défaut sont traitées comme si plusieurs fonctions différentes avaient été définies avec un nombre croissant d'arguments.

11 Les arguments variables en nombre

Dans tous nos précédents exemples, le nombre d'arguments fournis au cours de l'appel d'une fonction était prévu lors de l'écriture de cette fonction.

Or, dans certaines circonstances, on peut souhaiter réaliser une fonction capable de recevoir un nombre d'arguments susceptible de varier d'un appel à un autre.

En C++, on y parvient à l'aide des fonctions particulières *va_start* et *va_arg* (dont le prototype figure dans le fichier en-tête *cstdarg*). La seule contrainte à respecter est que la fonction doit posséder au moins un argument fixe (c'est-à-dire toujours présent). En effet, comme nous allons le voir, c'est le dernier argument fixe qui permet, en quelque sorte, d'initialiser le parcours de la liste d'arguments.

1. Ce qui revient à dire qu'il considère l'intersection des ensembles constitués des fonctions réalisant la meilleure correspondance pour chacun des arguments.

11.1 Premier exemple

Voici un premier exemple de fonction à arguments variables : les deux premiers arguments sont fixes, l'un étant de type *int*, l'autre de type *char*. Les arguments suivants, de type *int*, sont en nombre quelconque et l'on a supposé que le dernier d'entre eux était *-1*. Cette dernière valeur sert donc, en quelque sorte, de « sentinelle ». Par souci de simplification, nous nous sommes contentés, dans cette fonction, de lister les valeurs de ces différents arguments (fixes ou variables), à l'exception du dernier.

```
#include <iostream>
#include <cstdarg>       // pour va_arg et va_list
using namespace std ;
void essai (int, char, ...) ;
int main ()
{  cout << "premier essai\n" ;
   essai (125, 'a', 15, 30, 40, -1) ;
   cout << "deuxieme essai\n" ;
   essai (6264, 'S', -1) ;
}
void essai (int par1, char  par2, ...)
{  va_list adpar ;
   int parv ;
   cout << "premier parametre : " << par1 << "\n" ;
   cout << "second parametre  : " << par2 << "\n" ;
   va_start (adpar, par2) ;
   while ( (parv = va_arg (adpar, int) ) != -1)
       cout << "argument variable : " << parv << "\n" ;
}
```

```
premier essai
premier parametre : 125
second parametre  : a
argument variable : 15
argument variable : 30
argument variable : 40
deuxieme essai
premier parametre : 6264
second parametre  : S
```

Arguments en nombre variable, délimités par une sentinelle

Vous constatez la présence, dans l'en-tête de la fonction *essai*, des deux noms des paramètres fixes *par1* et *par2*, déclarés de manière classique ; les trois points servent à spécifier au compilateur l'existence de paramètres en nombre variable.

La déclaration :

```
va_list adpar ;
```

précise que *adpar* est un identificateur de liste variable. C'est lui qui nous servira à récupérer, les uns après les autres, les différents arguments variables.

Comme à l'accoutumée, une telle déclaration n'attribue aucune valeur à *adpar*. C'est effectivement la fonction *va_start* qui va permettre de l'initialiser à l'adresse du paramètre variable. Notez bien que cette dernière est déterminée par *va_start* à partir de la connaissance du nom du dernier paramètre fixe.

Le rôle de la fonction *va_arg* est double :

- d'une part, elle fournit comme résultat la valeur trouvée à l'adresse courante fournie par *adpar* (son premier argument), suivant le type indiqué par son second argument (ici *int*) ;
- d'autre part, elle incrémente l'adresse contenue dans *adpar*, de manière que celle-ci pointe alors sur l'argument variable suivant.

Ici, une instruction *while* nous permet de récupérer les différents arguments variables, sachant que le dernier a pour valeur *-1*.

Enfin, la norme ANSI prévoit que la macro *va_end* doit être appelée avant de sortir de la fonction concernée. Si vous manquez à cette règle, vous courez le risque de voir un prochain appel à la fonction conduire à un mauvais fonctionnement du programme.

Remarque

Les arguments variables peuvent être de types différents, à condition toutefois que la fonction soit en mesure de les connaître, d'une façon ou d'une autre.

Informations complémentaires

En toute rigueur, *va_start* et *va_arg* ne sont pas de véritables fonctions, mais des « macros » ; cette distinction n'a que peu d'incidence sur leur utilisation effective. Les macros, beaucoup moins utilisées en C++ qu'en C, seront présentées paragraphe 2.2 du chapitre 31.

11.2 Second exemple

La gestion de la fin de la liste des arguments variables est laissée au bon soin de l'utilisateur ; en effet, il n'existe aucune fonction permettant de connaître le nombre effectif de ces arguments.

Cette gestion peut se faire :

- par sentinelle, comme dans notre précédent exemple ;
- par transmission, en argument fixe, du nombre d'arguments variables.

Voici un exemple de fonction utilisant cette seconde technique. Nous n'y avons pas prévu d'autre argument fixe que celui spécifiant le nombre d'arguments variables.

```
#include <iostream>
#include <cstdarg>
using namespace std ;
void essai (int, ...) ;
int main ()
{  cout << "premier essai\n" ;
   essai (3, 15, 30, 40) ;
   cout << "\ndeuxieme essai\n" ;
   essai (0) ;
}
void essai (int nbpar, ...)
{  va_list adpar ;
   int parv, i ;
   cout << "nombre de valeurs : " << nbpar << "\n" ;
   va_start (adpar, nbpar) ;
   for (i=1 ; i<=nbpar ; i++)
     {  parv = va_arg (adpar, int) ;
        cout << "argument variable : " << parv << "\n" ;
     }
}
```

```
premier essai
nombre de valeurs : 3
argument variable : 15
argument variable : 30
argument variable : 40

deuxieme essai
nombre de valeurs : 0
```

Arguments variables dont le nombre est fourni en argument fixe

12 La conséquence de la compilation séparée

12.1 Compilation séparée et prototypes

Nous avons déjà été amenés à utiliser des fonctions prédéfinies telles que *sqrt*. Pour ce faire, nous incorporions le fichier en-tête *cmath* qui contient les déclarations des fonctions mathématiques telles que *sqrt*. Nous savons que le module objet correspondant à cette fonction a déjà été compilé, qu'il figure dans une bibliothèque et qu'il sera incorporé par l'éditeur de liens pour créer le programme exécutable.

Cette démarche, dans laquelle on réunit plusieurs modules objets compilés de façon indépendante l'une de l'autre (ici votre *main* d'une part, *sqrt* d'autre part) peut s'appliquer à des fichiers sources conçus par l'utilisateur. On parle alors de « compilation séparée ». Par exemple, vous pourriez tout à fait placer dans des fichiers sources différents les fonctions que nous

avons été amenés à créer auparavant (*fexple*, *echange*, *optimist*, *fct*) et les compiler séparément. Dans ce cas, la définition de la fonction ne figure plus dans le programme l'utilisant. On n'y trouvera que sa déclaration. Si certaines des fonctions que vous développez doivent être utilisées par plusieurs programmes, vous serez probablement amené à prévoir un fichier en-tête (nommé par exemple *MesFonc*) en contenant les déclarations, de façon à éviter d'éventuelles erreurs d'écriture (ou plutôt un fichier en-tête par groupe de fonctions ayant un rapport entre elles). Généralement, un tel fichier portera l'extension *.h*. Comme pour les fichiers en-têtes prédéfinis, vous incorporerez votre fichier en-tête par une directive *#include*. Il faut cependant savoir que la syntaxe en est légèrement modifiée pour les fichiers de l'utilisateur (utilisation de *"..."* au lieu de <...>) :

```
include "MesFonc.h"
```

Généralement, l'inclusion d'un fichier en-tête se fait à un niveau global comme dans ce schéma :

```
#include "MesFonc.h"  // Attention : "MesFonc.h" et non <MesFonc.h>
                      //  pour un fichier en-tête utilisateur
int main ()
{ ...                 // ici, on dispose des déclarations figurant dans MesFonc.h
}
void f()
{                     // ici, également
}
```

Bien que cela soit peu utilisé, il est possible, en théorie, d'effectuer une inclusion à un niveau local, comme dans ce schéma :

```
int main ()
{ #include "MesFonc.h"
  ...                 // ici, on dispose des déclarations figurant dans MesFonc.h
}
void f()
{                     // ici, non
}
```

12.2 Fonction manquante lors de l'édition de liens

Compte tenu de ces possibilités de compilation séparée, on voit qu'il est tout à fait possible d'écrire un programme dans lequel on a omis la définition d'une fonction (pour peu qu'elle soit correctement déclarée) :

```
int main ()
{ void f() ;    // déclaration de f
  .....
  f() ;         // utilisation de f
  .....
}
```

La compilation se déroulera sans problème. En revanche, si lors de l'édition de liens, la définition de *f* n'est trouvée dans aucun module objet (y compris dans ceux constituant la bibliothèque standard), on obtiendra une erreur.

Remarque

Ne confondez pas les fichiers en-tête qui ne contiennent que les déclarations de fonctions avec les modules objets qui, quant à eux, contiennent bien le code exécutable correspondant à leur définition. L'un ne remplace pas l'autre. Certes, la confusion peut être entretenue par les fonctions de la bibliothèque standard dont on a l'impression qu'il suffit de citer les fichiers en-têtes contenant leur déclaration pour en disposer. En fait, le travail de recherche de l'éditeur de liens est totalement indépendant de la compilation et n'a aucun rapport avec l'éventuelle inclusion de fichiers en-tête. Vous pourriez par exemple **utiliser *sqrt*, sans incorporer *cmath***, pour peu que vous en fournissiez la déclaration.

12.3 Le mécanisme de la surdéfinition de fonctions

Dans notre étude de la surdéfinition des fonctions du paragraphe 10, nous avions examiné la manière dont le compilateur faisait le choix de la « bonne fonction », en raisonnant sur un seul fichier source à la fois. Mais on voit maintenant qu'il est tout à fait envisageable :

- de compiler dans un premier temps un fichier source contenant les différentes définitions d'une fonction (telle que *sosie* ou *chose* dans nos précédents exemples) ; on peut même éclater ces surdéfinitions dans plusieurs fichiers sources ;
- d'utiliser ultérieurement ces fonctions dans un autre fichier source en nous contentant d'en fournir les prototypes.

Or, pour que cela soit possible, l'éditeur de liens doit être en mesure d'effectuer le lien entre le choix opéré par le compilateur et la « bonne fonction » figurant dans un autre module objet. Cette reconnaissance est fondée sur la modification, par le compilateur, des noms « externes » des fonctions ; celui-ci fabrique un nouveau nom fondé d'une part sur le nom interne de la fonction, d'autre part sur le nombre et la nature de ses arguments.

Il est très important de noter que ce mécanisme s'applique à toutes les fonctions, qu'elles soient surdéfinies ou non (il est impossible de savoir si une fonction compilée dans un fichier source sera surdéfinie dans un autre). On voit donc qu'un problème se pose, dès que l'on souhaite utiliser dans un programme C++ une fonction écrite et compilée en C (ou dans un autre langage utilisant les mêmes conventions d'appel de fonction, notamment l'assembleur ou le Fortran). En effet, une telle fonction n'aura pas son nom modifié suivant le mécanisme évoqué. Une solution existe toutefois : déclarer une telle fonction en faisant précéder son prototype de la mention *extern "C"*. Par exemple, si nous avons écrit et compilé en C une fonction d'en-tête :

```
double fct (int n, char c) ;
```

et que nous souhaitons l'utiliser dans un programme C++, il nous suffira de fournir son prototype de la façon suivante :

```
extern "C" double fct (int, char) ;
```

Remarques

1 Il existe une forme « collective » de la déclaration *extern,* qui se présente ainsi :

```
extern "C"
{ void exple (int) ;
  double chose (int, char, float) ;
  .....
} ;
```

2 Le problème évoqué pour les fonctions C (assembleur ou Fortran) se pose, a priori, pour toutes les fonctions de la bibliothèque standard C que l'on réutilise en C++. En fait, dans la plupart des environnements, cet aspect est automatiquement pris en charge au niveau des fichiers en-tête correspondants (ils contiennent des déclarations *extern* conditionnelles).

3 Il est possible d'employer, au sein d'un même programme C++, une fonction C (assembleur ou Fortran) et une ou plusieurs autres fonctions C++ de même nom (mais d'arguments différents). Par exemple, nous pouvons utiliser dans un programme C++ la fonction *fct* précédente et deux fonctions C++ d'en-tête :

```
void fct (double x)
void fct (float y)
```

en procédant ainsi :

```
extern "C" void fct (int) ;
void fct (double) ;
void fct (float) ;
```

Suivant la nature de l'argument d'appel de *fct*, il y aura bien appel de l'une des trois fonctions *fct*. Notez qu'il n'est pas possible de mentionner plusieurs fonctions C de nom *fct*.

12.4 Compilation séparée et variables globales

N.B. Ce paragraphe a surtout un intérêt si vous devez exploiter du code utilisant cette technique déconseillée de variables globales. Il peut également servir à distinguer la notion de portée (compilation) de celle de lien (édition de liens).

12.4.1 La portée d'une variable globale – la déclaration extern

A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source :

```
source 1                source 2
 int x ;                 void fct2()
 int main ()                  {
  {                          .....
     .....                  }
  }                          void fct3()
  void fct1()                {
  {                             .....
     .....                   }
  }
```

Il ne semble pas possible, dans les fonctions *fct2* et *fct3* de faire référence à la variable globale *x* déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).

En fait, C++ prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé *extern*. Ainsi, en faisant précéder notre second fichier source de la déclaration :

```
extern int x ;
```

il devient possible de mentionner la variable globale *x* (déclarée dans le premier fichier source) dans les fonctions *fct2* et *fct3*.

Remarque

Cette déclaration *extern* n'effectue **pas de réservation d'emplacement de variable**. Elle ne fait que préciser que la variable globale *x* est définie par ailleurs et elle en indique le type.

12.4.2 Les variables globales et l'édition de liens

Supposons que nous ayons compilé les deux fichiers source précédents et voyons d'un peu plus près comment l'éditeur de liens est en mesure de rassembler correctement les deux modules objets ainsi obtenus. En particulier, examinons comment il peut faire correspondre au symbole *x* du second fichier source l'adresse effective de la variable *x* définie dans le premier.

D'une part, après compilation du premier fichier source, on trouve, dans le module objet correspondant, une indication associant le symbole *x* et son adresse dans le module objet. Autrement dit, contrairement à ce qui se produit pour les variables locales, pour lesquelles ne subsiste aucune trace du nom après compilation, le nom des variables globales continue à exister au niveau des modules objets. On retrouve là un mécanisme analogue à ce qui se passe pour les noms de fonctions, lesquels doivent bien subsister pour que l'éditeur de liens soit en mesure de retrouver les modules objets correspondants.

D'autre part, après compilation du second fichier source, on trouve, dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom *x* provient de l'extérieur et qu'il faudra en fournir l'adresse effective.

Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable *x* et de la reporter dans le second module objet.

Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objets correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

12.4.3 Les variables globales cachées – la déclaration static

Il est possible de « œacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle » au lieu de « cacher » ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **static** comme dans cet exemple :

```
static int a ;
int main ()
{
   .....
}
void fct()
{
   .....
}
```

Sans la déclaration *static*, *a* serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de *a* ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par *extern*. Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source.

13 Compléments sur les références

L'essentiel concernant la notion de référence a été étudié au paragraphe 5. Ici, nous vous fournissons des informations complémentaires relatives à :

- la transmission par référence d'une « valeur de retour » ; ce point n'interviendra qu'à partir du chapitre consacré à la surdéfinition des opérateurs ;
- l'aspect général de la notion de référence, qui dépasse celle d'argument ou de valeur de retour ; il s'agit d'éléments peu usités qui peuvent très bien être omis dans un premier temps.

13.1 Transmission par référence d'une valeur de retour

N.B. L'étude de ce paragraphe peut être différée jusqu'à celle du chapitre sur la surdéfinition des opérateurs.

13.1.1 Introduction

Le mécanisme que nous avons exposé pour la transmission des arguments par référence s'applique à la valeur de retour d'une fonction. Considérons :

```
int & f ()
{ .....
  return n ;   // on suppose ici n de type int
}
```

Un appel de *f* provoquera la transmission en retour non plus d'une valeur, mais de la référence (adresse) de *n*. Là encore, on évite une recopie, ce qui entraîne un gain de temps d'exécution qui deviendra intéressant avec de gros objets. Cependant, cette fois, on voit que *n* ne peut pas être une variable locale à *f* car, sinon, elle serait détruite à la sortie de *f* et l'on récupérerait dans la fonction appelante, une adresse à quelque chose qui n'existe plus[1]. En revanche, on pourra ainsi fournir en valeur de retour la référence à un objet créé dynamiquement comme nous apprendrons à le faire plus tard. On pourra aussi renvoyer une référence qu'on aura reçue en argument (nous en rencontrerons un exemple ci-dessous).

13.1.2 Conséquences dans la définition de la fonction

Puisqu'il s'agit d'une transmission d'adresse, la valeur de retour mentionnée dans l'instruction *return* ne peut être qu'une *lvalue*. Il ne pourra pas s'agir d'une expression ou d'une constante (avec une exception cependant comme nous le verrons au paragraphe 13.1.5).

Toujours puisqu'il s'agit d'une adresse, toute conversion de cette valeur de retour s'avère impossible (même s'il s'agit d'une conversion non dégradante). La *lvalue* figurant dans l'instruction *return* doit donc être du type exact prévu dans l'en-tête.

13.1.3 Conséquences dans l'utilisation de la fonction

Dès lors qu'une fonction renvoie une référence, il devient possible d'utiliser son appel comme une *lvalue*. Voyez cet exemple :

```
int & f () ;
int n ;
float x ;
   .....
f() = 2 * n + 5 ;   // à la référence fournie par f, on range la valeur
                    // de l'expression 2*n+5, de type int
f() = x ;           // à la référence fournie par f, on range la valeur
                    // de x, après conversion en int
```

1. Cette erreur s'apparente à celle due à la transmission en valeur de retour d'un pointeur sur une variable locale (situation que nous rencontrerons plus tard). Elle est encore plus difficile à détecter dans la mesure où le seul moment où l'on peut utiliser la référence concernée est l'appel lui-même (alors qu'un pointeur peut être utilisé à volonté...). Dans un environnement ne modifiant pas la valeur d'une variable lors de sa « destruction », aucune erreur ne se manifeste ; ce n'est que lors du portage dans un environnement ayant un comportement différent que les choses deviennent catastrophiques.

Cette propriété sera largement exploitée lorsqu'il s'agira de surdéfinir un opérateur (en fait, une fonction) dont la nature imposera de fournir une *lvalue* en résultat. Ce sera précisément le cas de l'opérateur [].

En revanche, contrairement à ce qui se produisait pour les arguments transmis par référence, aucune contrainte d'exactitude de type ne pèse sur l'utilisation d'une valeur de retour fournie par référence, car il reste toujours possible de la soumettre à une conversion avant de l'utiliser :

```
inf & f () ;
float x ;
   .....
x = f() ;   // OK : on convertira en int la valeur située à la référence
            //  reçue en retour de f
```

Nous verrons cependant au paragraphe 13.1.5 qu'il n'en va plus de même lorsque la valeur de retour est une référence à une constante.

13.1.4 Exemple

Voici un exemple, un peu artificiel, d'une fonction recevant deux références d'entiers et renvoyant l'une d'entre elles. Un indicateur booléen permet de choisir une fois la première, une fois la seconde.

```
#include <iostream>
using namespace std ;
int & alterne (int &, int &) ;
int main ()
{ int n=1, p=3, q=5 ;
  alterne (n, p ) = 0 ;
  cout << "n = " << n << " p = " << p << "\n" ;
  alterne (p, q) = 12 ;
  cout << "p = " << p << " q = " << q << "\n" ;
}
int & alterne (int & n1, int & n2)
{ static bool indic = true ;
  if (indic) { indic = false ; return n1 ; }
        else { indic = true ;  return n2 ; }
}
```

```
n = 0 p = 3
p = 3 q = 12
```

Exemple de transmission d'une valeur de retour par référence

13.1.5 Valeur de retour constante

Il est possible de prévoir qu'une fonction fournisse par référence une valeur de retour constante, comme dans :

```
const int & f2(.....)
```

Dans ce cas, dans la définition de la fonction, l'instruction *return* pourra toujours mentionner une *lvalue* comme auparavant, mais cette fois, il deviendra également possible d'y faire figurer une constante. Dans ce cas, la norme a en effet prévu qu'on renvoie la référence d'une **copie temporaire de cette constante**. Qui plus est, puisqu'on crée une copie, une éventuelle conversion redevient possible.

```
const int & f2 (.....)    // cette fois l'en-tête de f2 mentionne
                          // la référence à un entier constant
{ .....
  return 5 ;   // OK : on renvoie la référence à une copie temporaire de 5
  return n ;   // OK
  return x ;   // OK : on renvoie la référence à un int temporaire
               // obtenu par conversion de la valeur de x
}
```

En revanche, dans l'appel de la fonction, la valeur de retour (référence à une constante temporaire) ne pourra plus être utilisée comme une *lvalue* ni faire l'objet de conversion :

```
const int & f2 () ;
int n ;
float x ;
   .....
f() = 2 * n + 5 ;  // erreur : f() n'est pas une lvalue
f() = x ;          // idem
```

13.2 La référence d'une manière générale

N.B. Ce paragraphe peut être ignoré dans un premier temps.

L'essentiel concernant la notion de référence réside dans la transmission d'arguments ou de valeur de retour. Cependant, en toute rigueur, la notion de référence peut intervenir en dehors de la notion d'argument ou de valeur de retour. C'est ce que nous allons examiner ici.

13.2.1 La notion de référence est plus générale que celle d'argument

D'une manière générale, il est possible de déclarer un identificateur comme référence d'une autre variable. Considérez, par exemple, ces instructions :

```
int n ;
int & p = n ;
```

La seconde signifie que *p* est une référence à la variable *n*. Ainsi, dans la suite, *n* et *p* désigneront le même emplacement mémoire. Par exemple, avec :

```
n = 3 ;
cout << p ;
```

nous obtiendrons la valeur 3.

Remarque

Il ne sera pas possible de définir des pointeurs sur des références, ni des tableaux de références.

13.2.2 Initialisation de référence

La déclaration :

```
int & p = n ;
```

est en fait une déclaration de référence (ici *p*) accompagnée d'une initialisation (à la référence de *n*). D'une façon générale, il n'est pas possible de déclarer une référence sans l'initialiser, comme dans :

```
int & p ;     // incorrect, car pas d'initialisation
```

Notez bien qu'une fois déclarée (et initialisée), une référence ne peut plus être modifiée. D'ailleurs, aucun mécanisme n'est prévu à cet effet : si, ayant déclaré *int & p=n ;* vous écrivez *p=q*, il s'agit obligatoirement de l'affectation de la valeur de *q* à l'emplacement de référence *p*, et non de la modification de la référence *q*.

On ne peut pas initialiser une référence avec une constante. La déclaration suivante est incorrecte :

```
int & n = 3 ;        // incorrecte
```

Cela est logique puisque, si cette instruction était acceptée, elle reviendrait à initialiser *n* avec une référence à la valeur (constante) 3. Dans ces conditions, l'instruction suivante conduirait à modifier la valeur de la constante 3 :

```
n = 5 ;
```

En revanche, il est possible de définir des références constantes qui peuvent alors être initialisées par des constantes. Ainsi la déclaration suivante est-elle correcte :

```
const int & n = 3 ;
```

Elle génère une variable temporaire (ayant une durée de vie imposée par l'emplacement de la déclaration) contenant la valeur 3 et place sa référence dans *n*. On peut dire que tout se passe comme si vous aviez écrit :

```
int temp = 3 ;
int & n = temp ;
```

avec cette différence que, dans le premier cas, vous n'avez pas explicitement accès à la variable temporaire.

Enfin, les déclarations suivantes sont encore correctes :

```
float x ;
const int & n = x ;
```

Elles conduisent à la création d'une variable temporaire contenant le résultat de la conversion de *x* en *int* et placent sa référence dans *n*. Ici encore, tout se passe comme si vous aviez écrit ceci (sans toutefois pouvoir accéder à la variable temporaire *temp*) :

```
float x ; int temp = x ;
const int & n = temp ;
```

Remarque

En toute rigueur, l'appel d'une fonction conduit à une « initialisation » des arguments muets. Dans le cas d'une référence, ce sont donc les règles que nous venons de décrire qui

sont utilisées. Il en va de même pour une valeur de retour. On retrouve ainsi le comportement décrit aux paragraphes 5.2 et 13.1.

14 La spécification *inline*

Comme on peut s'y attendre, le code exécutable correspondant à une fonction est généré une seule fois par le compilateur. Néanmoins, pour chaque appel de cette fonction, le compilateur doit prévoir, non seulement le branchement au code exécutable correspondant, mais également des instructions utiles pour établir la communication entre le programme appelant et la fonction, notamment :

- sauvegarde de l'état courant (valeurs de certains registres de la machine par exemple) ;
- allocation d'espace sur la pile et copie des valeurs des arguments ;
- branchement à la fonction (dont l'adresse définitive sera en fait fournie par l'éditeur de liens) ;
- recopie de la valeur de retour ;
- restauration de l'état courant et retour dans le programme appelant.

Dans le cas de « petites fonctions », ces différentes instructions de « service » peuvent représenter un pourcentage important du temps d'exécution total de la fonction. Lorsque l'efficacité du code devient un critère important, C++ vous permet de gagner du temps dans l'appel des fonctions, au détriment de la taille du code, grâce à la spécification *inline*.

Voyez cet exemple :

```
#include <cmath>         // ancien <math.h>    pour sqrt
#include <iostream>
using namespace std ;
  /* définition d'une fonction en ligne */
inline double norme (double vec[3])
{  int i ; double s = 0 ;
   for (i=0 ; i<3 ; i++)
     s+= vec[i] * vec[i] ;
   return sqrt(s) ;
}
        /* exemple d'utilisation d'une fonction en ligne */
int main ()
{  double v1[3], v2[3] ;
   int i ;

   for (i=0 ; i<3 ; i++)
   { v1[i] = i ; v2[i] = 2*i-1 ;
   }
   cout << "norme de v1 : " << norme(v1) << "\n" ;
   cout << "norme de v2 : " << norme(v2) << "\n" ;
}
```

```
norme de v1 : 2.23607
norme de v2 : 3.31662
```

Exemple de définition et d'utilisation d'une fonction en ligne

La fonction *norme* a pour but de calculer la norme d'un vecteur à trois composantes qu'on lui fournit en argument.

La présence du mot *inline* demande au compilateur de traiter la fonction *norme* différemment d'une fonction ordinaire. À chaque appel de *norme*, le compilateur devra incorporer au sein du programme les instructions correspondantes (en langage machine[1]). Le mécanisme habituel de gestion de l'appel et du retour n'existera plus (il n'y a plus besoin de sauvegardes, recopies...), ce qui permet une économie de temps. En revanche, les instructions correspondantes seront générées à chaque appel, ce qui consommera une quantité de mémoire croissant avec le nombre d'appels.

Il est très important de noter que, par sa nature même, une fonction en ligne doit être définie dans le même fichier source que celui où on l'utilise. **Elle ne peut plus être compilée séparément !** Cette absence de possibilité de compilation séparée constitue une contrepartie notable aux avantages offerts par la fonction en ligne. En effet, pour qu'une même fonction en ligne puisse être partagée par différents programmes, il faudra absolument la placer dans un fichier en-tête[2].

Remarques

1 La déclaration *inline* constitue une demande effectuée auprès du compilateur. Ce dernier peut éventuellement (par exemple, si la fonction est volumineuse) ne pas l'introduire en ligne et en faire une fonction ordinaire. De même, si vous utilisez quelque part (au sein du fichier source concerné) l'adresse d'une fonction déclarée *inline*, le compilateur en fera une fonction ordinaire (dans le cas contraire, il serait incapable de lui attribuer une adresse et encore moins de mettre en place un éventuel appel d'une fonction située à cette adresse).

2 Notez que, si nous avions fourni la définition de *norme* après celle de *main*, il aurait été nécessaire de déclarer notre fonction, comme n'importe quelle autre, en utilisant également le mot-clé *inline* :

```
inline double norme (double [3])
```

1. Notez qu'il s'agit bien ici d'un travail effectué par le compilateur lui-même, alors que dans le cas d'une macro, un travail comparable était effectué par le préprocesseur.

2. À moins d'en écrire plusieurs fois la définition, ce qui ne serait pas « raisonnable », compte tenu des risques d'erreurs que cela comporte.

Nous avons volontairement évité ici de procéder de cette manière car, dès lors qu'une fonction en ligne figure dans un fichier en-tête, son incorporation précédera son utilisation.

Informations complémentaires

C++ a hérité de C la possibilité de définir des « macros ». Il s'agit d'instructions fournies au préprocesseur qui effectue alors des substitutions paramétrées de texte. La macro s'appelle comme une fonction (d'ailleurs, certaines des « fonctions » de la bibliothèque standard du C sont des macros) et elle présente quelques similitudes avec l'emploi de *inline* :

– le code correspondant est introduit à chaque appel (au niveau du préprocesseur, cette fois, et non plus au niveau du compilateur) ;

– on obtient un gain de temps d'exécution, en contrepartie d'une perte d'espace mémoire.

Mais la ressemblance s'arrête là, car l'emploi des macros présente de très grands risques (notamment d'effets de bord). C'est ce qui explique que les macros soient déconseillées en C++ (*inline* n'existe pas en C !). Nous étudierons les macros au paragraphe 2.2 du chapitre 31.

15 Terminaison d'un programme

Nous avons déjà vu qu'on peut arrêter le déroulement de la fonction *main* par une instruction *return*, accompagnée d'une valeur entière indiquant si l'exécution s'est déroulée convenablement (valeur 0) ou non (valeur non nulle). On peut placer autant d'instructions *return* dans le *main* qu'on le ferait dans une fonction usuelle.

Mais, si une instruction *return* dans le *main* met fin à l'exécution du programme, il n'en va plus de même pour une instruction *return* figurant dans une fonction autre que *main*. Or, dans certaines situations, en général en cas d'erreur importante, on aimera pouvoir, depuis une fonction, mettre fin à l'exécution du programme, sans devoir « remonter » la pile des différents appels. Il est alors possible de faire appel à la fonction standard *exit* (prototype dans *cstdlib*), à laquelle on fournit, ici encore, un entier utilisable comme compte rendu du déroulement du programme.

Informations complémentaires

En toute rigueur, la fonction *exit* ferme les fichiers ouverts, détruit convenablement les objets dynamiques, mais pas les objets automatiques. Néanmoins, on considère que l'appel de cette fonction correspond à une « fin normale » du programme. La valeur de l'argument précise s'il s'agit d'une fin sans aucun échec (0) ou d'une fin avec échec.

Il existe une autre fonction *abort* (prototype dans *cstdlib*), utilisée pour des fins anormales, qui peut être appelée par certaines fonctions de la bibliothèque en cas de difficultés majeures compromettant la bonne poursuite de l'exécution (nous en verrons un exemple dans le cadre de la gestion des exceptions). Celle-ci met fin brutalement au programme, sans aucune intervention au niveau des fichiers ou des objets.

Enfin, il est possible de demander qu'une ou plusieurs fonctions de son choix soient exécutées lors de la fin d'un programme, en utilisant la fonction *atexit*[1].

1. On trouvera plus d'informations sur ces fonctions, héritées du langage C, dans *Langage C*, du même auteur, chez le même éditeur.

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++0x 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
{  int x ;
   int y ;
   void initialise (...) ;
    .....
} ;
```

```
class point
{ public :
    int x ;
    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++0x offrira 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++0x permettra 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 :

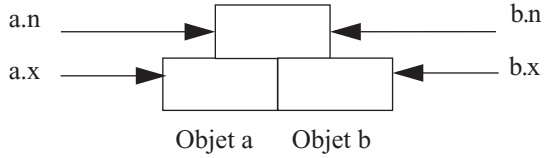
Une façon (parmi d'autres) de permettre à plusieurs objets de partager des données consiste à déclarer avec le qualificatif *static* les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe. Par exemple, si nous définissons une classe *exple2* par :

```
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++0x étendra 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 *static* 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++0x) 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.