Programmation Orientée Objet (POO)

UNIVERSITE GASTON BERGER DE SAINT-LOUIS

U.F.R. S.A.T.

DIETEL

(Diplôme d'Ingénieur en Electronique et Télécommunication)

GETI

(Génie en Electronique, Télécommunication et Informatique)

INTRODUCTION :

La programmation orientée objet (en abrégé P.O.O.) est dorénavant universellement reconnue pour les avantages qu'elle procure. Notamment, elle améliore largement la productivité des développeurs, la robustesse, la portabilité et l'extensibilité de leurs programmes. Enfin, et surtout, elle permet de développer des composants logiciels entièrement réutilisables. Un certain nombre de langages dits "langages orientés objet" (L.O.O.) ont été définis de toutes pièces pour appliquer les concepts de P.O.O. C'est ainsi que sont apparus dans un premier temps des langages comme Smalltalk, Simula ou Eiffel puis, plus récemment, Java. Le langage C++, quant à lui, a été conçu suivant une démarche quelque peu différente par B. Stroustrup (AT&T) ; son objectif a été, en effet, d'adjoindre au langage C un certain nombre de spécificités lui permettant d'appliquer les concepts de la P.O.O. Ainsi, C++ présente-t-il sur un vrai L.O.O. l'originalité d'être fondé sur un langage répandu. Ceci laisse au programmeur toute liberté d'adopter un style plus ou moins orienté objet, en se situant entre les deux extrêmes que constituent la poursuite d'une programmation classique d'une part, une pure P.O.O. d'autre part. Si une telle liberté présente le risque de céder, dans un premier temps, à la facilité en mélangeant les genres (la P.O.O. ne renie pas la programmation classique – elle l'enrichit), elle

permet également une transition en douceur vers la P.O.O pure, avec tout le bénéfice qu'on peut en escompter à terme.

GÉNÉRALITÉS CONCERNANT C++ :

Problématique de la programmation :

L'activité de programmation a toujours suscité des réactions diverses. Pour certains en effet, il ne s'agit que d'un jeu de construction enfantin, dans lequel il suffit d'enchaîner des instructions élémentaires (en nombre restreint) pour parvenir à résoudre n'importe quel problème ou presque. Pour d'autres au contraire, il s'agit de produire (au sens industriel du terme) des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères, notamment :

• l'exactitude : aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux spécifications) ;

• la robustesse : aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation ;

• l'extensibilité : facilitée avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications ;

• la réutilisabilité : possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;

• la portabilité : facilitée avec laquelle on peut exploiter un même logiciel dans différentes implémentations ;

• l'efficience : temps d'exécution, taille mémoire...

La programmation structurée :

La programmation structurée a manifestement fait progresser la qualité de la production des logiciels. Mais, avec le recul, il faut bien reconnaître que ses propres fondements lui imposaient des limitations "naturelles". En effet, la programmation structurée reposait sur ce que l'on nomme souvent "l'équation de Wirth", à savoir :

Algorithmes + Structures de données = Programmes

Or, en pratique, on s'est aperçu que l'adaptation ou la réutilisation d'un logiciel conduisait souvent à "casser" le module intéressant, et ceci parce qu'il était nécessaire de remettre en cause une structure de données. Précisément, ce type de difficultés émane directement de l'équation de Wirth, qui découple totalement les données des procédures agissant sur ces données.

Les apports de la Programmation Orientée Objet :

Objet :

C'est là qu'intervient la Programmation Orientée Objet (en abrégé P.O.O), fondée justement sur le concept d'objet, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données. Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la P.O.O. est :

Méthodes + Données = Objet

Exemple :

Considérons l'objet *voiture*, des méthodes possibles peuvent être : *accélérer, freiner, droiter* en gros l'ensemble des opérations susceptible d'être utilisées. Des données possibles peuvent être : *couleur, taille, marque,* forme en gros l'ensemble des informations caractérisant une voiture. L'équation de wirth est donc :

Méthodes (*accélérer, freiner, droiter*) + données (*couleur, taille, marque, forme*) = voiture (voiture)

Encapsulation :

Mais cette association est plus qu'une simple juxtaposition. En effet, dans ce que l'on pourrait qualifier de P.O.O. "pure", on réalise ce que l'on nomme une encapsulation des données. Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un "message" à l'objet.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel. Elle facilite considérablement la maintenance : une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était bien sûr pas le cas avec la programmation structurée). De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

Exemple :

dans notre exemple avec l'objet voiture, on dit qu'on a réalisé une encapsulation des données dès lors que l'équation de wirth est établie : l'objet forme un ensemble bien défini.

Classe

En P.O.O. apparaît généralement le concept de classe, qui correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une "instance" de sa classe). Une classe représente l'abstraction de la structure et n'est pas utilisable directement (on peut utiliser une voiture avec ses caractéristiques et non pas l'ensemble des voitures) mais pour être manipulé, une classe fait appel à ses variables (objet ou instance).

Exemple :

dans notre exemple avec voiture, on a en fait défini une *classe voiture* et une voiture donnée serait un objet de la classe.

Héritage :

Un autre concept important en P.O.O. est celui d'héritage. Il permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise en bloc !), à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La conception de la nouvelle classe, dite qui "hérite" des propriétés et des aptitudes de l'ancienne, peut ainsi s'appuyer sur des réalisations antérieures parfaitement au point et les "spécialiser" à volonté. Comme on peut s'en douter, l'héritage facilite largement la réutilisation de produits existants, d'autant plus qu'il peut être réitéré autant de fois que nécessaire (la classe C peut hériter de B, qui elle-même hérite de A).

Polymorphisme :

Généralement, en P.O.O, une classe dérivée peut "redéfinir" (c'est-à-dire modifier) certaines des méthodes héritées de sa classe de base. Cette possibilité est la clé de ce que l'on nomme polymorphisme, c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient tous de classes dérivées de la même classe de base. Plus précisément, on utilise chaque objet comme s'il était de cette classe de base, mais son comportement effectif dépend de sa classe effective (dérivée de cette classe de base), en pariculier de la manière dont ses propres méthodes ont été redéfinies. Le polymorphisme améliore l'extensibilité des programmes, en permettant d'ajouter de nouveaux objets dans un scénario préétabli et, éventuellement, écrit avant d'avoir connaissance du type effectif de ces objets.

Les spécificités de C++ :

C++ présente, par rapport au C ANSI, des extensions qui ne sont pas véritablement orientées P.O.O. En voici un bref résumé :

• nouvelle forme de commentaire (en fin de ligne),

• plus grande liberté dans l'emplacement des déclarations,

• notion de référence facilitant la mise en œuvre de la transmission d'arguments par adresse,

• surdéfinition des fonctions : attribution d'un même nom à différentes fonctions, la reconnaissance de la fonction réellement appelée se faisant d'après le type et le nombre des arguments figurant dans l'appel (on parle parfois de signature),

• nouveaux opérateurs de gestion dynamique de la mémoire : new et delete,

• possibilité de définir des fonctions "en ligne" (inline), ce qui accroît la vitesse d'exécution, sans perdre pour autant le formalisme des fonctions.

Les entrées-sorties conversationnelles du C++ :

C++ dispose de toutes les routines offertes par la bibliothèque standard du C ANSI. Mais il comporte également de nouvelles possibilités d'entrées-sorties. Celles-ci reposent sur les notions de flots et de surdéfinition d'opérateur que nous n'aborderons qu'ultérieurement.

Affichage à l'écran :

Là où en C, on écrivait :

```
printf ("bonjour") ;
```

en C++, on utilisera :

```
cout << "bonjour" ;
```

• cout désigne un "flot de sortie" prédéfini, associé à l'entrée standard du C (stdout);

• << est un "opérateur" dont l'opérande de gauche (ici cout) est un flot et l'opérande de droite une expression de type quelconque. L'instruction précédente peut être interprétée comme ceci : le flot cout reçoit la valeur "bonjour".

Considérons ces instructions :

```
int n = 25 ;
cout << "valeur : " ;
cout << n ;
```

Elles affichent le résultat suivant :

valeur : 25

peuvent se condenser en une seule

cout << "valeur :" << n ;

En C, l'utilisation de fonctions telles que printf ou scanf nécessitait l'incorporation du fichier en-tête <stdio.h> qui contenait les déclaration appropriées. De façon comparable, les déclarations nécessaires à l'utilisation des entrées-sorties spécifiques à C++ figurent dans un fichier en-tête de nom <iostream>. Cependant, l'utilisation des symboles déclarés dans ce fichier iostream fait appel à la notion d'espace de noms, introduite elle-aussi par la norme. Cette notion sera présentée ultérieurement. Pour l'instant, sachez qu'elle oblige à introduire dans votre programme une instruction de déclaration using, dont la portée se définit comme celle de toute déclaration, et qui se présente ainsi :

using namespace std ; /* on utilisera les symboles définis dans */

/* l'espace de noms standard s'appelant std */

Exemple :

```
#include <iostream>

using namespace std ;

main()

{

  int n = 25 ; long p = 250000; unsigned q = 63000 ;

  char c = 'a' ;

  float x = 12.3456789 ; double y = 12.3456789e16 ;

  char * ch = "bonjour" ;

  int * ad = & n ;

  cout << "valeur de n  : "  << n  << "\n" ;

  cout << "valeur de p  : "  << p  << "\n" ;
```

```
cout << "caractere c  : "  << c  << "\n" ;

cout << "valeur de q  : "  << q  << "\n" ;

cout << "valeur de x  : "  << x  << "\n" ;

cout << "valeur de y  : "  << y  << "\n" ;

cout << "chaine ch    : "  << ch << "\n" ;

cout << "adresse de n : " << ad << "\n" ;

cout << "adresse de ch : " << (void *) ch << "\n" ;
```

Résultat :

```
valeur de n  : 25

valeur de p  : 250000

caractere c  : a

valeur de q  : 63000

valeur de x  : 12.3457

valeur de y  : 1.23457e+017

chaine ch    : bonjour

adresse de n : 006AFDF4

adresse de ch : 0046D0D4
```

Lecture au clavier :

De même qu'il existe un flot de sortie prédéfini cout associé à la sortie standard du C (stdout), il existe un flot d'entrée prédéfini, nommé cin, associé à l'entrée standard du C (stdin). De même que l'opérateur << permet d'envoyer des informations sur un flot de sortie (donc, en particulier, sur cout), l'opérateur >> permet de recevoir de l'information en provenance d'un flot d'entrée (donc, en par Par exemple, l'instruction (n étant de type int):

```
cin >> n ;
```

demandera de lire des caractères sur le flot cin et de les convertir en une valeur de type int.

On peut aussi faire :

```
cin>>n>>p>>…>> ;
```

Les spécificités du C++ :

Le langage C++ dispose d'un certain nombre de spécificités axées sur la P.O.O. Il s'agit essentiellement des suivantes :

• nouvelle forme de commentaire (en fin de ligne),

• emplacement libre des déclarations,

• notion de référence,

• arguments par défaut dans les déclarations des fonctions,

• surdéfinition de fonctions,

•opérateurs new et delete,

• fonctions "en ligne" (inline),

• nouveaux opérateurs de cast,

• existence d'un type booléen bool,

• notion d'espace de noms.

Le commentaire de fin de ligne :

Exemple :

```
cout << "bonjour\n" ; // formule de politesse
```

Déclarations et initialisations :

C++ s'avère plus souple que le C ANSI en matière de déclarations. Plus précisément, en C++, il n'est plus obligatoire de regrouper au début les déclarations effectuées au sein d'une fonction ou au sein d'un bloc. Celles-ci peuvent être effectuées où bon vous semble, pour peu qu'elles apparaissent avant que l'on en ait besoin : leur portée reste limitée à la partie du bloc ou de la fonction suivant leur déclaration.

Voici un exemple incorrect en C ANSI et accepté en C++ :

```
main()
```

```
{
  int n ;
    .....
  n = ...
    .....
  int q = 2*n - 1 ;
    .....
}
```

L'instruction suivante est acceptée en C++ :

```
for (int i=0 ; ... ; ...)
{
.....
}
```

La notion de référence :

En C, les arguments et la valeur de retour d'une fonction sont transmis par valeur. Pour simuler en quelque sorte ce qui se nomme "transmission par adresse" dans d'autres langages, il est alors nécessaire de "jongler" avec les pointeurs (la transmission se fait toujours par valeur mais, dans ce cas, il s'agit de la valeur d'un pointeur). En C++, le principal intérêt de la notion de référence est qu'elle permet de laisser le compilateur mettre en œuvre les "bonnes instructions" pour assurer un transfert par adresse. Pour mieux vous en faire saisir l'intérêt, nous vous proposons de vous rappeler comment il fallait procéder en C

Exemple :

Considérons l'exemple suivant, qui illustre le fait qu'en C les arguments mis par valeur :

```
#include <iostream>
using namespace std ;
main()
```

```
{
void echange (int, int) ;
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 << "debut echange : " << a << " " << b << "\n" ;
c = a ; a = b ; b = c ;
cout << "fin echange   : " << a << " " << b << "\n" ;
}
```

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

Lors de l'appel d'echange, il y a transmission des valeurs de n et de p ; on peut considérer que la fonction les a recopiées dans des emplacements locaux, correspondant à ses arguments formels a et b, et qu'elle a effectivement "travaillé" sur ces copies.

Bien entendu, il est toujours possible de programmer une fonction echange pour qu'elle opère sur des variables de la fonction qui l'appelle ; il suffit tout simplement de lui fournir en argument l'adresse de ces variables, comme dans l'exemple suivant :

```
#include <iostream>
```

```
using namespace std ;

main()

{  void echange (int *, int *) ;

   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 << "debut echange : " << * a << " " << * b << "\n" ;

   c = *a ; *a = *b ; *b = c ;

   cout << "fin echange   : " << * a << " " << * b << "\n" ;

}
```

```
avant appel :  10 20

debut echange : 10 20

fin echange   : 20 10

apres appel :  20 10
```

Exemple de transmission d'argument par référence :

C++ permet de demander au compilateur de prendre lui-même en charge la transmission des arguments par adresse : on parle alors de transmission d'argument par référence. Le programme ci-dessous montre comment appliquer un tel mécanisme à notre fonction echange :

```
#include <iostream>

using namespace std ;

main()
```

```
{  void echange (int &, int &) ;
   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

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 (et non la valeur) : il n'est plus utile (et ce serait une erreur !) de faire appel à l'opérateur d'indirection *.

Autrement dit, il suffit d'avoir fait ce choix de transmission par référence au niveau de l'en-tête de la fonction pour que le processus soit entièrement pris en charge par le compilateur .

Le même phénomène s'applique au niveau de l'utilisation de la fonction. Il suffit en effet d'avoir spécifié, dans le prototype, les arguments (ici, les deux) que l'on souhaite voir transmis par référence. Au niveau de l'appel :

```
echange (n, p) ;
```

nous n'avons plus à nous préoccuper du mode de transmission utilisé.

Remarques :

```
void f (int & n) ;   // f reçoit la référence à un entier
float x ;
  .....
f(x) ;       // appel illégal car f est defini avec une référence sur int, il ny a jamais de conversion de type
```

Supposons qu'une fonction fct ait pour prototype :

```
void fct (int &) ;
```

Le compilateur refusera alors un appel de la forme suivante (n étant de type int):

```
fct (3) ;     // incorrect : f ne peut pas modifier une constante
```

Il en ira de même pour :

```
const int c = 15 ;
  .....
fct (c) ;     // incorrect : f ne peut pas modifier une constante
```

Lorsqu'on doit transmettre en argument la référence à un pointeur, on est amené à utiliser ce genre d'écriture :

```
int * & adr       // adr est une référence à un pointeur sur un int
```

En revanche, considérons une fonction de prototype :

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

La déclaration const int & correspond à une référence à une constante . Les appels suivants seront corrects :

```
const int c = 15 ;

.....

fct1 (3) ;    // correct ici

fct1 (c) ;    // correct ici
```

En effet, une référence n'est pas un pointeur, même si, au bout du compte, l'information transmise à la fonction est une adresse. En particulier, l'usage qui est fait dans la traduction des instructions de la fonction n'est pas le même : dans le cas d'un pointeur, on utilise directement sa valeur, quitte à mentionner une indirection avec l'opérateur * ; avec une référence, l'indirection est ajoutée automatiquement par le compilateur.

La référence d'une manière générale :

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.

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 ;

Les arguments par défaut :

Considérez l'exemple suivant :

```
#include <iostream>

using namespace std ;
```

```
main()
{
  int n=10, p=20 ;
  void fct (int, int=12) ; // proto avec une valeur par défaut
  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
```

Surdéfinition de fonctions :

Exemple de surdéfinition de la fonction sosie :

```
#include <iostream>
using namespace std ;
void sosie (int) ;          // les prototypes
void sosie (double) ;
```

```
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
```

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).

Les opérateurs new et delete :

En langage C, la gestion dynamique de mémoire fait appel à des fonctions de la bibliothèque standard telles que malloc et free. Bien entendu, comme toutes les fonctions standard, celles-ci restent utilisables en C++. Mais dans le contexte de la Programmation Orientée Objet, C++ a introduit deux nouveaux opérateurs, new et delete, particulièrement adaptés à la gestion dynamique d'objets.

Avec la déclaration :

int * ad ;

l'instruction :

ad = new int ;

permet d'allouer l'espace mémoire nécessaire pour un élément de type int et d'affecter à ad l'adresse correspondante. En C, vous auriez obtenu le même résultat en écrivant (l'opérateur

de cast, ici int *, étant facultatif). :

ad = (int *) malloc (sizeof (int)) ;

Comme les déclarations ont un emplacement libre en C++, vous pouvez même déclarer la variable ad au moment où vous en avez besoin en écrivant, par exemple :

int * ad = new int ;

Avec la déclaration :

char * adc ;

l'instruction :

adc = new char[100] ;

alloue l'emplacement nécessaire pour un tableau de 100 caractères et place l'adresse (de début) dans adc. En C, vous auriez obtenu le même résultat en écrivant :

adc = (char *) malloc (100) ;

L'opérateur unaire (à un seul opérande) new s'utilise ainsi :

new type

où type représente un type absolument quelconque. Il fournit comme résultat un pointeur (de type type*) sur l'emplacement correspondant, lorsque l'allocation a réussi. L'opérateur new accepte également une syntaxe de la forme :

new type [n]

où n désigne une expression entière quelconque (non négative). Cette instruction alloue alors l'emplacement nécessaire pour n éléments du type indiqué ; si l'opération a réussi, elle fournit en résultat un pointeur (toujours de type type *) sur le premier élément de ce tableau.

Lorsque l'on souhaite libérer un emplacement alloué préalablement par new, on doit absolument utiliser l'opérateur delete. Ainsi, pour libérer les emplacements créés, on écrit :

```
delete ad ;
```

pour l'emplacement alloué par :

```
ad = new int ;
```

et :

```
delete adc ;
```

pour l'emplacement alloué par :

```
adc = new char [100] ;
```

La syntaxe usuelle de l'opérateur delete est la suivante (adresse étant une expression devan avoir comme valeur un pointeur sur un emplacement alloué par new):

```
delete adresse ;
```

L'opérateur new (nothrow) :

new déclenche une exception bad_alloc en cas d'échec. Dans les versions d'avant la norme, new fournissait (comme malloc) un pointeur nul en cas d'échec. Avec la norme, on peut retrouver ce comportement en utilisant, au lieu de new, l'opérateur new(std::nothrow) (std:: est superflu dès lors qu'on a bien déclaré cet espace de nom par using).

Exemple :

```
#include <cstdlib>     //  ancien <stdlib.h>   pour exit
#include <iostream>
using namespace std ;
main()
{  long taille ;
  int * adr ;
  int nbloc ;
  cout << "Taille souhaitee ? " ;
```

```
  cin >> taille ;
  for (nbloc=1 ; ; nbloc++)
  { adr = new (nothrow) int [taille] ;
   if (adr==0) { cout << "**** manque de memoire ****\n" ;
                 exit (-1) ;
                }
   cout << "Allocation bloc numero : " << nbloc << "\n" ;
  }
}
```

```
Taille souhaitee ? 4000000
Allocation bloc numero : 1
Allocation bloc numero : 2
Allocation bloc numero : 3
**** manque de memoire ****
```

Gestion des débordements de mémoire avec set_new_handler :

Comme on l'a vu, new déclenche une exception bad_alloc en cas d'échec. Mais il est également possible de définir une fonction de votre choix et de demander qu'elle soit appelée en cas de manque de mémoire. Il vous suffit pour cela d'appeler la fonction set_new_handler en lui fournissant, en argument, l'adresse de la fonction que vous avez prévue pour traiter le cas de manque de mémoire. Voici comment nous pourrions adapter

l'exemple précédent :

```
#include <cstdlib>    // ancien <stdlib.h>   pour exit
#include <new>        // pour set_new_handler  (parfois <new.h>)
#include <iostream>
using namespace std ;
```

```
main()
{
  void deborde () ;   // proto fonction appelée en cas manque mémoire
  set_new_handler (deborde) ;
  long taille ;
  int * adr ;
  int nbloc ;
  cout << "Taille de bloc souhaitee (en entiers) ? " ;
  cin >> taille ;
  for (nbloc=1 ; ; nbloc++)
     { adr = new int [taille] ;
       cout << "Allocation bloc numero : " << nbloc << "\n" ;
     }
}
void deborde ()       // fonction appelée en cas de manque mémoir
{  cout << "Memoire insuffisante\n" ;
  cout << "Abandon de l'execution\n" ;
  exit (-1) ;
}
```

```
Taille de bloc souhaitee (en entiers) ? 4000000
Allocation bloc numero : 1
Allocation bloc numero : 2
Allocation bloc numero : 3
Memoire insuffisante pour allouer 16000000 octets
```

```
Abandon de l'execution

Press any key to continue
```

La spécification inline :

En langage C, vous savez qu'il existe deux notions assez voisines, à savoir les macros et les fonctions. Une fonction en ligne se définit et s'utilise comme une fonction ordinaire, à la seule différence qu'on fait précéder son en-tête de la spécification inline. En voici un 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 */
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
```

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. A chaque appel de norme, le compilateur devra incorporer au sein du programme les instructions correspondantes (en langage machine). 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 !

	Avantages	Inconvénients
Macro	- Économie de temps d'exécution	- Perte d'espace mémoire - Risque d'effets de bord non désirés - Pas de compilation séparée possible
Fonction	- Économie d'espace mémoire - Compilation séparée possible	- Perte de temps d'exécution
Fonction "en ligne"	- Économie de temps d'exécution	- Perte d'espace mémoire - Pas de compilation séparée possible

Comparaison entre macro, fonction et fonction en ligne

Les espaces de noms :

Lorsque l'on doit utiliser plusieurs bibliothèques dans un programme, on peut être confronté au problème dit de "pollution de l'espace des noms", lié à ce qu'un même identificateur peut très bien avoir été utilisé par plusieurs bibliothèques. Le même problème peut se poser, à un degré moindre toutefois, lors du développement de gros programmes. C'est la raison pour laquelle la norme ANSI du C++ a introduit le concept d'"espace de noms". Il s'agit simplement de donner un nom à un "espace" de déclarations, en procédant ainsi :

```
namespace une_bibli
```

```
{  // déclarations usuelles

}
```

Pour se référer à des identificateurs définis dans cet espace de noms, on utilisera une instruction using :

```
using namespace une_bibli

// ici, les identificateurs de une_bibli sont connus
```

On peut lever l'ambiguïté risquant d'apparaître lorsqu'on utilise plusieurs espaces de noms comportant des identificateurs identiques ; il suffit pour cela de faire appel à l'opérateur de **résolution de portée, par exemple :**

```
une_bibli::point ...     // on se réfère à l'identificateur point

                         //   de l'espace de noms une_bibli
```

On peut aussi utiliser l'instruction using pour faire un choix permanent :

```
using une_bibli::point ;    // dorénavant, l'identificateur point, employé seul

                            //   correspondra à celui défini dans

                            //   l'espace de noms une_bibli
```

Tous les identificateurs des fichiers en-tête standard sont définis dans l'espace de noms std ; aussi est-il nécessaire de recourir systématiquement à l'instruction :

```
using namespace std ;       /* utilisation des fichiers en-tête standard */
```

Généralement, cette instruction figurera à un niveau global, comme nous l'avons fait dans les exemples précédents. En revanche, elle ne peut apparaître que si l'espace de noms qu'elle mentionne existe déjà ; en pratique, cela signifie que cette instruction sera placée après l'inclusion des fichiers en-tête.

Le type bool :

Ce type est tout naturellement formé de deux valeurs notées true et false. En théorie, les résultats des comparaisons ou des combinaisons logiques doivent être de ce type. Toutefois, il existe des **conversions implicites :**

•de bool en numérique, true devenant 1 et false devenant 0 ;

• de numérique (y compris flottant) en bool, toute valeur non nulle devenant true et zéro devenant false.

Dans ces conditions, tout se passe finalement comme si bool était un type énuméré ainsi défini :

```
typedef enum { false=0, true } bool ;
```

En définitive, ce type bool sert surtout à apporter plus de clarté aux programmes, sans remettre en cause quoi que ce soit.

Exercices :

Résolution de l'équation du second degré : $ax^2 + bx + c = 0$

Correction :

```
#include <cmath>
#include <iostream>
using namespace std;
float a,b,c,disc;
main()
{
cout<<" RESOLUTION SECOND DEGRE ";
cout<<" donner le coefficient  a="<<endl;
cin>>a;
cout<<" donner le coefficient  b="<<endl;
cin>>b;
cout<<" donner le coefficient  c="<<endl;
cin>>c;
disc=pow(b,2)-4*a*c;
        if(a==0 && b==0 && c==0)
         cout<<" tout reel est solution "<<endl;
```

```
if(a==0)
  { if (b==0 && c!=0)
      cout<<" solution impossible "<<endl;
   if (b!=0)
      cout<<" la solution est  : "<<-c/b<<endl;
   }
  else {
      if(disc==0)
       cout<<" on a une solution double  xo = "<<-b/2*a<<endl;
      if(disc>0)
       { cout<<" on a deux solutions reelles :"<<endl<<" x1 = "<<(-b-
sqrt(disc))/(2*a)<<endl;
        cout<<"x2 = "<<(-b+sqrt(disc))/(2*a)<<endl;
       }
      if(disc<0)
        { cout<<" on a deux solutions immaginaires :"<<endl<<" z1 = "<<-b/2*a<<" – J
"<<sqrt(-disc)/(2*a)<<endl;
cout<<"z2 = "<<-b/2*a<<" +J "<<sqrt(-disc)/(2*a)<<endl;
        }
   }

system("pause");
}
```

Classe et Objet :

Les structures en C++ :

En C, une déclaration telle que :

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

définit un type structure nommé point (on dit aussi un modèle de structure nommé point ou parfois, par abus de langage, la structure point). Quant à x et y, on dit que ce sont des champs ou des membres de la structure point.

On déclare ensuite des variables du type point par des instructions telles que :

```
struct point a, b ;
```

Celle-ci réserve l'emplacement pour deux structures nommées a et b, de type point. L'accès aux membres (champs) de a ou de b se fait à l'aide de l'opérateur point (.) ; par exemple, a.y désigne le membre de la structure.

Supposons que nous souhaitions associer à la structure point précédente 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.

Voici comment nous pourrions déclarer notre structure point :

```
/* ------------ Déclaration du type 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 () ;
} ;
```

Définition des fonctions membres :

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

Manipulation :

```
a.initialise (5,2) ;
a.deplace(1,-4) ;
```

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.

Exercice :

Faire un programme reprenant la déclaration du type point (dans l'espace), la définition de ses fonctions membres et faites un exemple d'utilisation dans la fonction main.

Correction :

```
#include <iostream>
#include <cmath>
using namespace std;
struct point
    {
    float x,y,z;
   void initialise(float a,float b,float c);
     void deplace(float,float,float);
   void affiche();
   float norme(float,float,float);
    };
 void point::initialise(float a,float b,float c)
       {
        x=a;
         y=b;
         z=c;
```

```
        }
 void point::deplace(float a, float b, float c)
        {
        x+=a; y+=b; z+=c;
        }
 void point::affiche()
   {
   cout<<" point de coordonnees "<<"( "<<x<<" , "<<y<<" , "<<z<<" ) "<<endl;
   cout<<" la norme euclidienne est : "<<norme(x,y,z)<<endl;
   }
 float point::norme(float a,float b, float c)
    {
    return sqrt(pow(a,2)+pow(b,2)+pow(c,2));
    }
main()
{
point pt1;
float abs,ord,haut;
cout<<" donner un abscisse "<<endl;
cin>>abs;
cout<<" donner un ordonnee "<<endl;
cin>>ord;
cout<<" donner une hauteur "<<endl;
cin>>haut;
```

```
pt1.initialise(abs,ord,haut);
 cout<<" donner les coordonnees de deplacement "<<endl<<endl;
 cout<<" donner un abscisse de deplacement "<<endl;
 cin>>abs;
 cout<<" donner un ordonnee de deplacement"<<endl;
 cin>>ord;
 cout<<" donner une hauteur de deplacement "<<endl;
 cin>>haut;
cout<<" avant deplacement "<<endl<<endl;;
pt1.affiche();
pt1.deplace(abs,ord,haut);
cout<<"apres deplacement"<<endl<<endl;;
pt1.affiche();
 system("pause");
 }
```

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 :

```
/* ------------ Déclaration de la classe point ------------- */
```

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

- Dans le jargon de la P.O.O., une variable de la classe point est une instances de la classe point, ou encore que ce sont des objets de type point ; c'est généralement ce dernier terme que nous utiliserons.
- 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.

- 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 .
- 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 :
    ...
} ;
```

- Par défaut les class se font avec le status private par contre les struct se font avec public.
- 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.

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.

Notions de constructeur et de destructeur :

A priori, les objets suivent les règles habituelles concernant leur initialisation par défaut. 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 : allocation dynamique de mémoire , 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.

Un objet pourra aussi posséder un **destructeur**, c'est-à-dire une fonction membre appelé 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 (~).

```
#include <iostream>

#include <cmath>
```

```
using namespace std;
class point
      {
      float x,y,z;
    public :
       point()
         {
             cout<<" appel du constructeur par defaut "<<endl;
         x=y=z=0;
         }
       point(float a,float b,float c);
       ~point()
           { cout<<" appel du destructeur "<<endl; }
       void deplace(float,float,float);
        void affiche();
       float norme(float,float,float);
      };
 point::point(float a,float b,float c)
         {
          cout<<" appel du second constructeur "<<endl;
           x=a;
            y=b;
            z=c;
            }
```

```
void point::deplace(float a, float b, float c)
        {
        x+=a; y+=b; z+=c;
        }
void point::affiche()
   {
   cout<<" point de coordonnees "<<"( "<<x<<" , "<<y<<" , "<<z<<" ) "<<endl;
   cout<<" la norme euclidienne est : "<<norme(x,y,z)<<endl;
   }
float point::norme(float a,float b, float c)
    {
    return sqrt(pow(a,2)+pow(b,2)+pow(c,2));
    }
    void fonction(void)  // fonction sans grand interet qui consite à visualiser le destructeur
       {
       point unpts;
       }


main()
{
point pt1;
float abs,ord,haut;
cout<<" donner un abscisse "<<endl;
cin>>abs;
```

```
 cout<<" donner un ordonnee "<<endl;

 cin>>ord;

 cout<<" donner une hauteur "<<endl;

 cin>>haut;

point pt2(abs,ord,haut);

 cout<<" donner les coordonnees de deplacement "<<endl<<endl;

 cout<<" donner un abscisse de deplacement "<<endl;

 cin>>abs;

 cout<<" donner un ordonnee de deplacement"<<endl;

 cin>>ord;

 cout<<" donner une hauteur de deplacement "<<endl;

 cin>>haut;

cout<<" avant deplacement "<<endl<<endl;;

pt1.affiche();

pt1.deplace(abs,ord,haut);

cout<<"apres deplacement"<<endl<<endl;;

pt1.affiche();

fonction();  // pour visuliser le rôle du constructure

 system("pause");

 }
```

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

point a ; // correct si ya un constructeur sans argument, déclaration classique
```

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.

Exemple :

Annalysons le code suivant :

```
#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" ;
}
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
```

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.

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). 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.

Les membres données statiques :

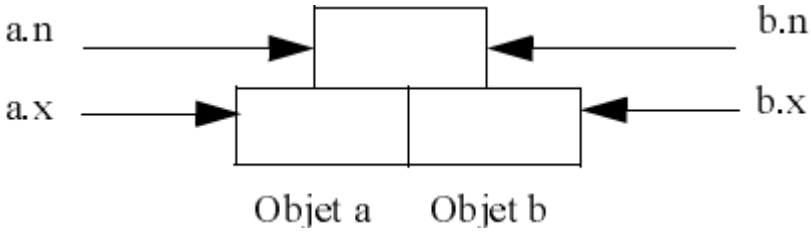
Une façon (parmi d'autres) de permettre à plusieurs objets de partager des données consiste à déclarer avec le qualificatif static les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe. Par exemple, si nous définissons une classe :

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

la déclaration :

```
exple2 a, b ;
```

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



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
   .....
} ;
```

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 ;
#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" ;
```

```
}
cpte_obj::~cpte_obj ()          // destructeur
{   cout << "-- destruction  : il reste maintenant " << --ctr << " objets\n" ;
}
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
```

La classe comme composant logiciel :

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 () ;
} ;
```

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" ;

}
```

Exercices :

modulariser la classe point en deux fichier : declaration.h et classe_point.cpp.

correction :

declaration.h

```
#include <cmath>

#include <iostream>

#include <cstdlib>

using namespace std;

class point

      {

     float x,y,z;

    public :

       point()

         {

             cout<<" appel du constructeur par defaut "<<endl;

        x=y=z=0;

         }

      point(float a,float b,float c);

      ~point()

          { cout<<" appel du destructeur "<<endl; }

      void deplace(float,float,float);

       void affiche();
```

```
    float norme(float,float,float);
  };
point::point(float a,float b,float c)
    {
    cout<<" appel du second constructeur "<<endl;
    x=a;
     y=b;
     z=c;
     }
void point::deplace(float a, float b, float c)
    {
    x+=a; y+=b; z+=c;
    }
void point::affiche()
  {
  cout<<" point de coordonnees "<<"( "<<x<<" , "<<y<<" , "<<z<<" ) "<<endl;
  cout<<" la norme euclidienne est : "<<norme(x,y,z)<<endl;
  }
float point::norme(float a,float b, float c)
   {
   return sqrt(pow(a,2)+pow(b,2)+pow(c,2));
   }
   void fonction(void)
     {
```

```
        point unpts;
        }
```

Class_point.cpp

```
#include <cstdlib>
#include <iostream>
#include "declaration.h"
using namespace std;


main()
{
   point pt1;
float abs,ord,haut;
cout<<" donner un abscisse "<<endl;
cin>>abs;
 cout<<" donner un ordonnee "<<endl;
 cin>>ord;
 cout<<" donner une hauteur "<<endl;
 cin>>haut;
point pt2(abs,ord,haut);
 cout<<" donner les coordonnees de deplacement "<<endl<<endl;
 cout<<" donner un abscisse de deplacement "<<endl;
 cin>>abs;
 cout<<" donner un ordonnee de deplacement"<<endl;
 cin>>ord;
```

```
 cout<<" donner une hauteur de deplacement "<<endl;
 cin>>haut;
cout<<" avant deplacement "<<endl<<endl;;
pt2.affiche();
pt1=pt2;
pt1.deplace(abs,ord,haut);
cout<<"apres deplacement"<<endl<<endl;;
pt1.affiche();
fonction();
   system("PAUSE");
}
```

Les propriétés des fonctions membres :

Il est possible de transmettre explicitement en argument l'adresse d'un objet. Rappelons que, dans un tel cas, on ne change pas le mode de transmission de l'argument (contrairement à ce qui se produit avec la transmission par référence) ; on se contente de transmettre une valeur qui se trouve être une adresse et qu'il faut donc interpréter en conséquence dans la fonction (notamment en employant l'opérateur d'indirection *). A titre d'exemple :

```
#include <iostream>
using namespace std ;
class point                    // Une classe point contenant seulement :
{  int x, y ;
 public :
  point (int abs=0, int ord=0)       // un constructeur ("en ligne")
      { x=abs; y=ord ; }
  int coincide (point *) ;          // une fonction membre : coincide
```

```
} ;

int point::coincide (point * adpt)

{  if ( (adpt->x == x) && (adpt->y == y) ) return 1 ;

                              else return 0 ;

}

main()                          // Un petit programme d'essai

{  point a, b(1), c(1,0) ;

  cout << "a et b : " << a.coincide(&b) << " ou " << b.coincide(&a) << "\n" ;

  cout << "b et c : " << b.coincide(&c) << " ou " << c.coincide(&b) << "\n" ;

}
```

```
a et b : 0 ou 0

b et c : 1 ou 1
```

Remarque :

N'oubliez pas qu'à partir du moment où vous fournissez l'adresse d'un objet à une fonction membre, celle-ci peut en modifier les valeurs (elle a accès à tous les membres s'il s'agit d'un objet de type de sa classe, aux seuls membres publics dans le cas contraire). Si vous craignez de tels effets de bord au sein de la fonction membre concernée, vous pouvez toujours employer le qualificatif const. Ainsi, ici, l'en-tête de *coincide* aurait pu être :

```
int point::coincide (const point * adpt)
```

en modifiant parallèlement son prototype :

```
int coincide (const point *) ;
```

Transmission par référence :

Comme nous l'avons vu, l'emploi des références permet de mettre en place une transmission par adresse, sans avoir à en prendre en charge soi-même la gestion. Elle simplifie d'autant l'écriture de la

fonction concernée et ses différents appels. Voici une adaptation de *coincide* dans laquelle son argument est transmis par référence :

```
#include <iostream>

using namespace std ;

class point                    // Une classe point contenant seulement :

{  int x, y ;

 public :

   point (int abs=0, int ord=0)       // un constructeur ("en ligne")

       { x=abs; y=ord ; }

   int coincide (point &) ;          // une fonction membre : coincide

} ;

int point::coincide (point & pt)

{  if ( (pt.x == x) && (pt.y == y) ) return 1 ;

                          else return 0 ;

}

main()                             // Un petit programme d'essai

{

   point a, b(1), c(1,0) ;

   cout << "a et b : " << a.coincide(b) << " ou " << b.coincide(a) << "\n" ;

   cout << "b et c : " << b.coincide(c) << " ou " << c.coincide(b) << "\n" ;

}
```

```
a et b : 0 ou 0

b et c : 1 ou 1
```

La remarque précédente sur les risques d'effets de bord s'applique également ici. Le qualificatif const pourrait y intervenir de manière analogue :

```
  int point::coincide (const point & pt)
```

Définition et utilisation d'une fonction membre statique :

```
#include <iostream>
using namespace std ;
class cpte_obj
{  static int ctr ;          // compteur (statique) du nombre d'objets créés
 public :
   cpte_obj () ;
   ~cpte_obj() ;
   static void compte () ;   // pour afficher le nombre d'objets créés
} ;
int cpte_obj::ctr = 0 ;      // initialisation du membre statique ctr
cpte_obj::cpte_obj ()        // constructeur
{
   cout << "++ construction : il y a maintenant   " << ++ctr << " objets\n" ;
}
cpte_obj::~cpte_obj ()       // destructeur
{  cout << "-- destruction  : il reste maintenant " << --ctr << " objets\n" ;
}
void cpte_obj::compte ()
{  cout << "   appel compte : il y a             " << ctr   << " objets\n" ;
}
main()
{  void fct () ;
```

```
  cpte_obj::compte () ;     // appel de la fonction membre statique compte
                            // alors qu'aucun objet de sa classe n'existe
  cpte_obj a ;
  cpte_obj::compte () ;
  fct () ;
  cpte_obj::compte () ;
  cpte_obj b ;
  cpte_obj::compte () ;
}
void fct()
{  cpte_obj u, v ;
}
  appel compte : il y a             0 objets
++ construction : il y a maintenant   1 objets
  appel compte : il y a             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
  appel compte : il y a             1 objets
++ construction : il y a maintenant   2 objets
  appel compte : il y a             2 objets
-- destruction  : il reste maintenant 1 objets
-- destruction  : il reste maintenant 0 objets
```

Autoréférence : le mot clé this :

Exemple d'utilisation de this :

```
#include <iostream>

using namespace std ;

class point                    // Une classe point contenant seulement :

{  int x, y ;

 public :

   point (int abs=0, int ord=0)     // Un constructeur ("inline")

       { x=abs; y=ord ; }

   void affiche () ;               // Une fonction affiche

} ;

void point::affiche ()

{  cout << "Adresse : " << this << " - Coordonnees " << x << " " << y << "\n" ;

}

main()                          // Un petit programme d'essai

{  point a(5), b(3,15) ;

   a.affiche ();

   b.affiche ();

}
```

```
Adresse : 006AFDF0 - Coordonnees 5 0

Adresse : 006AFDE8 - Coordonnees 3 15
```

Les membres mutables :

Exemple récapitulatif :

```
class truc
```

```
{ int x, y ;
  mutable int n ;    // n est modifiable par une fonction membre constante
  void f(.....)
    { x = 5 ; n++ ; }    // rien de nouveau ici

void f1(.....) const
    { n++ ;              // OK car n est déclaré mutable
      x = 5 ;            // erreur : f1 est const et x n'est pas mutable
    }
} ;
```

Comme on peut s'y attendre, les membres publics déclarés avec le qualificatif mutable son modifiables par affectation :

```
class truc2
{ public :
   int n ;
   mutable int p ;
   .....
} ;
  .....
const truc c ;
c.n = 5 ;   // erreur : l'objet c est constant et n n'est pas mutable
c.p = 5 ;   // OK : l'objet c est constant, mais p est mutable
```

Le constructeur de recopie :

Nous avons vu comment C++ garantissait l'appel d'un constructeur pour un objet créé par une déclaration ou par new. Ce point est fondamental puisqu'il donne la certitude qu'un objet ne pourra être créé sans avoir été placé dans un "état initial convenable". Mais il existe des circonstances dans lesquelles il est nécessaire de construire un objet, même si le programmeur n'a pas prévu de constructeur pour cela. La situation la plus fréquente est celle où la valeur d'un objet doit être transmise en argument à une fonction. Dans ce cas, il est nécessaire de créer, dans un emplacement local à la fonction, un objet qui soit une copie de l'argument effectif. Le même problème se pose dans le cas d'un objet renvoyé par valeur comme résultat d'une fonction ; il faut alors créer, dans un emplacement local à la fonction appelante, un objet qui soit une copie du résultat. On a aussi le cas où un objet est initialisé, lors de sa déclaration, avec un autre objet de même type.

D'une manière générale, on regroupe ces trois situations sous le nom *d'initialisation par recopie*. Une initialisation par recopie d'un objet est donc la création d'un objet par recopie d'un objet existant de même type. Pour réaliser une telle initialisation, C++ a prévu d'utiliser un constructeur particulier dit ***constructeur de recopie***.

Emploi du constructeur de recopie par défaut :

```
#include <iostream>

using namespace std ;

class vect

{  int nelem ;                  // nombre d'éléments

  double * adr ;                // pointeur sur ces éléments

 public :

  vect (int n)                  // constructeur "usuel"

  { adr = new double [nelem = n] ;

    cout << "+ const. usuel - adr objet : " << this

        << " - adr vecteur : " << adr << "\n" ;

  }

  ~vect ()                      // destructeur

   { cout << "- Destr. objet - adr objet : "
```

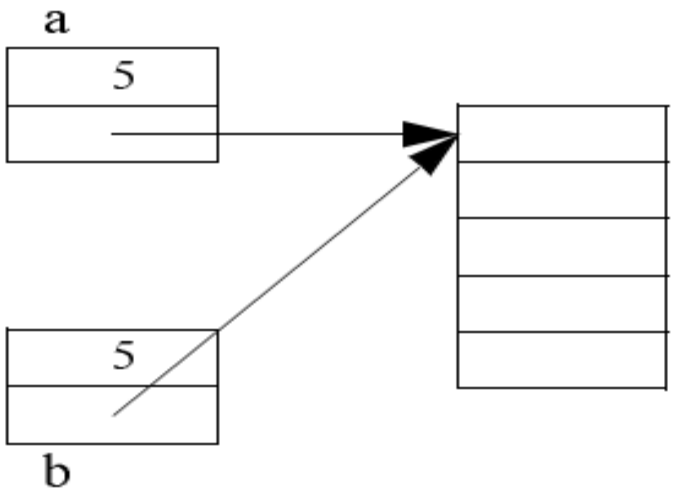
```
        << this << " - adr vecteur : " << adr << "\n" ;
    delete adr ;
  }
} ;
void fct (vect b)
{ cout << "*** appel de fct ***\n" ;
}
main()
{ vect a(5) ;
 fct (a) ;
}
```

```
+ const. usuel - adr objet : 006AFDE4 - adr vecteur : 007D0320
*** appel de fct ***
- Destr. objet - adr objet : 006AFD90 - adr vecteur : 007D0320
- Destr. objet - adr objet : 006AFDE4 - adr vecteur : 007D0320
```

Comme vous pouvez le constater, l'appel :

```
fct (a) ;
```

a créé un nouvel objet, dans lequel on a recopié les valeurs des membres nelem et adr de a. La situation peut être schématisée ainsi (b est le nouvel objet ainsi créé) :



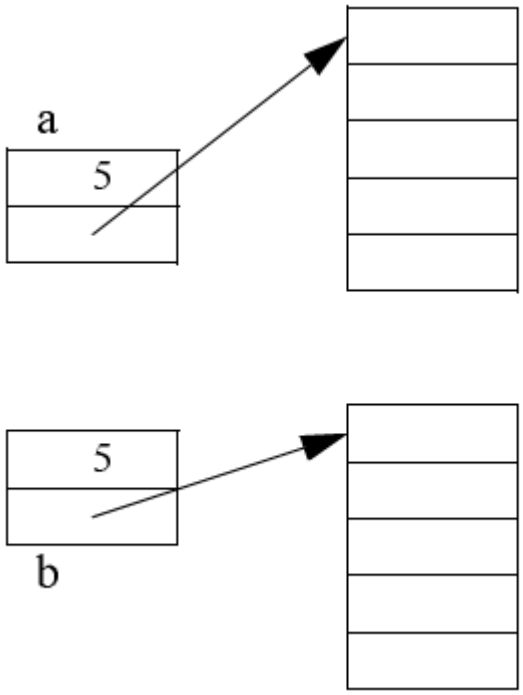
A la fin de l'exécution de la fonction fct, le destructeur ~point est appelé pour b, ce qui libère l'emplacement pointé par adr ; à la fin de l'exécution de la fonction main, le destructeur est appelé pour a, ce qui libère... le même emplacement. Cette tentative constitue une erreur d'exécution dont les conséquences varient avec l'implémentation.

Définition et utilisation d'un constructeur de recopie :

On peut éviter ce problème en faisant en sorte que l'appel :

fct (a) ;

conduise à créer "intégralement" un nouvel objet de type vect, avec ses membres données nelem et adr, mais aussi son propre emplacement de stockage des valeurs du tableau. Autrement dit, nous souhaitons aboutir à cette situation :



Pour ce faire, nous définissons, au sein de la classe vect, un constructeur par recopie de la forme :

vect (const vect &) ; // ou, a la rigueur vect (vect &)

dont nous savons qu'il sera appelé dans toute situation d'initialisation donc, en particulier, lors de l'appel de fct.

Ce constructeur (appelé après la création d'un nouvel objet) doit :

• créer dynamiquement un nouvel emplacement dans lequel il recopie les valeurs correspondant à l'objet reçu en argument,

• renseigner convenablement les membres données du nouvel objet (nelem = valeur du membre nelem de l'objet reçu en argument, adr = adresse du nouvel emplacement).

```
#include <iostream>

using namespace std ;

class vect

{

  int nelem ;                  // nombre d'éléments

  double * adr ;               // pointeur sur ces éléments
```

```
 public :
  vect (int n)                    // constructeur "usuel"
  { adr = new double [nelem = n] ;
    cout << "+ const. usuel   - adr objet : " << this
         << " - adr vecteur : " << adr << "\n" ;
  }

 vect (const vect & v)            // constructeur de recopie
  { adr = new double [nelem = v.nelem] ;              // création nouvel objet
    int i ; for (i=0 ; i<nelem ; i++) adr[i]=v.adr[i] ; // recopie de l'ancien
    cout << "+ const. recopie - adr objet : " << this
         << " - adr vecteur : " << adr << "\n" ;
  }
  ~vect ()                        // destructeur
  { cout << "- Destr. objet   - adr objet : "
         << this << " - adr vecteur : " << adr << "\n" ;
    delete adr ;
  }
} ;
void fct (vect b)
{ cout << "*** appel de fct ***\n" ; }
main()
{ vect a(5) ;   fct (a) ;
}
```

```
+ const. usuel   - adr objet : 006AFDE4 - adr vecteur : 007D0320

+ const. recopie - adr objet : 006AFD88 - adr vecteur : 007D0100

*** appel de fct ***

- Destr. objet   - adr objet : 006AFD88 - adr vecteur : 007D0100

- Destr. objet   - adr objet : 006AFDE4 - adr vecteur : 007D0320
```

Appel du constructeur de recopie en cas de transmission par valeur :

```
#include <iostream>

using namespace std ;

class point

{  int x, y ;

 public :

   point (int abs=0, int ord=0)     // constructeur "usuel"

   { x=abs ; y=ord ;

     cout << "++ Appel Const. usuel   " << this << " " << x << " " << y << "\n" ;

   }

   point (const point & p)          // constructeur de recopie

   { x=p.x ; y=p.y ;

     cout << "++ Appel Const. recopie " << this << " " << x << " " << y << "\n" ;

   }

   ~point ()

   { cout << "-- Appel Destr.        " << this << " " << x << " " << y << "\n" ;

   }

   point symetrique () ;

} ;
```

```
point point::symetrique ()
{ point res ; res.x = -x ; res.y = -y ; return res ;
}
main()
{   point a(1,3), b ;
    cout << "** avant appel de symetrique\n" ;
    b = a.symetrique () ;
    cout << "** apres appel de symetrique\n" ;
}
++ Appel Const. usuel   006AFDE4 1 3
++ Appel Const. usuel   006AFDDC 0 0
** avant appel de symetrique
++ Appel Const. usuel   006AFD60 0 0
++ Appel Const. recopie 006AFDD4 -1 -3
-- Appel Destr.        006AFD60 -1 -3
-- Appel Destr.        006AFDD4 -1 -3
** apres appel de symetrique
-- Appel Destr.        006AFDDC -1 -3
-- Appel Destr.        006AFDE4 1 3
```

Construction et initialisation d'un tableau d'objets :

```
#include <iostream>
using namespace std ;
class point
{   int x, y ;
```

```
 public :
  point (int abs=0, int ord=0)     // constructeur (0, 1 ou 2 arguments)
    { x=abs ; y =ord ;
      cout << "++ Constr. point : " << x << " " << y << "\n" ;
    }
  ~point ()
  { cout << "-- Destr. point : " << x << " " << y << "\n" ;
  }
} ;
main()
{  int n = 3 ;
  point courbe[5] = { 7, n, 2*n+5 } ;
  cout << "*** fin programme ***\n" ;
}
++ Constr. point : 7 0
++ Constr. point : 3 0
++ Constr. point : 11 0
++ Constr. point : 0 0
++ Constr. point : 0 0
*** fin programme ***
-- Destr. point : 0 0
-- Destr. point : 0 0
-- Destr. point : 11 0
-- Destr. point : 3 0
```

-- Destr. point : 7 0

Si l'on dispose d'une classe point, on peut créer dynamiquement un tableau de points en faisant appel à l'opérateur new. Par exemple :

```
point * adcourbe = new point[20] ;
```

Pour détruire notre tableau d'objets, il suffira de l'instruction (notez la présence des crochets [] qui précisent que l'on a affaire à un tableau d'objets) :

```
delete [] adcourbe
```

exercices :

Ecrivez une classe nommée pile_entier permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un tableau d'entiers alloués dynamiquement. La classe comportera les fonctions membres suivantes :

• pile_entier (int n) : constructeur allouant dynamiquement un emplacement de n en-

tiers,

• pile_entier ( ) :constructeur sans argument allouant par défaut un emplacement de

vingt entiers,

• ~pile_entier ( ): destructeur

• void empile (int p) : ajoute l'entier p sur la pile,

• int depile ( ) : fournit la valeur de l'entier situé en haut de la pile, en le supprimant

de la pile,

• int pleine ( ) : fournit 1 si la pile est pleine, 0 sinon,

• int vide ( ) : fournit 1 si la pile est vide, 0 sinon.

Ecrivez une fonction main utilisant des objets automatiques et dynamiques du type pile_entier défini précédemment.

Ajoutez à la classe pile_entier le constructeur de recopie permettant de régler les problèmes précédents.

Correction :

```
#include <iostream>

using namespace std;

class pile_entier

        {

        int nelet,dim;

        int* tab;

        public :

              pile_entier( int n=10 )

                {

                 tab=new int[dim=n];

                }

               pile_entier(const pile_entier & cons)

                  { cout<< "appel du cons copi";

                    tab=new int[cons.dim];

                  //   dim=cons.dim;

                     for(int i=0;i<cons.nelet;i++)

                           tab[i]=cons.tab[i];


                   }

              ~pile_entier()

                {

                cout<<" appel du destructeur "<<endl;

                system("pause");

                delete [] tab;
```

```
  }
void empile(int p)
  { cout<<nelet<<endl;
   if (plein()) cout<<" la pile est pleine "<<endl;
    else
        tab[nelet++]=p;
  }
int depile()
 {  int res;
   if(vide())
         exit(-1);
         else
           res=tab[nelet-1];

          delete ( tab + nelet-1);
           nelet--;
           return res;

 }
int plein()
  {
  return nelet>10?1:0;
  }
 int vide()
```

```
            {
                return nelet<=0 ? 1:0;
            }

    };
main()
{
 pile_entier p(7); int val; pile_entier q;
 for(int i=0;i<5;i++)
            {
            cout<<" donner le "<<i+1<<" ème elements de la pile"<<endl;
            cin>>val;
            p.empile(val);
            }
val=p.depile();
 cout<<" le dernier venu est :  "<<val<<endl;
q=p;

   for(int i=0;i<5;i++)
          { if (!q.vide())
          cout<<q.depile()<<endl;
          }
 system("pause");
}
```

Les fonctions amies :

La P.O.O. pure impose l'encapsulation des données. Nous avons vu comment la mettre en œuvre en C++ : les membres privés (données ou fonctions) ne sont accessibles qu'aux fonctions membres (publiques ou privées) et seuls les membres publics sont accessibles "de l'extérieur". Nous avons aussi vu qu'en C++ "l'unité de protection" est la classe, c'est-à-dire qu'une même fonction membre peut accéder à tous les objets de sa classe.

En revanche, ce même principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe. Or cette contrainte s'avère gênante dans certaines circonstances. Supposez par exemple que vous ayez défini une classe vecteur et une classe matrice. Il est probable que vous souhaiterez alors définir une fonction permettant de calculer le produit d'une matrice par un vecteur. Or, avec ce que nous connaissons actuellement de C++, nous ne pourrions définir cette fonction ni comme fonction membre de la classe vecteur, ni comme fonction membre de la classe matrice, et encore moins comme fonction indépendante (c'est-à-dire membre d'aucune classe). Bien entendu, vous pourriez toujours rendre publiques les données de vos deux classes, mais vous perdriez alors le bénéfice de leur protection. Vous pourriez également introduire dans les deux classes des fonctions publiques permettant d'accéder aux données, mais vous seriez alors pénalisé en temps d'exécution...

En fait, la notion de fonction amie propose une solution intéressante, sous la forme d'un compromis entre encapsulation formelle des données privées et des données publiques. Lors de la définition d'une classe, il est en effet possible de déclarer qu'une ou plusieurs fonctions (extérieures à la classe) sont des "amies" ; une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.

Il existe plusieurs situations d'amitiés :

• fonction indépendante, amie d'une classe,

• fonction membre d'une classe, amie d'une autre classe,

• fonction amie de plusieurs classes,

• toutes les fonctions membres d'une classe, amies d'une autre classe.

Exemple de fonction indépendante amie d'une classe :

```
#include <iostream>

using namespace std ;
```

```
class point
  { int x, y ;
  public :
    point (int abs=0, int ord=0)        // un constructeur ("inline")
          { x=abs ; y=ord ; }
      // déclaration fonction amie (indépendante) nommée coincide
    friend int coincide (point, point) ;      // on a accès ici aux membres pri-
                                              // vés de tout objet de type point
  } ;
int coincide (point p, point q)          // définition de coincide
{   if ((p.x == q.x) && (p.y == q.y)) return 1 ;
                          else return 0 ;
}
main()                                   // programme d'essai
{  point a(1,0), b(1), c ;
  if (coincide (a,b)) cout << "a coincide avec b \n" ;
              else cout << "a et b sont differents \n" ;
  if (coincide (a,c)) cout << "a coincide avec c \n" ;
              else cout << "a et c sont differents \n" ;
}
```

```
a coincide avec b
a et c sont differents
```

Bien que nous l'ayons placée ici dans la partie publique de point, nous vous rappelons que la déclaration d'amitié peut figurer n'importe où dans la classe.

Fonction membre d'une classe, amie d'une autre classe :

A titre indicatif, voici une façon de compiler nos deux classes A et B et la fonction f :

```
class A ;

class B

{    .....

     int f(char, A) ;

     .....

} ;

class A

{    .....

     friend int B::f(char, A) ;

     .....

} ;

int B::f(char..., A...)

{    .....

}
```

Fonction amie de plusieurs classes :

```
Class B ;

class A

{ // partie privée

.....

// partie publique

friend void f(A, B) ;

.....
```

```
} ;
 class B
 { // partie privée
      .....
  // partie publique
    friend void f(A, B) ;
      .....
 } ;
void f(A..., B...)
{  // on a accès ici aux membres privés
 // de n'importe quel objet de type A ou B
}
```

Toutes les fonctions d'une classe amies d'une autre classe :

C'est une généralisation du cas évoqué. On pourrait d'ailleurs effectuer autant de déclarations d'amitié qu'il y a de fonctions concernées. Mais il est plus simple d'effectuer une déclaration globale. Ainsi, pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on placera, dans la classe A, la déclaration :

```
 friend class B ;
```

exemples :

Produit d'une matrice par un vecteur à l'aide d'une fonction indépendante amie des deux classes :

```
using namespace std ;
class matrice ;       // pour pouvoir compiler la déclaration de vect
      // ********** La classe vect ******************
class vect
{   double v[3] ;     // vecteur à 3 composantes
```

```
 public :
  vect (double v1=0, double v2=0, double v3=0)   // constructeur
    { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ;
    }
  friend vect prod (matrice, vect) ;   // prod = fonction amie indépendante
 void affiche ()
    { int i ;
      for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
      cout << "\n" ;
    }
} ;
     // ********** La classe matrice ***************
class matrice
{   double mat[3] [3] ;      // matrice 3 X 3
 public :
  matrice (double t[3][3])    // constructeur, à partir d'un tableau 3 x 3
    { int i ; int j ;
      for (i=0 ; i<3 ; i++)
        for (j=0 ; j<3 ; j++)
          mat[i] [j] = t[i] [j] ;
    }
  friend vect prod (matrice, vect) ;   // prod = fonction amie indépendante
} ;
     // ********** La fonction prod ***************
```

```
vect prod (matrice m, vect x)
{  int i, j ;
   double som ;
   vect res ;     // pour le résultat du produit
   for (i=0 ; i<3 ; i++)
     { for (j=0, som=0 ; j<3 ; j++)
          som += m.mat[i] [j] * x.v[j] ;
       res.v[i] = som ;
     }
   return res ;
}
      // ********* Un petit programme de test ********
main()
{  vect w (1,2,3) ;
   vect res  ;
   double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
   matrice a =  tb  ;
   res = prod(a, w) ;
   res.affiche () ;
}
```

Produit d'une matrice par un vecteur à l'aide d'une fonction membre amie d'une autre classe :

```
#include <iostream>
using namespace std ;
      // ******** Déclaration de la classe matrice ***********
```

```
class vect ;                    // pour pouvoir compiler correctement

class matrice

{   double mat[3] [3] ;         // matrice 3 X 3

  public :

    matrice (double t[3][3])    // constructeur, à partir d'un tableau 3 x 3

      { int i ; int j ;

        for (i=0 ; i<3 ; i++)

          for (j=0 ; j<3 ; j++)

            mat[i] [j] = t[i] [j] ;

      }

    vect prod (vect) ;          // prod = fonction membre (cette fois)

} ;

      // ********* Déclaration de la classe vect **************

class vect

{   double v[3] ;      // vecteur à 3 composantes

  public :

    vect (double v1=0, double v2=0, double v3=0)   // constructeur

      { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }

    friend vect matrice::prod (vect) ;            // prod = fonction amie

    void affiche ()

      { int i ;

        for (i=0 ; i<3 ; i++) cout << v[i] << " " ;

        cout << "\n" ;

      }
```

```
} ;

      // ********* Définition de la fonction prod ***********

vect matrice::prod (vect x)

{  int i, j ;

  double som ;

  vect res ;     // pour le résultat du produit

  for (i=0 ; i<3 ; i++)

    { for (j=0, som=0 ; j<3 ; j++)

        som += mat[i] [j] * x.v[j] ;

      res.v[i] = som ;

    }

  return res ;

}

      // ********** Un petit programme de test *********

main()

{  vect w (1,2,3) ;

  vect res  ;

  double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;

  matrice a =  tb  ;

  res = a.prod (w) ;

  res.affiche () ;

}
```

La surdéfinition d'opérateurs :

C++ permet, dans certaines conditions, de surdéfinir des opérateurs. En fait, le langage C, comme beaucoup d'autres, réalise déjà la surdéfinition de certains opérateurs. Par exemple, dans une expression telle que :

a + b

le symbole + peut désigner, suivant le type de a et b :

• l'addition de deux entiers,

• l'addition de deux réels (float),

• l'addition de deux réels double précision (double),

•etc.

De la même manière, le symbole * peut, suivant le contexte, représenter la multiplication d'entiers ou de réels ou une "indirection" (comme dans a = * adr).

En C++, vous pourrez surdéfinir n'importe quel opérateur existant (unaire ou binaire) pour peu qu'il porte sur au moins un objet . Il s'agit là d'une technique fort puissante puisqu'elle va vous permettre de créer, par le biais des classes, des types à part entière, c'est-à-dire munis, comme les types de base, d'opérateurs parfaitement intégrés. La notation opératoire qui en découlera aura l'avantage d'être beaucoup plus concise et (du moins si l'on s'y prend "intelligemment" !) lisible qu'une notation fonctionnelle (par appel de fonction).

Par exemple, si vous définissez une classe complexe destinée à représenter des nombres complexes, il vous sera possible de donner une signification à des expressions telles que :

a + b a - b a * b a/b

Le mécanisme de la surdéfinition d'opérateurs :

```
#include <iostream>

using namespace std ;

class point

  {  int x, y ;

  public :

     point (int abs=0, int ord=0) { x=abs ; y=ord ;}  // constructeur
```

```
    friend point operator+ (point, point) ;
    void affiche () { cout << "coordonnees : " << x << " " << y << "\n"
  } ;
point operator + (point a, point b)
{  point p ;
  p.x = a.x + b.x ; p.y = a.y + b.y ;
  return p ;
}
main()
{  point a(1,2) ; a.affiche() ;
  point b(2,5) ; b.affiche() ;
  point c ;
  c = a+b ;     c.affiche() ;
  c = a+b+c ;   c.affiche() ;
}
coordonnees : 1 2
coordonnees : 2 5
coordonnees : 3 7
coordonnees : 6 14
```

Remarques :

1- Une expression telle que a+b est en fait interprétée par le compilateur comme l'appel :

```
operator + (a, b)
```

Bien que cela ne présente guère d'intérêt, nous pourrions écrire :

c = operator + (a, b)

au lieu de c=a+b.

2 - Une expression telle que a+b+c est évaluée en tenant compte des règles de priorité et d'associativité "habituelles" de l'opérateur +. Nous reviendrons plus loin sur ce point.

(a + b) + c

c'est-à-dire en utilisant la notation fonctionnelle :

operator + (operator + (a, b), c)

Surdéfinition de l'opérateur + pour des objets de type point, en employant une fonction membre :

```
#include <iostream>
using namespace std ;
class point
{  int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ;}  // constructeur
  point operator + (point) ;
  void affiche () { cout << "coordonnees : " << x << " " << y << "\n" ; }
} ;
point point::operator + (point a)
{  point p ;  // création d'un objet
  p.x = x + a.x ; p.y = y + a.y ;  // x de l'objet en cours + celle de l'argument formel
  return p ;
}
main()
{  point a(1,2) ; a.affiche() ;
```

```
  point b(2,5) ; b.affiche() ;
  point c ;
  c = a+b ;      c.affiche() ;  //  est équivalent à c= a.operator + (b) ;
  c = a+b+c ;    c.affiche() ;
}
```

```
coordonnees : 1 2
coordonnees : 2 5
coordonnees : 3 7
coordonnees : 6 14
```

Le symbole suivant le mot clé operator doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est donc pas possible de créer de nouveaux symboles. Nous verrons d'ailleurs que certains opérateurs ne peuvent pas être redéfinis du tout (c'est le cas de .) et que d'autres imposent quelques contraintes supplémentaires.

Pluralité	Opérateurs	Associativité
Binaire	()(3) [](3) ->(1)(2)(3)	->
Unaire	+ - ++(5) --(5) ! ~ * &(1) new(1)(4)(6) new[](1)(4)(6) delete(1)(4)(6) delete[](1)(4)(6) (cast)	<-
Binaire	* / %	->
Binaire	*->(1) .*(1)	->
Binaire	+ -	->
Binaire	<< >>	->
Binaire	< <= > >=	->
Binaire	== !=	->
Binaire	&	->
Binaire	^	->
Binaire	||	->
Binaire	&&	->
Binaire	|	->
Binaire	=(1)(3) += -= *= /= %= &= ^= |= <<= >>=	<-
Binaire	,(2)	->

Les opérateurs surdéfinissables en C++ (classés par priorité décroissante)

Exemple de surdéfinition de ++ en notation préfixée et postfixée :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  point  operator ++ ()        // notation préfixée
```

```
   { x++ ; y++ ; return *this ;
   }
 point  operator ++ (int n)    // notation postfixée
   { point p = *this ;
     x++ ; y++ ;
     return p ;
   }
 void affiche () { cout << x << " " << y << "\n" ; }
} ;
main()
{ point a1 (2, 5), a2(2, 5), b ;
 b = ++a1 ; cout << "a1 : " ; a1.affiche () ;   // affiche    a1 : 3 6
          cout << "b  : " ; b.affiche () ;    // affiche    b  : 3 6
 b = a2++ ; cout << "a2 : " ; a2.affiche () ;   // affiche    a2 : 3 6
          cout << "b  : " ; b.affiche () ;    // affiche    b  : 2 5
}
```

Exemple de surdéfinition de l'opérateur[] :

```
#include <iostream>
using namespace std ;
class vect
{  int nelem ;
   int * adr ;
 public :
   vect (int n) { adr = new int [nelem=n] ; }
```

```
  ~vect () {delete adr ;}
  int & operator [] (int) ;
} ;
int & vect::operator [] (int i)
{ return adr[i] ; }
main()
{  int i ;
  vect a(3), b(3), c(3) ;
  for (i=0 ; i<3 ; i++) {a[i] = i ; b[i] = 2*i ; }
  for (i=0 ; i<3 ; i++) c[i] = a[i]+b[i] ;
  for (i=0 ; i<3 ; i++) cout << c[i] << " " ;
}
```

Exemple d'utilisation d'un opérateur de cast pour la conversion point -> int :

```
#include <iostream>
using namespace std ;
class point
{  int x, y ;
 public :
  point (int abs=0, int ord=0)          // constructeur 0, 1 ou 2 arguments
    { x = abs ; y = ord ;
      cout << "++ construction point : " << x << " " << y << "\n" ;
    }
  operator int()                        // "cast" point --> int
    { cout << "== appel int() pour le point " << x << " " << y << "\n" ;
```

```
      return x ;
    }
} ;
main()
{  point a(3,4), b(5,7) ;
  int n1, n2 ;
  n1 = int(a) ; // ou  n1 = (int) a      appel explicite de int ()
                // on peut aussi écrire :  n1 = (int) a  ou n1 = static_cast<int> (a)
  cout << "n1 = " << n1 << "\n" ;
  n2 = b ;                          // appel implicite de int()
  cout << "n2 = " << n2 << "\n" ;
}
```

```
++ construction point : 3 4
++ construction point : 5 7
== appel int() pour le point 3 4
n1 = 3
== appel int() pour le point 5 7
n2 = 5
```

Exercices :

Création d'une classe complexe : on surdéfinit les opérateurs + (avec une onction ami), -(avec un membre).

Correction :

Fichier en-tête : dec_complex

```
#include <iostream>
```

```
using namespace std;
class complexe
  {
     float x,y;
     public :
       complexe( float r=0, float i=0)
            {
            this->x=r; this->y=i;
            }
     friend complexe operator +(complexe a,complexe b)
          { complexe p;
               p.x=a.x+b.x; p.y=b.y+a.y;
          return p;
          }
   complexe operator-(complexe p)
       {
        complexe a;
         a.x= x - p.x;
         a.y=y - p.y;
       return a;
       }
      void affiche()
        {
         if (y>0)
```

```
            cout<<endl<<" nombre : "<<this->x<<" + i "<<y<<endl;
             else
             cout<<endl<<" nombre : "<<this->x<<" - i "<<-y<<endl;
          }
  };

float pr,pi;
complexe c;
```

fonction main() :

```
#include <iostream>
#include "dec_complex.h"
using namespace std;
main()
{

cout<<" donner la partie reelle "<<endl;
cin>>pr;
cout<<" donner la partie immaginaire "<<endl;
cin>>pi;
complexe a(pr,pi);
cout<<" deuxieme nombre"<<endl;
cout<<" donner la partie reelle "<<endl;
cin>>pr;
cout<<" donner la partie immaginaire "<<endl;
```

```
cin>>pi;
complexe b(pr,pi);
c=a+b;
c.affiche();
c=a-b;
c.affiche();
system("pause");
}
```

Les patrons de fonctions :

Création d'un patron de fonctions :

Supposons que nous ayons besoin d'écrire une fonction fournissant le minimum de deux valeurs de même type reçues en arguments. Nous pourrions écrire une définition pour le type int :

```
int min (int a, int b)
{
  if (a < b) return a ;   // ou return a < b ? a : b ;
       else return b ;
}
```

Bien entendu, il nous faudrait probablement écrire une autre définition pour le type float, c'est-à-dire (en supposant que nous lui donnions le même nom min, ce que nous avons tout intérêt à faire) :

```
float min (float a, float b)
{
  if (a < b) return a ;   // ou return a < b ? a : b ;
       else return b ;
}
```

En fait, nous pouvons simplifier considérablement les choses en définissant un seul patron de fonctions, de la manière suivante :

```
   // création d'un patron de fonctions
template <class T> T min (T a, T b)
{
  if (a < b) return a ;   // ou return a < b ? a : b ;
      else return b ;
}
```

Comme vous le constatez, seul l'en-tête de notre fonction a changé (il n'en ira pas toujours ainsi) : template <class T> T min (T a, T b). La mention template <class T> précise que l'on a affaire à un patron (template) dans lequel apparaît un "paramètre de type" nommé T. Notez que C++ a décidé d'employer le mot clé class pour préciser que T est un paramètre de type (on aurait préféré le mot clé type !). Autrement dit, dans la définition de notre fonction, T représente un type quelconque.

Le reste de l'en-tête : T min (T a, T b) précise que min est une fonction recevant deux arguments de type T et fournissant un résultat du même type.

Remarque :

Dans la définition d'un patron, on utilise le mot clé class pour indiquer en fait un type quelconque, classe ou non. La norme a introduit le mot clé typename qui peut se subtituer à class dans la définition :

```
template <typename T> T min (T a, T b) { ..... }  // idem template <class T>
```

Cependant, son arrivée tardive fait que la plupart des programmes continuent d'utiliser le mot clé class dans ce cas.

Définition et utilisation d'un patron de fonctions :

```
#include <iostream>
using namespace std ;
   // création d'un patron de fonctions
```

```
template <class T> T min (T a, T b)
{   if (a < b) return a ;   // ou return a < b ? a : b ;
        else return b ;
}
  // exemple d'utilisation du patron de fonctions min
main()
{
  int n=4, p=12 ;
  float x=2.5, y=3.25 ;
  cout << "min (n, p) = " << min (n, p) << "\n" ;  //  int min(int, int)
  cout << "min (x, y) = " << min (x, y) << "\n" ;    //  float min (float, float)
}
```

```
min (n, p) = 4
min (x, y) = 2.5
```

En pratique, on placera les définitions de patron dans un fichier approprié d'extension h.La norme a introduit le mot clé export. Appliqué à la définition d'un patron, il précise que celle-ci sera accessible depuis un autre fichier source. Par exemple, en écrivant ainsi notre patron de fonctions min du patron :

```
export template <class T> T min (T a, T b)
{   if (a < b) return a ;   // ou return a < b ? a : b ;
        else return b ;
}
```

on peut alors utiliser ce patron depuis un autre fichier source, en se contentant de mentionner sa "déclaration" (cette fois, il s'agit bien d'une véritable déclaration et non plus d'une définition) :

```
template <class T> T min (T a, T b) ;  // déclaration seule de min
```

.....

min (x, y)

En pratique, on aura alors intérêt à prévoir deux fichiers en-têtes distincts, un pour la déclaration, un pour la définition. On pourra à volonté inclure le premier, dans la définition du patron, ou dans son utilisation.

Un patron de fonctions peut donc comporter un ou plusieurs paramètres de type, chacun devant être précédé du mot clé class par exemple :

```
template <class T, class U> fct (T a, T * b, U c)
{ ...
}
```

Exemple de patron de fonctions comportant un paramètre expression (n) :

```
#include <iostream>
using namespace std ;
template <class T> int compte (T * tab, int n)
{  int i, nz=0 ;
   for (i=0 ; i<n ; i++) if (!tab[i]) nz++ ;
   return nz ;
}
main ()
{ int t [5] = { 5, 2, 0, 2, 0} ;
  char c[6] = { 0, 12, 0, 0, 0, 5} ;
  cout << "compte (t) = " << compte (t, 5) << "\n" ;
  cout << "compte (c) = " << compte (c, 6) << "\n" ;
}
```

compte (t) = 2

compte (c) = 4

Les patrons de classes :

Le précédent chapitre a montré comment C++ permettait, grâce à la notion de patron de fonctions, de définir une famille de fonctions paramétrées par un ou plusieurs types, et éventuellement des expressions. D'une manière comparable, C++ permet de définir des "patrons de classes". Là encore, il suffira d'écrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter à différents types.

Création d'un patron de classes :

Nous avons souvent été amené à créer une classe point de ce genre (nous ne fournissons pas ici la définition des fonctions membres) :

```
class point
{  int x ; int y ;
 public :
  point (int abs=0, int ord=0) ;
  void affiche () ;
  // .....
}
```

Lorsque nous procédons ainsi, nous imposons que les coordonnées d'un point soient des valeurs de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type (float, double, long, unsigned int...), nous devons définir une autre classe en remplaçant simplement, dans la classe précédente, le mot clé int par le nom de type voulu.

Ici encore, nous pouvons simplifier considérablement les choses en définissant un seul patron de classe de cette façon :

```
template <class T> class point
{  T x ; T y ;
 public :
  point (T abs=0, T ord=0) ;
```

```
  void affiche () ;
} ;
```

Exemple :

```
#include <iostream>
using namespace std ;
 // création d'un patron de classe
template <class T> class point
{  T x ; T y ;
 public :
  point (T abs=0, T ord=0)
  {  x = abs ; y = ord ;
  }
  void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{  cout << "Coordonnées : " << x << " " << y << "\n" ;
}
```

Utilisation d'un patron de classes :

Après avoir créé ce patron, une déclaration telle que :

```
point <int> ai ;
```

conduit le compilateur à instancier la définition d'une classe point dans laquelle le paramètre T prend la valeur int. Autrement dit, tout se passe comme si nous avions fourni une définition complète de cette classe.

Si nous déclarons :

```
point <double> ad ;
```

Le compilateur instancie la définition d'une classe point dans laquelle le paramètre T prend la valeur double, exactement comme si nous avions fourni une autre définition complète de cette classe.

```
point <int> ai (3, 5) ;

point <double> ad (3.5, 2.3) ;
```

Création et utilisation d'un patron de classes :

```
#include <iostream>

using namespace std ;

 // création d'un patron de classe

template <class T> class point

{

  T x ; T y ;

 public :

  point (T abs=0, T ord=0)

  {  x = abs ; y = ord ;

  }

  void affiche () ;

} ;

template <class T> void point<T>::affiche ()

{

 cout << "Coordonnees : " << x << " " << y << "\n" ;

}

main ()

{

  point <int> ai (3, 5) ;       ai.affiche () ;
```

```
  point <char> ac ('d', 'y') ;   ac.affiche () ;
  point <double> ad (3.5, 2.3) ; ad.affiche () ;
}
coordonnees : 3 5
coordonnees : d y
coordonnees : 3.5 2.3
```

Les paramètres de type dans la création d'un patron de classes :

Les paramètres de type peuvent être en nombre quelconque et utilisés comme bon vous semble dans la définition du patron de classes. En voici un exemple :

```
template <class T, class U, class V> // liste de trois param. de nom (muet) T, U et V
class essai
{  T x ;            // un membre x de type T
  U t[5]      ;    // un tableau t de 5 éléments de type U
    ...
  V fm1 (int, U) ;  // déclaration d'une fonction membre recevant 2 arguments
                    //  de type int et U et renvoyant un résultat de type V
    ...
} ;
```

Comme utilisation on peut faire :

```
essai <int, float, int> ce1 ;
essai <int, int *, double > ce2 ;
essai <char *, int, obj>  ce3 ;
```

Il est même possible d'utiliser comme paramètre de type effectif un type instancié à l'aide d'un patron de classes. Par exemple, si nous disposons du patron de classes nommé point tel qu'il a été défini dans le paragraphe précédent, nous pouvons déclarer :

```
essai <float, point<int>, double> ce4 ;

essai <point<int>, point<float>, char *> ce5 ;
```

Exemple de classe patron comportant un paramètre expression

```
#include <iostream>

using namespace std ;

template <class T, int n> class tableau

{  T tab [n] ;

  public :

   tableau () { cout << "construction tableau \n" ; }

   T & operator [] (int i)

    { return tab[i] ;

    }

} ;

class point

{  int x, y ;

  public :

   point (int abs=1, int ord=1 )   // ici init par défaut à 1

     { x=abs ; y=ord ;

       cout << "constr point " << x << " " << y << "\n" ;

     }

   void affiche () { cout << "Coordonnees : " << x << " " << y << "\n" ; }

} ;

main()

{  tableau <int,4> ti ;
```

```
  int i ; for (i=0 ; i<4 ; i++) ti[i] = i ;
  cout << "ti : " ;
  for (i=0 ; i<4 ; i++) cout << ti[i] << " " ;
  cout << "\n" ;
  tableau <point, 3> tp ;
  for (i=0 ; i<3 ; i++) tp[i].affiche() ;
}
construction tableau
ti : 0 1 2 3
const point 1 1
const point 1 1
const point 1 1
construction tableau
coordonnées : 1 1
coordonnées : 1 1
coordonnées : 1 1
```

Exemple de spécialisation d'une fonction membre d'une classe patron

```
#include <iostream>
using namespace std ;
 // création d'un patron de classe
template <class T> class point
{  T x ; T y ;
 public :
  point (T abs=0, T ord=0)
```

```
  {  x = abs ; y = ord ;
  }
  void affiche () ;
} ;
 // définition de la fonction affiche
template <class T> void point<T>::affiche ()
{ cout << "Coordonnees : " << x << " " << y << "\n" ;
}
 // ajout d'une fonction affiche spécialisée pour les caractères
void point<char>::affiche ()
{  cout << "Coordonnees : " << (int)x << " " << (int)y << "\n" ;
}
main ()
{  point <int> ai (3, 5) ;       ai.affiche () ;
  point <char> ac ('d', 'y') ;   ac.affiche () ;
  point <double> ad (3.5, 2.3) ; ad.affiche () ;
}
```

```
coordonnées : 3 5
coordonnées : 100 121
coordonnées : 3.5 2.3
```

Paramètres par défaut :

Dans la définition d'un patron de classes, il est possible de spécifier des valeurs par défaut pour certains paramètres , suivant un mécanisme semblable à celui utilisé pour les paramètres de fonctions usuelles. Voici quelques exemples :

```
template <class T, class U=float> class A { ..... } ;
```

```
template <class T, int n=3>       class B { ..... } ;

  .....

A<int,long> a1 ;     /* instanciation usuelle          */

A<int> a2 ;          /* équivaut à A<int, float> a2 ;  */

B<int, 3> b1 ;       /* instanciation usuelle          */

B<int> b2 ;          /* équivaut à B<int, 3> b2 ;      */
```

Déclaration d'un autre patron de fonctions ou de class :

Voici un exemple faisant appel aux mêmes patrons point et fct que ci-dessus :

```
template <class T, class U>

class essai2

{ int x ;

 public :

  template <class X> friend class point <X> ;

  template <class X> friend class int fct (point <X>) ;

} ;
```

Cette fois, toutes les instances du patron point sont amies de n'importe quelle instance du patron essai2. De même, toutes les instances du patron de fonctions fct sont amies de n'importe quelle instance du patron essai.

Exercices :

Patron de classe qui gère à la fois les complexes et les points.

Correction :

En –tête : **dec_patron.h**

```
#include <iostream>

using namespace std;

template <class U>
```

```
class dim_deux
{
   U x,y;
   public :
    dim_deux( U r=0, U i=0)
          {
          this->x=r; this->y=i;
          }
  friend dim_deux<U> operator +(dim_deux<U> a,dim_deux<U> b)
         { dim_deux<U> p;
               p.x=a.x+b.x; p.y=b.y+a.y;
         return p;
         }
dim_deux<U> operator-(dim_deux<U> p)
      {
       dim_deux<U> a;
        a.x= x - p.x;
        a.y=y - p.y;
      return a;
      }
   void affiche(int n=0)
       {  if ( n==0)
            {
        if (y>0)
```

```
cout<<endl<<" nombre : "<<this->x<<" + i "<<y<<endl;
 else
 cout<<endl<<" nombre : "<<this->x<<" - i "<<-y<<endl;
   }
   else
      {
       if (y>0)
cout<<endl<<" point : "<<"( "<<this->x<<" , "<<y<<" )"<<endl;
 else
cout<<endl<<" point : "<<"( "<<this->x<<" , "<<y<<" )"<<endl;

      }

  }
};
```

Fichier source :

```
#include <iostream>
#include "dec_patron.h"
using namespace std;
template <class U> class dim_deux;
main()
{
float pr,pi;
int abs,ord;
```

```
dim_deux<float> c;

dim_deux<int> f;

cout<<" manipulation de complexe "<<endl;

cout<<" donner la partie reelle "<<endl;

cin>>pr;

cout<<" donner la partie immaginaire "<<endl;

cin>>pi;

dim_deux<float> a(pr,pi);

cout<<" deuxieme nombre"<<endl;

cout<<" donner la partie reelle "<<endl;

cin>>pr;

cout<<" donner la partie immaginaire "<<endl;

cin>>pi;

dim_deux<float> b(pr,pi);

c=a+b;

c.affiche();

c=a-b;

c.affiche();

cout<<" manipulation de points "<<endl;

cout<<" donner l'abscisse "<<endl;

cin>>abs;

cout<<" donner l ' ordonnee "<<endl;

cin>>ord;

dim_deux<int> d(abs,ord);
```

```
cout<<" deuxieme point "<<endl;

cout<<" donner l'abscisse  "<<endl;

cin>>abs;

cout<<" donner l'ordonnee "<<endl;

cin>>ord;

dim_deux<int> e(abs,ord);

f=e+d;

f.affiche(2);

f=d-e;

f.affiche(2);

system("pause");

}
```

L'héritage simple

On sait que le concept d'héritage (on parle également de classes dérivées) constitue l'un des fondements de la P.O.O. En particulier, il est à la base des possibilités de réutilisation de composants logiciels (en l'occurrence, de classes). En effet, il vous autorise à définir une nouvelle classe, dite "dérivée", à partir d'une classe existante dite "de base". La classe dérivée "héritera" des "potentialités" de la classe de base, tout en lui en ajoutant de nouvelles, et cela sans qu'il soit nécessaire de remettre en question la classe de base. Il ne sera pas utile de la recompiler, ni même de disposer du programme source correspondant (exception faite de sa déclaration).

Considérons la première classe point, dont nous rappelons la déclaration :

```
    /* ------------ Déclaration de la classe point ------------- */

class point

{           /* déclaration des membres privés */

   int x ;

   int y ;
```

```
            /* déclaration des membres publics */
 public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

Une classe pointcol, dérivée de point

```
class pointcol : public point        // pointcol dérive de point
{  short couleur ;
 public :
   void colore (short cl)
       { couleur = cl ; }
} ;
```

Notez la déclaration :

```
class pointcol : public point
```

Elle spécifie que pointcol est une classe dérivée de la classe de base point. De plus, le mot *public* signifie que les membres publics de la classe de base (point) seront des membres publics de la classe dérivée (pointcol) ; cela correspond à l'idée la plus fréquente que l'on peut avoir de l'héritage, sur le plan général de la P.O.O.

Une nouvelle classe pointcol et son utilisation :

```
#include <iostream>
#include "point.h"    /* déclaration de la classe point  (nécessaire */
                      /*   pour compiler la définition de pointcol)  */
using namespace std ;
class pointcol : public point
```

```
{  short couleur ;
 public :
   void colore (short cl)
       { couleur = cl ; }
   void affichec () ;
   void initialisec (int, int, short) ;
} ;
void pointcol::affichec ()
{  affiche () ;
   cout << "     et ma couleur est : " << couleur << "\n" ;
}
void pointcol::initialisec (int abs, int ord, short cl)
{  initialise (abs, ord) ;
   couleur = cl ;
}
main()
{
  pointcol p ;
  p.initialisec (10,20, 5) ; p.affichec () ; p.affiche () ;
  p.deplace (2,4) ;          p.affichec () ;
  p.colore (2) ;             p.affichec () ;
}
```

```
Je suis en 10 20
     et ma couleur est : 5
```

```
Je suis en 10 20

Je suis en 12 24

    et ma couleur est : 5

Je suis en 12 24

    et ma couleur est : 2
```

Redéfinition des fonctions membres d'une classe dérivée :

Une classe pointcol dans laquelle les méthodes initialise et affiche sont redéfinies :

```
#include <iostream>

#include "point.h"

using namespace std ;

class pointcol : public point

{  short couleur ;

 public :

   void colore (short cl)

       { couleur = cl ; }

   void affiche () ;                  // redéfinition de affiche de point

   void initialise (int, int, short) ;  // redéfinition de initialise de point

} ;

void pointcol::affiche ()

{  point::affiche () ;              // appel de affiche de la classe point

   cout << "     et ma couleur est : " << couleur << "\n" ;

}

void pointcol::initialise (int abs, int ord, short cl)

{  point::initialise (abs, ord) ;    // appel de initialise de la classe point
```

```
  couleur = cl ;
}
main()
{ pointcol p ;
 p.initialise (10,20, 5) ; p.affiche () ;
 p.point::affiche () ;            // pour forcer l'appel de affiche de point
 p.deplace (2,4) ;        p.affiche () ;
 p.colore (2) ;           p.affiche () ;
}
```

```
Je suis en 10 20
    et ma couleur est : 5
Je suis en 10 20
Je suis en 12 24
    et ma couleur est : 5
Je suis en 12 24
    et ma couleur est : 2
```

Redéfinition des membres données d'une classe dérivée :

Bien que cela soit d'un emploi moins courant, ce que nous avons dit à propos de la redéfinition des fonctions membres s'applique tout aussi bien aux membres données. Plus précisément, si une classe A est définie ainsi :

```
class A
{    .....
     int a ;
     char b ;
     .....
```

```
} ;
```

une classe B dérivée de A pourra, par exemple, définir un autre membre donnée nommé a :

```
class B : public A
{    float a ;
     .....
} ;
```

Dans ce cas, si l'objet b est de type B, b.a fera référence au membre a de type float de b. Il sera toujours possible d'accéder au membre donnée a de type int (hérité de A) par b.A::a . Notez bien que le membre a défini dans B s'ajoute au membre a hérité de A ; il ne le remplace pas.

Dans ce dernier cas, on voit qu'une redéfinition d'une méthode dans une classe dérivée cache

en quelque sorte les autres. Voici un dernier exemple :

```
class A
{ public :
  void f(int n)  { ..... }
  void f(char c) { ..... }
} ;
class B : public A
{ public :
  void f(int, int) { ..... }
main()
{ int n ; char c ; B b ;
  b.f(n) ;   // erreur de compilation
  b.f(c) ;   // erreur de compilation
}
```

Ici, pour les appels b.f(n) et b.f(c), le compilateur n'a considéré que l'unique fonction f(int, int) de B, laquelle ne convient manifestement pas.

En résumé :

Lorsqu'une fonction membre est définie dans une classe, elle masque toutes les fonctions membres de même nom de la classe de base (et des classes ascendantes). Autrement dit, la recherche d'une fonction (surdéfinie ou non) se fait dans une seule portée, soit celle de la classe concernée, soit celle de la classe de base (ou d'une classe ascendante), mais jamais dans plusieurs classes à la fois.

Remarque :

Il est possible d'imposer que la recherche d'une fonction surdéfinie se fassse dans plusieurs classes en utilisant une directive using. Par exemple, si dans la classe A précédente, on introduit (à un niveau public) l'instruction :

```
using A::f ;   // on réintroduit les fonctions f de A
```

l'instruction b.f(c) conduira alors à l'appel de A::f(char) (le comportement des autres appels restant, ici, le même).

Statut dans la classe de base	Accès aux fonctions membres et amies de la classe dérivée	Accès à un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée, en cas de nouvelle dérivation
public	oui	oui	public
protégé	oui	non	protégé
privé	non	non	privé

La dérivation publique

Les membres protégés restent inaccessibles à l'utilisateur de la classe, pour qui ils apparaissent comme des membres privés. Mais ils seront accessibles aux membres d'une éventuelle classe dérivée, tout en restant dans tous les cas inaccessibles aux utilisateurs de cette classe.

Transmission d'informations entre constructeurs :

Toutefois, un problème se pose lorsque le constructeur de A nécessite des arguments. En effet, les informations fournies lors de la création d'un objet de type B sont a priori destinés à son constructeur ! En fait, C++ a prévu la possibilité de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre à un constructeur de la classe de base. Le

mécanisme est le même que celui que nous vous avons exposé dans le cas des objets membres. Par exemple, si l'on a

ceci :

```
class point                                    class pointcol : public point
{    .....                                     {    .....
 public :                                                public :
    point (int, int) ;                                          pointcol (int, int, char) ;
    .....                                                       .....
} ;                                            } ;
```

et que l'on souhaite que pointcol retransmette à point les deux premières informations reçues,

on écrira son en-tête de cette manière :

```
pointcol (int abs, int ord, char cl) : point (abs, ord)
```

Le compilateur mettra en place la transmission au constructeur de point des informations abs et ord correspondant (ici) aux deux premiers arguments de pointcol. Ainsi, la déclaration :

```
pointcol a (10, 15, 3) ;
```

entraînera :

•l'appel de point qui recevra les arguments 10 et 15,

•l'appel de pointcol qui recevra les arguments 10, 15 et 3.

En revanche, la déclaration :

```
pointcol q (5, 2)
```

sera rejetée par le compilateur puisqu'il n'existe aucun constructeur pointcol à deux arguments. Bien entendu, il reste toujours possible de mentionner des arguments par défaut dans pointcol, par exemple :

```
pointcol (int abs = 0, int ord = 0, char cl = 1) : point (abs, ord)
```

Dans ces conditions, la déclaration :

```
pointcol b (5) ;
```

entraînera :

• l'appel de point avec les arguments 5 et 0,

• l'appel de pointcol avec les arguments 5, 0 et 1.

```
#include <iostream>
using namespace std ;
class point
{ protected :          // pour que x et y soient accessibles à pointcol
  int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  void affiche ()
    { cout << "Je suis un point \n" ;
      cout << "   mes coordonnees sont : " << x << " " << y << "\n" ;
    }
} ;
class pointcol : public point
{   short couleur ;
 public :
  pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
    { couleur = cl ;
    }
  void affiche ()
    { cout << "Je suis un point colore \n" ;
```

```
      cout << "   mes coordonnees sont : " << x << " " << y ;
      cout << "   et ma couleur est :   " << couleur << "\n" ;
    }
} ;
main()
{  point p(3,5) ; point * adp = &p ;
  pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
  adp->affiche () ; adpc->affiche () ;
  cout << "-----------------\n" ;
  adp = adpc ;          //  adpc = adp serait rejeté
  adp->affiche () ; adpc->affiche () ;
}
```

```
Je suis un point
  mes coordonnees sont : 3 5
Je suis un point colore
  mes coordonnees sont : 8 6   et ma couleur est :   2
-----------------
Je suis un point
  mes coordonnees sont : 8 6
Je suis un point colore
  mes coordonnees sont : 8 6   et ma couleur est :   2
```

Pour forcer l'appel d'un constructeur de recopie de la classe de base :

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

```
class point
{  int x, y ;
 public :
  point (int abs=0, int ord=0)        // constructeur usuel
    { x = abs ; y = ord ;
      cout << "++ point    " << x << " " << y << "\n" ;
    }
  point (point & p)                   // constructeur de recopie
    { x = p.x ; y = p.y ;
      cout << "CR point    " << x << " " << y << "\n" ;
    }
} ;
class pointcol : public point
{  char coul ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) // constr usuel
    { coul = cl ;
      cout << "++ pointcol " << int(coul) << "\n" ;
    }
  pointcol (pointcol & p) : point (p)  // constructeur de recopie
                                       // il y aura conversion implicite
                                       // de p dans le type point
    { coul = p.coul ;
      cout << "CR pointcol " << int(coul) << "\n" ;
```

```
    }
} ;
void fct (pointcol pc)
{  cout << "*** entree dans fct ***\n" ;
}
main()
{  void fct (pointcol) ;
   pointcol a (2,3,4) ;
   fct (a) ;              // appel de fct, à qui on transmet a par valeur
}
++ point    2 3
++ pointcol 4
CR point    2 3
CR pointcol 4
*** entree dans fct ***
```

Quand la classe de base et la classe dérivée surdéfinissent l'opérateur = :

```
#include <iostream>
using namespace std ;
class point
{  protected :
     int x, y ;
  public :
     point (int abs=0, int ord=0) { x=abs ; y=ord ;}
     point & operator = (point & a)
```

```
    { x = a.x ; y = a.y ;
      cout << "operateur = de point \n" ;
      return * this ;
    }
} ;
class pointcol : public point
{  protected :
   int coul ;
  public :
   pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) { coul=cl ; }
   pointcol & operator = (pointcol & b)
    { coul = b.coul ;
      cout << "operateur = de pointcol\n" ;
      return * this ;
    }
   void affiche ()
    { cout << "pointcol : " << x << " " << y << " " << coul << "\n" ;
    }
} ;
main()
{ pointcol p(1, 3, 10) , q(4, 9, 20) ;
 cout << "p      = " ; p.affiche () ;
 cout << "q avant = " ; q.affiche () ;
 q = p ;
```

```
 cout << "q apres = " ; q.affiche () ;
}
p      = pointcol : 1 3 10
q avant = pointcol : 4 9 20
operateur = de pointcol
q apres = pointcol : 4 9 10
```

On voit clairement que l'opérateur = défini dans la classe point n'a pas été appelé lors d'une affectation entre objets de type pointcol.

Comment forcer, dans une classe dérivée, l'utilisation de l'opérateur = surdéfini dans la classe de base :

```
#include <iostream>
using namespace std ;
class point
{ protected :
    int x, y ;
  public :
    point (int abs=0, int ord=0) { x=abs ; y=ord ;}
    point & operator = (point & a)
     {  x = a.x ; y = a.y ;
       cout << "operateur = de point \n" ;
       return * this ;
     }
} ;
class pointcol : public point
{ protected :
```

```
  int coul ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord) { coul=cl ; }
  pointcol & operator = (pointcol & b)
   { point * ad1, * ad2 ;
     cout << "operateur = de pointcol\n" ;
      ad1 = this ;    // conversion pointeur sur pointcol en pointeur sur point
     ad2 = & b ;     //   idem
     * ad1 = * ad2 ; // affectation de la "partie point" de b
     coul = b.coul ; // affectation de la partie propre à pointcol
     return * this ;
   }
  void affiche ()
   { cout << "pointcol : " << x << " " << y << " " << coul << "\n" ;
   }
} ;
main()
{ pointcol p(1, 3, 10) , q(4, 9, 20) ;
 cout << "p       = " ; p.affiche () ;
 cout << "q avant = " ; q.affiche () ;
 q = p ;
 cout << "q apres = " ; q.affiche () ;
}
p       = pointcol : 1 3 10
```

```
q avant = pointcol : 4 9 20

operateur = de pointcol

operateur = de point

q apres = pointcol : 1 3 10
```

Classe "ordinaire" dérivant d'une classe patron :

```
#include <iostream>

using namespace std ;

template <class T> class point

{  T x ; T y ;

  public :

   point (T abs=0, T ord=0)  {  x = abs ; y = ord ; }

   void affiche () {  cout << "Coordonnees : " << x << " " << y << "\n" ; }

} ;

class pointcol_int : public point <int>

{  int coul ;

  public :

   pointcol_int (int abs=0, int ord=0, int cl=1) : point <int> (abs, ord)

    { coul = cl ;

    }

   void affiche ()

    { point<int>::affiche () ; cout << "      couleur : " << coul << "\n" ;

    }

} ;

main ()
```

```
{  point <float> pf (3.5, 2.8) ; pf.affiche () ; // instanciation classe patron
   pointcol_int p (3, 5, 9) ; p.affiche ();    // emploi (classique) de la classe
                                               //      pointcol_int
}
Coordonnees : 3.5 2.8
Coordonnees : 3 5
     couleur : 9
#include <iostream>
using namespace std ;
template <class T> class point
{  T x ; T y ;
 public :
   point (T abs=0, T ord=0)  {  x = abs ; y = ord ; }
   void affiche () {  cout << "Coordonnees : " << x << " " << y << "\n" ; }
} ;
template <class T> class pointcol : public point <T>
{  T coul ;
 public :
   pointcol (T abs=0, T ord=0, T cl=1) : point <T> (abs, ord) { coul = cl ; }
   void affiche () { point<T>::affiche () ; cout << "couleur : " << coul ; }
} ;
main ()
{  point <long> p (34, 45) ; p.affiche () ;
   pointcol <short> q (12, 45, 5) ; q.affiche () ;
```

```
}
```

```
Coordonnees : 34 45
```

```
Coordonnees : 12 45
```

```
couleur : 5
```

L'héritage multiple

Considérons une situation simple, celle où une classe, que nous nommerons *pointcoul*, hérite de deux autres classes nommées *point* et *coul* :

```
class point
{    int x, y ;
 public :
    point (...) {...}
    ~point () {...}
    affiche () {...}
} ;
```

```
class coul
{    short couleur ;
        public :
         coul (...) {...}
         ~coul () {...}
         affiche () {...}
         } ;
```

Nous pouvons définir une classe pointcoul héritant de ces deux classes en la déclarant ainsi (ici, nous avons choisi public pour les deux classes, mais nous pourrions employer private ou protected).

```
class pointcoul : public point, public coul
{ ... } ;
```

Ainsi, la fonction affiche de *pointcoul* sera :

```
void affiche ()
{    point::affiche () ; coul::affiche () ; }
```

Un exemple d'héritage multiple : pointcoul hérite de point et de coul :

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

```
class point
{
 int x, y ;
 public :
  point (int abs, int ord)
    { cout << "++ Constr. point \n" ; x=abs ; y=ord ;
    }
  ~point () { cout << "-- Destr. point \n" ; }
  void affiche ()
   { cout << "Coordonnees : " << x << " " << y << "\n" ;
   }
} ;
class coul
{
  short couleur ;
 public :
  coul (int cl)
    { cout << "++ Constr. coul \n" ; couleur = cl ;
    }
  ~coul () { cout << "-- Destr.  coul \n" ; }
  void affiche ()
    { cout << "Couleur : " << couleur << "\n" ;
    }
} ;
```

```
class pointcoul : public point, public coul
{
 public :
   pointcoul (int, int, int) ;
   ~pointcoul () { cout << "---- Destr. pointcoul \n" ; }
   void affiche ()
    { point::affiche () ; coul::affiche () ;
    }
} ;
pointcoul::pointcoul (int abs, int ord, int cl) : point (abs, ord), coul (cl)
{  cout << "++++ Constr. pointcoul \n" ;
}
main()
{  pointcoul p(3,9,2) ;
  cout << "------------\n" ;
  p.affiche () ;              // appel de affiche de pointcoul
  cout << "------------\n" ;
  p.point::affiche () ;       // on force l'appel de affiche de point
  cout << "------------\n" ;
  p.coul::affiche () ;        // on force l'appel de affiche de coul
  cout << "------------\n" ;
}
++ Constr. point
++ Constr. coul
```

++++ Constr. pointcoul

Coordonnees : 3 9

Couleur : 2

Coordonnees : 3 9

Couleur : 2

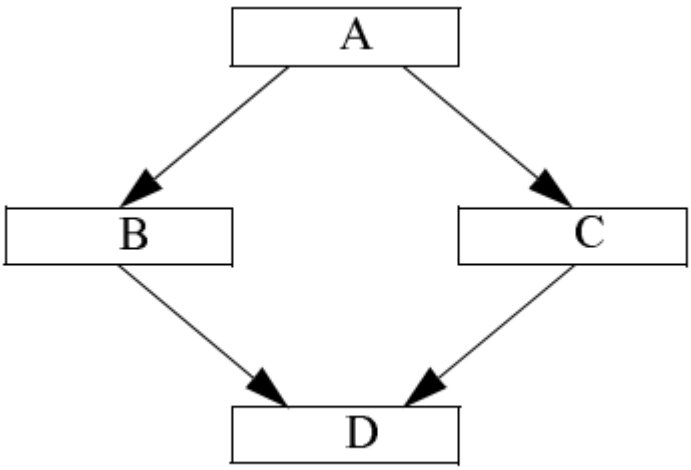
---- Destr. pointcoul

-- Destr. coul

-- Destr. Point

Pour régler les éventuels conflits : les classes virtuelles

Considérons la situation suivante :



correspondant à des déclarations telles que :

class A

```
{     .....

     int x, y ;

} ;

class B : public A {.....} ;

class C : public A {.....} ;

class D : public B, public C

{     .....

} ;
```

En quelque sorte, D hérite deux fois de A ! Dans ces conditions, les membres de A (fonctions ou données) apparaissent deux fois dans D. En ce qui concerne les fonctions membres, cela est manifestement inutile (ce sont les mêmes fonctions), mais sans importance puisqu'elles ne sont pas réellement dupliquées (il n'en existe qu'une pour la classe de base). En revanche, les membres données (x et y) seront effectivement dupliqués dans tous les objets de type D.

Y a-t-il redondance ? En fait, la réponse dépend du problème. Si l'on souhaite que D dispose de deux jeux de données (de A), on ne fera rien de particulier et on se contentera de les distinguer à l'aide de l'opérateur de résolution de portée. Par exemple, on distinguera :

```
A::B::x   de   A::C::x
```

ou, éventuellement, si B et C ne possèdent pas de membre x :

```
B::x      de   C::x
```

En général, cependant, on ne souhaitera pas cette duplication des données. Dans ces conditions, on peut toujours "se débrouiller" pour travailler avec l'un des deux jeux (toujours le même !), mais cela risque d'être fastidieux et dangereux. En fait, vous pouvez demander à C++ de n'incorporer qu'une seule fois les membres de A dans la classe D. Pour cela, il vous faut préciser, dans les déclarations des classes B et C (attention, pas dans celle de D !) que la classe A est "*virtuelle*" (mot clé *virtual*) :

```
class B : public virtual A {.....} ;

class C : public virtual A {.....} ;

class D : public B, public C {.....} ;
```

Notez bien que virtual apparaît ici dans B et C. En effet, définir A comme "virtuelle" dans la déclaration de B signifie que A ne devra être introduite qu'une seule fois dans les descendants éventuels de C. Autrement dit, cette déclaration n'a guère d'effet sur les classes B et C elles-mêmes (si ce n'est une information "cachée" mise en place par le compilateur pour marquer A comme virtuelle au sein de B et C !). Avec ou sans le mot virtual, les classes B et C, se comportent de la même manière tant qu'elles n'ont pas de descendants.

Remarque :

Le mot *virtual* peut être placé indifféremment avant ou après le mot *public* (ou le mot *private*).

En ce qui concerne les transferts d'informations on peut très bien imaginer que B et C n'aient pas prévu les mêmes arguments en ce qui concerne A. Par exemple, on peut avoir :

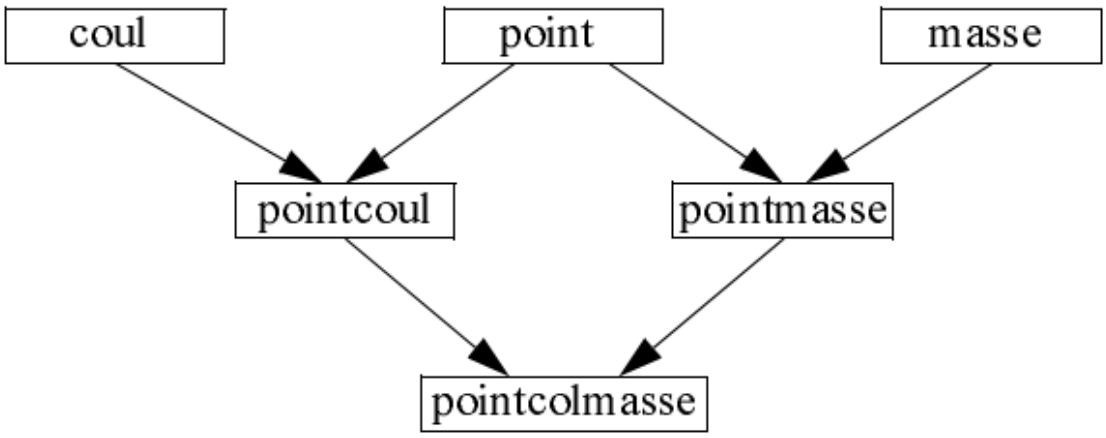
B (int n, int p, double z) : A (n, p)

C (int q, float x) : A (q)

Cela n'a aucune importance puisqu'il y aura en définitive construction de deux objets distincts de type A. Mais si A a été déclarée virtuelle dans B et C, il en va tout autrement. En effet, dans ce cas, on ne construira qu'un seul objet de type A. Quels arguments faut-il transmettre alors au constructeur ? Ceux prévus par B ou ceux prévus par C ? En fait, C++ résout cette ambiguïté de la façon suivante : Le choix des informations à fournir au constructeur de A a lieu non plus dans B ou C, mais dans D. Pour ce faire, C++ vous autorise (uniquement dans ce cas de "dérivation virtuelle") à spécifier, dans le constructeur de D, des informations destinées à A. Ainsi, nous pourrons avoir :

D (int n, int p, double z) : B (n, p, z), A (n, p)

Exemple d'utilisation de l'héritage multiple et de la dérivation virtuelle :



```
#include <iostream>
class point
{ int x, y ;
 public :
  point (int abs, int ord)
    { cout << "++ Constr. point " << abs << " " << ord << "\n" ;
      x=abs ; y=ord ;
    }
  point ()  // constr. par défaut nécessaire pour dérivations virtuelles
    { cout << "++ Constr. defaut point \n" ; x=0 ; y=0 ; }
  void affiche ()
   { cout << "Coordonnees : " << x << " " << y << "\n" ;
   }
} ;
class coul
{  short couleur ;
 public :
  coul (short cl)
    { cout << "++ Constr. coul " << cl <<  "\n" ;
      couleur = cl ;
    }
  void affiche ()
    { cout << "Couleur : " << couleur << "\n" ;
    }
```

```
} ;
class masse
{  int mas ;
 public :
  masse (int m)
    { cout << "++ Constr. masse " << m << "\n" ;
      mas = m ;
    }
  void affiche ()
    { cout << "Masse   : " << mas << "\n" ;
    }
} ;
class pointcoul : public virtual point, public coul
{ public :
   pointcoul (int abs, int ord, int cl) : coul (cl)
     // pas d'info pour point car dérivation virtuelle
   { cout << "++++ Constr. pointcoul " << abs << " " << ord << " "
                                << cl << "\n" ;
   }
   void affiche ()
    { point::affiche () ; coul::affiche () ;
    }
} ;
class pointmasse : public virtual point, public masse
```

```
{ public :
   pointmasse (int abs, int ord, int m) : masse (m)
    // pas d'info pour point car dérivation virtuelle
   { cout << "++++ Constr. pointmasse " << abs << " " << ord << " "
                                  << m << "\n" ;
   }
   void affiche ()
    { point::affiche () ; masse::affiche () ;
    }
} ;
class pointcolmasse : public pointcoul, public pointmasse
{ public :
   pointcolmasse (int abs, int ord, short c, int m) : point (abs, ord),
     pointcoul (abs, ord, c), pointmasse (abs, ord, m)
     // infos abs ord en fait inutiles pour pointcol et pointmasse
   { cout << "++++ Constr. pointcolmasse " << abs + " " << ord << " "
                                  << c << " " << m << "\n" ;
   }
   void affiche ()
    { point::affiche () ; coul::affiche() ; masse::affiche () ;
    }
} ;
main()
{  pointcoul p(3,9,2) ;
```

```
    p.affiche () ;                // appel de affiche de pointcoul
    pointmasse pm(12, 25, 100) ;
    pm.affiche () ;
    pointcolmasse pcm (2, 5, 10, 20) ;
    pcm.affiche () ;
    int n ; cin >> n ;
}
++ Constr. defaut point
++ Constr. coul 2
++++ Constr. pointcoul 3 9 2
Coordonnees : 0 0
Couleur : 2
++ Constr. defaut point
++ Constr. masse 100
++++ Constr. pointmasse 12 25 100
Coordonnees : 0 0
Masse   : 100
++ Constr. point 2 5
++ Constr. coul 10
++++ Constr. pointcoul 2 5 10
++ Constr. masse 20
++++ Constr. pointmasse 2 5 20
++++ Constr. pointcolmasse  5 10 20
Coordonnees : 2 5
```

Couleur : 10

Masse : 20

Les fonctions virtuelles et le polymorphisme

Rappel d'une situation où le typage dynamique est nécessaire :

```
class point                          class pointcol : public point
{    .....                           {    .....
    void affiche () ;                    void affiche () ;
    .....                                .....
} ;                                      } ;
point p ;
pointcol pc ;
point * adp = &p ;
```

L'instruction :

```
adp -> affiche () ;
```

appelle la méthode affiche du type point.

Mais si nous exécutons cette affectation (autorisée) :

```
adp = & pc ;
```

le pointeur adp pointe maintenant sur un objet de type pointcol. Néanmoins, l'instruction :

```
adp -> affiche () ;
```

fait toujours appel à la méthode affiche du type point, alors que le type pointcol dispose lui aussi d'une méthode affiche. En effet, le choix de la méthode à appeler a été réalisé lors de la compilation ; il a donc été fait en fonction du type de la variable adp. C'est la raison pour laquelle on parle de *"ligature statique"*.

Le mécanisme des fonctions virtuelles :

Le mécanisme des fonctions virtuelles proposé par C++ va nous permettre de faire en sorte que l'instruction :

```
adp -> affiche ()
```

appelle non plus systématiquement la méthode affiche de point, mais celle correspondant au type de l'objet réellement désigné par adp (ici point ou pointcol). Pour ce faire, il suffit de déclarer "virtuelle" (mot clé virtual) la méthode affiche de la classe

point :

```
class point
{    .....
     virtual void affiche () ;
     .....
} ;
```

Cette instruction indique au compilateur que les éventuels appels de la fonction affiche doivent utiliser une ligature dynamique et non plus une ligature statique. Autrement dit, lorsque le compilateur rencontrera un appel tel que :

```
adp -> affiche () ;
```

il ne décidera pas de la procédure à appeler. Il se contentera de mettre en place un dispositif permettant de n'effectuer le choix de la fonction qu'au moment de l'exécution de cette instruction, ce choix étant basé sur le type exact de l'objet ayant effectué l'appel (plusieurs exécutions de cette même instruction pouvant appeler des fonctions différentes).

Dans la classe pointcol, on ne procédera à aucune modification : il n'est pas nécessaire de déclarer virtuelle dans les classes dérivées une fonction déclarée virtuelle dans une classe de base (cette information serait redondante).

Mise en œuvre d'une ligature dynamique (ici pour affiche) par la technique des fonctions virtuelles :

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

```
{ protected :          // pour que x et y soient accessibles à pointcol
  int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void affiche ()
    { cout << "Je suis un point \n" ;
      cout << "   mes coordonnees sont : " << x << " " << y << "\n" ;
    }
} ;
class pointcol : public point
{  short couleur ;
 public :
  pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
    { couleur = cl ;
    }
  void affiche ()
    { cout << "Je suis un point colore \n" ;
      cout << "   mes coordonnees sont : " << x << " " << y ;
      cout << "   et ma couleur est :   " << couleur << "\n" ;
    }
} ;
main()
{  point p(3,5) ; point * adp = &p ;
  pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
```

```
  adp->affiche () ; adpc->affiche () ;
  cout << "-----------------\n" ;
  adp = adpc ;          //  adpc = adp serait rejeté
  adp->affiche () ; adpc->affiche () ;
}
Je suis un point
  mes coordonnées sont : 3 5
Je suis un point colore
  mes coordonnées sont : 8 6   et ma couleur est :   2
-----------------
Je suis un point colore
  mes coordonnées sont : 8 6   et ma couleur est :   2
Je suis un point colore
  mes coordonnées sont : 8 6   et ma couleur est :   2
```

Mise en œuvre de ligature dynamique (ici pour identifie) par la technique des fonctions virtuelles :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void identifie ()
    { cout << "Je suis un point \n" ; }
  void affiche ()
```

```
    { identifie () ;
      cout << "Mes coordonnees sont : " << x << " " << y << "\n" ;
    }
} ;
class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs, ord)
    { couleur = cl ; }
  void identifie ()
    { cout << "Je suis un point colore de couleur : " << couleur << "\n" ; }
} ;
main()
{ point p(3,4) ; pointcol pc(5,9,5) ;
  p.affiche () ; pc.affiche () ;      cout << "----------------\n" ;
  point * adp = &p ; pointcol * adpc = &pc ;
  adp->affiche () ; adpc->affiche () ; cout << "----------------\n" ;
  adp = adpc ;
  adp->affiche () ; adpc->affiche () ;
}
```

```
Je suis un point
Mes coordonnees sont : 3 4
Je suis un point colore de couleur : 5
Mes coordonnees sont : 5 9
```

```
---------------

Je suis un point

Mes coordonnees sont : 3 4

Je suis un point colore de couleur : 5

Mes coordonnees sont : 5 9

---------------

Je suis un point colore de couleur : 5

Mes coordonnees sont : 5 9

Je suis un point colore de couleur : 5

Mes coordonnees sont : 5 9
```

La classe string :

Construction :

La classe string dispose de beaucoup de constructeurs ; certains correspondent aux constructeurs d'un vecteur :

```
string ch1 ;        /* construction d'une chaîne vide : ch1.size() == 0  */

string ch2 (10, '*') ;    /* construction d'une chaîne de 10 caractères  */

                    /*    égaux à '*' ; ch2.size() == 10        */

string ch3 (5, '\0') ;    /* construction d'une chaîne de 5 caractères   */

                    /*  de code nul ; ch2.size() == 5           */
```

D'autres permettent d'initialiser une chaîne lors de sa construction, à partir de chaînes usuelles, constantes ou non :

```
string mess1 ("bonjour") ;    /* construction  chaîne de longueur 7 : bonjour */

   // ou string mess1 = "bonjour" ;

char * adr = "salut" ;
```

```
string mess2 (adr) ;        /* construction chaîne de 5 caractères : salut  */
   // ou string mess2 = adr ;
```

Bien entendu, on dispose d'un constructeur par recopie usuel :

```
string s1 ;
  .....
string s2(s1)  /* ou string s2 = s1 ;    construction de s2 par recopie de s1 */
            /*  s2.size() == s1.size()                                  */
```

Opérations globales :

```
string ch ;
getline (cin, ch) ;     // lit une suite de caractères terminée par une fin de ligne
                        // et la range dans l'objet ch (fin de ligne non comprise)
getline (cin, ch, 'x') ; // lit une suite de caractères terminée par le caractère 'x'
                         // et la range dans l'objet ch (caractère 'x' non compris)
```

Concaténation :

L'opérateur + a été surdéfini de manière à permettre la concaténation :

• de deux objets de type string,

• d'un objet de type string avec une chaîne usuelle ou avec un caractère, et ceci dans n'importe quel ordre,

L'opérateur += est défini de façon concomitante. Voici quelques exemples :

```
string ch1 ("bon") ;     /* ch1.length() == 3 */
string ch2 ("jour") ;    /* ch2.length() == 4 */
string ch3 ;             /* ch3.length() == 0 */
ch3 = ch1 + ch2 ;  /* ch3.length() == 7 ; ch3 contient la chaîne "bonjour"  */
ch3 = ch1 + ' ' ;  /* ch3.length() == 4 */
```

```
ch3 += ch2 ;      /* ch3.length() == 8 ; ch3 contient la chaîne "bon jour" */

ch3 += " monsieur" /* ch3 contient la chaîne "bon jour monsieur"        */
```

On notera cependant qu'il n'est pas possible de concaténer deux chaînes usuelles ou une chaîne usuelle et un caractère :

```
char c1, c2 ;

ch3 = ch1 + c1 + ch2 + c2 ; /* correct */

ch3 = ch1 + c1 + c2 ;      /* incorrect ; mais on peut toujours faire : */

                    /*    ch3 = ch1 + c1 ; ch3 += c2 ;        */
```

Recherche d'une chaîne ou d'un caractère :

La fonction membre find permet de rechercher, dans une chaîne donnée, la première occurrence :

• d'une autre chaîne (on parle souvent de sous-chaîne) fournie soit par un objet de type string, soit par une chaîne usuelle,

• d'un caractère donné.

Par défaut, la recherche commence au début de la chaîne, mais on peut la faire débuter à un caractère de rang donné. Voici quelques exemples :

```
string ch = "anticonstitutionnellement" ;

string mot ("on");

char * ad = "ti" ;

int i ;

i = ch.find ("elle")  ;   /* i == 17              */

i = ch.find ("elles") ;   /* i <0 ou i > ch.length() */

i = ch.find (mot) ;       /* i == 5               */

i = ch.find (ad) ;        /* i == 2               */

i = ch.find ('n') ;       /* i == 1               */

i = ch.find ('n', 5)   /* i == 6  , car ici, la recherche débute à ch[5] */

i = ch.find ('p') ;       /* i <0 ou i > ch.length() */
```

De manière semblable, la fonction rfind permet de rechercher la dernière occurrence d'une autre chaîne ou d'un caractère.

```
string ch = "anticonstitutionnellement" ;

string mot ("on");

char * ad = "ti" ;

int i ;

i = ch.rfind ("elle")  ;   /* i == 17 */

i = ch.rfind ("elles") ;   /* i <0 ou i > ch.length() */

i = ch.rfind (mot) ;       /* i == 14 */

i = ch.rfind (ad) ;        /* i == 12 */

i = ch.rfind ('n') ;       /* i == 23 */

i = ch.rfind ('n', 18) ;   /* i == 16 */
```

Insertions :

```
#include <iostream>

#include <string>

#include <list>

using namespace std ;

main()

{

 string ch ("0123456") ;

 string voy ("aeiou") ;

 char t[] = {"778899"} ;

       /* insere le caractere a en ch.begin()+1 */

 ch.insert (ch.begin()+1, 'a') ;   cout << ch << "\n" ;
```

```
        /* insere le caractere b en position d'indice 4   */
 ch.insert (4, 1, 'b') ;          cout << ch << "\n" ;
        /* insere 3 fois le caractere x en fin de ch  */
 ch.insert (ch.end(), 3, 'x') ;    cout << ch << "\n" ;
        /* insere 3 fois le caractere x en position d'indice 6 */
 ch.insert (6, 3, 'x') ;          cout << ch << "\n" ;
        /* insere la chaine voy en position 0 */
 ch.insert (0, voy) ;             cout << ch << "\n" ;
   /* insere en debut, la chaine voy, a partir de position 1, longueur 3 */
 ch.insert (0, voy, 1, 3) ;       cout << ch << "\n" ;
        /* insertion d'une sequence */
 ch.insert (ch.begin()+2, t, t+6) ; cout << ch << "\n" ;
}
```

```
0a123456
0a12b3456
0a12b3456xxx
0a12b3xxx456xxx
aeiou0a12b3xxx456xxx
eioaeiou0a12b3xxx456xxx
ei778899oaeiou0a12b3xxx456xxx
```

Suppressions :

```
#include <iostream>
#include <string>
#include <list>
```

```
using namespace std ;

main()

{ string ch ("0123456789"), ch_bis=ch ;

      /* supprime, a partir de position d'indice 3, pour une longueur de 2 */

 ch.erase (3, 2) ;                    cout << "A : " << ch << "\n" ;

 ch = ch_bis ;

     /* supprime, de begin()+3 à begin()+6 */

 ch.erase (ch.begin()+3, ch.begin()+6) ;  cout << "B : " << ch << "\n" ;

     /* supprime, a partir de position d'indice 3   */

 ch.erase (3) ;                       cout << "C : " << ch << "\n" ;

 ch = ch_bis ;

     /* supprime le caractere de position begin()+4 */

 ch.erase (ch.begin()+4) ;              cout << "D : "<< ch << "\n" ;

}
```

```
A : 01256789

B : 0126789

C : 012

D : 012356789
```

Remplacements :

```
#include <iostream>

#include <string>

using namespace std ;

main()

{ string ch ("0123456") ;
```

```
 string voy ("aeiou") ;
 char t[] = {"+*-/=<>"} ;
 char * message = "hello" ;
   /* remplace, a partir de indice 2, sur longueur 3, par voy */
 ch.replace (2, 3, voy) ;                    cout << ch << "\n" ;
   /* remplace, a partir de indice 0 sur longueur 1, par voy,  */
   /*  a partir de indice 2, longueur 3                        */
 ch.replace (0, 1, voy, 1, 2) ;              cout << ch << "\n" ;
   /* remplace, a partir de indice 1 sur longueur 2, par 8 fois '*' */
 ch.replace (1, 2, 8, '*') ;                 cout << ch << "\n" ;
   /* remplace, a partir de indice 1 sur longueur 2, par 5 fois '#' */
 ch.replace (1, 2, 5, '#') ;                 cout << ch << "\n" ;
   /* remplace, a partir de indice 2, sur longueur 4, par "xxxxxx" */
 ch.replace (2, 4, "xxxxxx" ) ;              cout << ch << "\n" ;
   /* remplace les 7 derniers caracteres par les 3 premiers de message */
 ch.replace (ch.length()-7, ch.length(), message, 3) ;   cout << ch << "\n" ;
   /* remplace tous les caracteres, sauf le dernier, par (t, t+5) */
 ch.replace (ch.begin(), ch.begin()+ch.length()-1, t, t+5) ; cout << ch << "\n" ;
}
01aeiou56
ei1aeiou56
e********aeiou56
e#####******aeiou56
e#xxxxxx******aeiou56
```

e#xxxxxx******hel

+*-/=l

La gestion des exceptions

La version 3 de C++ a intégré dans le langage un mécanisme très puissant de traitement des anomalies, nommé *gestion des exceptions*. Il a le mérite de découpler totalement la détection d'une anomalie (exception) de son traitement. D'une manière générale, une exception est une rupture de séquence déclenchée (on dit aussi "levée" ou "lancée") par une instruction throw comportant une expression d'un type donné. Il y a alors branchement à un ensemble d'instructions nommé gestionnaire d'exception (on parle aussi de "capture par un gestionnaire"), dont le nom est déterminé par la nature de l'exception. Plus précisément, chaque exception est caractérisée par un type, et le choix du bon gestionnaire se fait en fonction de la nature de l'expression mentionnée à throw. Pour ce faire, il est nécessaire de respecter deux conditions :

• inclure dans un bloc particulier, dit "bloc try", toutes les instructions dans lesquelles on souhaite pouvoir détecter une exception ; un tel bloc se présente ainsi :

```
try
{
  // instructions
}
```

• faire suivre ce bloc de la définition des différents "gestionnaires d'exceptions" nécessaires (ici, un seul suffit). Chaque définition est précédée d'un en-tête introduit par le mot clé catch (comme si catch était le nom d'une fonction gestionnaire...)

Premier exemple de gestion d'exception :

```
#include <iostream>
#include <cstdlib>      /* ancien <stdlib.h> : pour exit */
using namespace std ;
 /* déclaration de la classe vect */
```

```
class vect
{ int nelem ;
  int * adr ;
 public :
  vect (int) ;
  ~vect () ;
  int & operator [] (int) ;
} ;
 /* déclaration et définition d'une classe vect_limite (vide pour l'instant) */
class vect_limite
{ } ;
 /* définition de la classe vect */
vect::vect (int n)
{ adr = new int [nelem = n] ; }
vect::~vect ()
{ delete adr ; }
int & vect::operator [] (int i)
{ if (i<0 || i>nelem)
   { vect_limite l ; throw (l) ;
   }
  return adr [i] ;
}
/* test interception exception vect_limite */
main ()
{ try
  { vect v(10) ;
    v[11] = 5 ;    /* indice trop grand */
```

```
 }
 catch (vect_limite l)  /* nom d'argument superflu ici */
 { cout << "exception limite \n" ;
   exit (-1) ;
 }
}
```

exception limite

Exemple de gestion de deux exceptions :

Examinons maintenant un exemple un peu plus réaliste dans lequel on trouve deux exceptions différentes et où il y a transmission d'informations aux gestionnaires. Nous allons reprendre la classe vect précédente, en lui permettant de lancer deux sortes d'exceptions :

• une exception de type vect_limite comme précédemment mais, cette fois, on prévoit de transmettre au gestionnaire la valeur de l'indice qui a déclenché l'exception ;

• une exception vect_creation déclenchée lorsque l'on transmet au constructeur un nombre d'éléments incorrect (négatif ou nul) ; là encore, on prévoit de transmettre ce nombre au gestionnaire.

```
#include <iostream>
#include <cstdlib>    // ancien <stdlib.h>   pour exit
using namespace std ;
 /* déclaration de la classe vect */
class vect
{ int nelem ;
  int * adr ;
 public :
  vect (int) ;
  ~vect () ;
  int & operator [] (int) ;
} ;
 /* déclaration - définition des deux classes exception */
```

```
class vect_limite
{ public :
  int hors ;          // valeur indice hors limites (public)
  vect_limite (int i)   // constructeur
   { hors = i ; }
} ;
class vect_creation
{ public :
  int nb ;            // nombre elements demandes (public)
  vect_creation (int i)  // constructeur
   { nb = i ; }
} ;
 /* définition de la classe vect */
vect::vect (int n)
{ if (n <= 0)
   { vect_creation c(n) ;    // anomalie
     throw c ;
   }
  adr = new int [nelem = n] ; // construction normale
}
vect::~vect ()
{ delete adr ;
}
int & vect::operator [] (int i)
{ if (i<0 || i>nelem)
    { vect_limite l(i) ;    // anomalie
      throw l ;
```

```
    }
 return adr [i] ;          // fonctionnement normal
}
/* test exception */
main ()
{
 try
 { vect v(-3) ;     // provoque l'exception vect_creation
   v[11] = 5 ;      // provoquerait l'exception vect_limite
 }
 catch (vect_limite l)
 { cout << "exception indice " << l.hors << " hors limites \n" ;
   exit (-1) ;
 }
 catch (vect_creation c)
 { cout << "exception creation vect nb elem = " << c.nb << "\n" ;
   exit (-1) ;
 }
}
exception creation vect nb elem = -3
```

Poursuite de l'exécution du programme :

Le gestionnaire d'exception peut très bien ne pas comporter d'instruction d'arrêt de l'exécution (exit, abort). Dans ce cas, après l'exécution des intructions du gestionnaire concerné, on passe tout simplement à la suite du bloc try concerné. Cela revient à dire qu'on passe à la première instruction suivant le dernier gestionnaire.

Lorsqu'on "passe à travers" un gestionnaire d'exception :

```
// déclaration et définition des classes vect, vect_limite, vect_creation
//    comme dans le paragraphe 2
```

```
//    .....
main()
{ void f(int) ;
  cout << "avant appel de f(3) \n" ;
  f(3) ;
  cout << "avant appel de f(8) \n" ;
  f(8) ;
  cout << "apres appel de f(8) \n" ;
}
void f(int n)
{ try
  { cout << "debut bloc try\n" ;
    vect v(5) ;
    v[n] = 0 ;     // OK pour n=3 ; déclenche une exception pour n=8
    cout << "fin bloc try\n" ;
  }
  catch (vect_limite l)
  { cout << "exception indice " << l.hors << " hors limites \n" ;
  }
  catch (vect_creation c)
  { cout << "exception creation\n" ;
  }
  // après le bloc try
  cout << "dans f apres bloc try - valeur de n = " << n << "\n" ;
}
avant appel de f(3)
debut bloc try
```

```
fin bloc try

dans f apres bloc try - valeur de n = 3

avant appel de f(8)

debut bloc try

exception indice 8 hors limites

dans f apres bloc try - valeur de n = 8

apres appel de f(8)
```

CONCLUSION :

Ce résumé du langage C++ vous permettra de maitriser l'essentiel des principes du langage pour bien mené la programmation orienté objet. Toutefois, pour plus de détails, nous vous renvoyons à l'ouvrage de **Claude Delannoy** « *C++ pour les programmeurs C »* ainsi qu'aux tutoriels sur internet.

Abdourahmane FALL

fallprofessionnel@hotmail.fr

+221 77 274 46 71