Le langage C++

Henri Garreta

Table des matières

31 mai 2006

`henri.garreta@luminy.univ-mrs.fr`

1 Eléments préalables

Ce document est le support du cours sur le langage C++, considéré comme une extension de C (tel que normalisé par l'ISO en 1990), langage présumé bien connu.

Attention, la présentation faite ici est déséquilibrée : des concepts importants ne sont pas expliqués, pour la raison qu'ils sont réalisés en C++ comme en C, donc supposés acquis. En revanche, nous insistons sur les différences entre C et C++ et notamment sur tous les éléments « orientés objets » que C++ ajoute à C.

Cette première section expose un certain nombre de notions qui, sans être directement liés à la méthodologie objets, font déjà apparaître C++ comme une amélioration notable de C.

1.1 Placement des déclarations de variables

En C les déclarations de variables doivent apparaître au début d'un bloc. En C++, au contraire, on peut mettre une déclaration de variable partout où on peut mettre une instruction[1].

Cette différence paraît mineure, mais elle est importante par son esprit. Elle permet de repousser la déclaration d'une variable jusqu'à l'endroit du programme où l'on dispose d'assez éléments pour l'initialiser. On lutte aussi contre les variables « déjà déclarées mais non encore initialisées » qui sont un important vivier de bugs dans les programmes.

Exemple, l'emploi d'un fichier. Version C :

```
{
    FILE *fic;
    ...
    obtention de nomFic, le nom du fichier à ouvrir
    Danger ! Tout emploi de fic ici est erroné
    ...
    fic = fopen(nom, "w");
    ...
    ici, l'emploi de fic est légitime
    ...
}
```

Version C++ :

```
{
    ...
    obtention de nomFic, le nom du fichier à ouvrir
    ici pas de danger d'utiliser fic à tort
    ...
    FILE *fic = fopen(nom, "w");
    ...
    ici, l'emploi de fic est légitime
    ...
}
```

1.2 Booléens

En plus des types définis par l'utilisateur (ou classes, une des notions fondamentales de la programmation orientée objets) C++ possède quelques types qui manquaient à C, notamment le type booléen et les types références.

Le type `bool` (pour booléen) comporte deux valeurs : `false` et `true`. Contrairement à C :

- le résultat d'une opération logique (`&&`, `||`, etc.) est de type booléen,
- là où une condition est attendue on doit mettre une expression de type booléen et
- il est déconseillé de prendre les entiers pour des booléens et réciproquement.

Conséquence pratique : n'écrivez pas « `if (x) ...` » au lieu de « `if (x != 0) ...` »

1.3 Références

1.3.1 Notion

A côté des pointeurs, les références sont une autre manière de manipuler les adresses des objets placés dans la mémoire. Une référence est un pointeur géré de manière interne par la machine. Si T est un type donné, le

[1] Bien entendu, une règle absolue reste en vigueur : une variable ne peut être utilisée qu'après qu'elle ait été déclarée.

type « référence sur T » se note T`&`. Exemple :

```
int i;
int & r = i;            // r est une référence sur i
```

Une valeur de type référence est une adresse mais, hormis lors de son initialisation, toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse. Il en découle qu'il est obligatoire d'initialiser une référence lors de sa création ; après, c'est trop tard :

```
r = j;                  // ceci ne transforme pas r en une référence
                        // sur j mais copie la valeur de j dans i
```

1.3.2 Références paramètres des fonctions

L'utilité principale des références est de permettre de donner aux fonctions des paramètres modifiables, sans utiliser explicitement les pointeurs. Exemple :

```
void permuter(int & a, int & b) {
    int w = a;
    a = b; b = w;
    }
```

Lors d'un appel de cette fonction, comme

```
permuter(t[i], u);
```

ses paramètres formels `a` et `b` sont initialisés avec les adresses des paramètres effectifs `t[i]` et `u`, mais cette utilisation des adresses reste cachée, le programmeur n'a pas à s'en occuper. Autrement dit, à l'occasion d'un tel appel, `a` et `b` ne sont pas des variables locales de la fonction recevant des copies des valeurs des paramètres, mais d'authentiques synonymes des variables `t[i]` et `u`. Il en résulte que l'appel ci-dessus permute effectivement les valeurs des variables `t[i]` et `u`[2].

1.3.3 Fonctions renvoyant une référence

Il est possible d'écrire des fonctions qui renvoient une référence comme résultat. Cela leur permet d'être le membre gauche d'une affectation, ce qui donne lieu à des expressions élégantes et efficaces.

Par exemple, voici comment une fonction permet de simuler un tableau indexé par des chaînes de caractères[3] :

```
char *noms[N];
int ages[N];

int & age(char *nom) {
    for (int i = 0; i < N; i++)
        if (strcmp(nom, noms[i]) == 0)
            return ages[i];
}
```

Une telle fonction permet l'écriture d'expressions qui ressemblent à des accès à un tableau dont les indices seraient des chaînes :

```
age("Amélie") = 20;
```

ou encore

```
age("Benjamin")++;
```

1.3.4 Références sur des données constantes

Lors de l'écriture d'une fonction il est parfois souhaitable d'associer l'efficacité des arguments références (puisque dans le cas d'une référence il n'y a pas recopie de la valeur de l'argument) et la sécurité des arguments par valeur (qui ne peuvent en aucun cas être modifiés par la fonction).

Cela peut se faire en déclarant des arguments comme des références sur des objets immodifiables, ou constants, à l'aide du qualifieur `const` :

[2] Notez qu'une telle chose est impossible en C, où une fonction appelée par l'expression `permuter(t[i], u)` ne peut recevoir que des *copies* des valeurs de de `t[i]` et `u`.

[3] C'est un exemple simpliste, pour faire court nous n'y avons pas traité le cas de l'absence de la chaîne cherchée.

```
void uneFonction(const unType & arg) {
    ...
    ici arg n'est pas une recopie de l'argument effectif (puisque référence)
    mais il ne peut pas être modifié par la fonction (puisque const)
    ...
}
```

1.3.5 Références ou pointeurs ?

Il existe quelques situations, nous les verrons plus loin, où les références se révèlent indispensables. Cependant, la plupart du temps elles font double emploi avec les pointeurs et il n'existe pas de critères simples pour choisir des références plutôt que des pointeurs ou le contraire.

Par exemple, la fonction `permuter` montrée à la page précédente peut s'écrire aussi :

```
void permuter(int *a, int *b) {
    int w = *a;
    *a = *b; *b = w;
    }
```

son appel s'écrit alors

```
permuter(&t[i], &u);
```

Attention On notera que l'opérateur `&` (à un argument) a une signification très différente selon le contexte dans lequel il apparaît :

- employé dans une déclaration, comme dans
  ```
  int &r = x;
  ```
  il sert à indiquer un type référence : « `r` est une référence sur un `int` »
- employé ailleurs que dans une déclaration il indique l'opération « obtention de l'adresse », comme dans l'expression suivante qui signifie « affecter l'adresse de `x` à `p` » :
  ```
  p = &x;
  ```
- enfin, on doit se souvenir qu'il y a en C++, comme en C, un opérateur `&` binaire (à deux arguments) qui exprime la conjonction bit-à-bit de deux mots-machine.

1.4 Fonctions en ligne

Normalement, un appel de fonction est une *rupture de séquence* : à l'endroit où un appel figure, la machine cesse d'exécuter séquentiellement les instructions en cours ; les arguments de l'appel sont disposés sur la pile d'exécution, et l'exécution continue ailleurs, là où se trouve le code de la fonction.

Une fonction en ligne est le contraire de cela : là où l'appel d'une telle fonction apparaît il n'y a pas de rupture de séquence. Au lieu de cela, le compilateur remplace l'appel de la fonction par le corps de celle-ci, en mettant les arguments affectifs à la place des arguments formels.

Cela se fait dans un but d'efficacité : puisqu'il n'y a pas d'appel, pas de préparation des arguments, pas de rupture de séquence, pas de retour, etc. Mais il est clair qu'un tel traitement ne peut convenir qu'à des fonctions fréquemment appelées (si une fonction ne sert pas souvent, à quoi bon la rendre plus rapide ?) de petite taille (sinon le code compilé risque de devenir démesurément volumineux) et rapides (si une fonction effectue une opération lente, le gain de temps obtenu en supprimant l'appel est négligeable).

En C++ on indique qu'une fonction doit être traitée en ligne en faisant précéder sa déclaration par le mot `inline` :

```
inline int abs(int x) {
    return x >= 0 ? x : -x;
    }
```

Les fonctions en ligne de C++ rendent le même service que les macros (avec arguments) de C, mais on notera que les fonctions en ligne sont plus fiables car, contrairement à ce qui se passe pour les macros, le compilateur peut y effectuer tous les contrôles syntaxiques et sémantiques qu'il fait sur les fonctions ordinaires.

La portée d'une fonction en ligne est réduite au fichier où la fonction est définie. Par conséquent, de telles fonctions sont généralement écrites dans des fichiers en-tête (fichiers « `.h` »), qui doivent être inclus dans tous les fichiers comportant des appels de ces fonctions.

1.5 Valeurs par défaut des arguments des fonctions

Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut. Exemple :

```
void trier(void *table, int nbr, int taille = sizeof(void *), bool croissant = true);
```

Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants peuvent alors être omis (ainsi que les virgules correspondantes), les paramètres formels seront initialisés avec les valeurs par défaut. Exemples :

```
trier(t, n, sizeof(int), false);
trier(t, n, sizeof(int));        // croissant = true
trier(t, n);                     // taille = sizeof(void *), croissant = true
```

1.6 Surcharge des noms de fonctions

La *signature d'une fonction* est la suite des types de ses arguments formels (et quelques éléments supplémentaires, que nous verrons plus loin). Le type du résultat rendu par la fonction ne fait pas partie de sa signature.

La *surcharge des noms des fonctions* consiste en ceci : en C++ des fonctions différentes peuvent avoir le même nom, à la condition que leurs signatures soient assez différentes pour que, lors de chaque appel, le nombre et les types des arguments effectifs permettent de choisir sans ambiguïté la fonction à appeler. Exemple :

```
int puissance(int x, int n) {
    calcul de x^n avec x et n entiers
}
double puissance(double x, int n) {
    calcul de x^n avec x flottant et n entier
}
double puissance(double x, double y) {
    calcul de x^y avec x et y flottants
}
```

On voit sur cet exemple l'intérêt de la surcharge des noms des fonctions : la même notion abstraite « x à la puissance n » se traduit par des algorithmes très différents selon que n est entier ($x^n = x.x...x$) ou réel ($x^n = e^{n \log x}$) ; de plus, pour n entier, si on veut que x^n soit entier lorsque x est entier, on doit écrire deux fonctions distinctes, une pour x entier et une pour x réel.

Or le programmeur n'aura qu'un nom à connaître, `puissance`. Il écrira dans tous les cas

```
c = puissance(a, b);
```

le compilateur se chargeant de choisir la fonction la plus adaptée, selon les types de `a` et `b`.

NOTES. 1. Le type du résultat rendu par la fonction ne fait pas partie de la signature. Par conséquent, on ne peut pas donner le même nom à deux fonctions qui ne différent que par le type du résultat qu'elles rendent.

2. Lors de l'appel d'une fonction surchargée, la fonction effectivement appelée est celle dont la signature correspond avec les types des arguments effectifs de l'appel. S'il y a une correspondance exacte, pas de problème. S'il n'y a pas de correspondance exacte, alors des règles complexes (trop complexes pur les expliquer ici) s'appliquent, pour déterminer la fonction à appeler. Malgré ces règles, il existe de nombreux cas de figure ambigus, que le compilateur ne peut pas résoudre.

Par exemple, si `x` est flottant et `n` entier, l'appel `puissance(n, x)` est erroné, alors qu'il aurait été correct s'il y avait eu une seule définition de la fonction `puissance` (les conversions entier $\leftrightarrow$ flottant nécessaires auraient alors été faites).

1.7 Appel et définition de fonctions écrites en C

Pour que la compilation et l'édition de liens d'un programme avec des fonctions surchargées soient possibles, le compilateur C++ doit fabriquer pour chaque fonction un *nom long* comprenant le nom de la fonction et une représentation codée de sa signature ; c'est ce nom long qui est communiqué à l'éditeur de liens.

Or, les fonctions produites par un compilateur C n'ont pas de tels noms longs ; si on ne prend pas de disposition particulière, il sera donc impossible d'appeler dans un programme C++ une fonction écrite en C, ou réciproquement. On remédie à cette impossibilité par l'utilisation de la déclaration « extern "C" » :

```
extern "C" {
    déclarations et définitions d'éléments
    dont le nom est représenté à la manière de C
}
```

1.8 Entrées-sorties simples

Cette section traite de l'utilisation simple des flux standard d'entrée-sortie, c'est-à-dire la manière de faire en C++ les opérations qu'on fait habituellement en C avec les fonctions `printf` et `scanf`.

Un programme qui utilise les flux standard d'entrée-sortie doit comporter la directive

```
#include <iostream.h>
```

ou bien, si vous utilisez un compilateur récent et que vous suivez de près les recommandations de la norme[4] :

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

Les flux d'entrée-sortie sont représentés dans les programmes par les trois variables, prédéclarées et préinitialisées, suivantes :

- `cin`, le flux standard d'entrée (l'équivalent du `stdin` de C), qui est habituellement associé au clavier du poste de travail,
- `cout`, le flux standard de sortie (l'équivalent du `stdout` de C), qui est habituellement associé à l'écran du poste de travail,
- `cerr`, le flux standard pour la sortie des messages d'erreur (l'équivalent du `stderr` de C), également associé à l'écran du poste de travail.

Les écritures et lectures sur ces unités ne se font pas en appelant des fonctions, mais à l'aide des opérateurs `<<`, appelé *opérateur d'injection* (« injection » de données dans un flux de sortie), et `>>`, appelé *opérateur d'extraction* (« extraction » de données d'un flux d'entrée). Or, le mécanisme de la surcharge des opérateurs (voir la section 3) permet la détection des types des données à lire ou à écrire. Ainsi, le programmeur n'a pas à s'encombrer avec des spécifications de format.

La syntaxe d'un injection de donnée sur la sortie standard `cout` est :

`cout <<` *expression à écrire*

le résultat de cette expression est l'objet `cout` lui-même. On peut donc lui injecter une autre donnée, puis encore une, etc. :

`((cout <<` *expression* `) <<` *expression* `) <<` *expression*

ce qui, l'opérateur `<<` étant associatif à gauche, se note aussi, de manière bien plus agréable :

`cout <<` *expression* `<<` *expression* `<<` *expression*

Le même procédé existe avec l'extraction depuis `cin`. Par exemple, le programme suivant est un programme C++ complet. Il calcule x^n (pour x flottant et n entier).

```
#include <iostream.h>

double puissance(double x, int n) {
     algorithme de calcul de x^n
}

void main() {
    double x;
    int n;
    cout << "Donne x et n : ";
    cin >> x >> n;
    cout << x << "^" << n << " = " << puissance(x, n) << "\n";
}
```

Exemple d'exécution de ce programme :

```
Donne x et n : 2 10
2^10 = 1024
```

[4] C'est-à-dire, si vous utilisez les espaces de noms, ou *namespace* (tous les éléments de la bibliothèque standard sont dans l'espace de noms `std`).

1.9 Allocation dynamique de mémoire

Des différences entre C et C++ existent aussi au niveau de l'allocation et de la restitution dynamique de mémoire.

Les fonctions `malloc` et `free` de la bibliothèque standard C sont disponibles en C++. Mais il est fortement conseillé de leur préférer les opérateurs `new` et `delete`. La raison principale est la suivante : les objets créés à l'aide de `new` sont initialisés à l'aide des constructeurs (cf. section 2.4) correspondants, ce que ne fait pas `malloc`. De même, les objets liberés en utilisant `delete` sont finalisés en utilisant le destructeur (cf. section 2.6) de la classe correspondante, contrairement à ce que fait `free`.

– Pour allouer un unique objet :

`new` *type*

– Pour allouer un tableau de n objets :

`new` *type*`[`n`]`

Dans les deux cas, `new` renvoie une valeur de type pointeur sur un *type*, c'est-à-dire « *type* `*` ». Exemples (on suppose que `Machin` est un type défini par ailleurs) :

```
Machin *ptr = new Machin;          // un objet Machin
int *tab = new int[n];             // un tableau de n int
```

Si *type* est une classe possédant un constructeur par défaut, celui-ci sera appelé une fois (cas de l'allocation d'un objet simple) ou n fois (allocation d'un tableau d'objets) pour construire l'objet ou les objets alloués. Si *type* est une classe sans constructeur par défaut, une erreur sera signalée par le compilateur.

Pour un tableau, la dimension n peut être donnée par une variable, c'est-à-dire être inconnue lors de la compilation, mais la taille des composantes doit être connue. Il en découle que dans le cas d'un tableau à plusieurs indices, seule la première dimension peut être non constante :

```
double (*M)[10];              // pointeur de tableaux de 10 double
...
acquisition de la valeur de n
...
M = new double[n][10];        // allocation d'un tableau de n tableaux de 10 double
```

`M` pourra ensuite être utilisé comme une matrice à `n` lignes et 10 colonnes.

L'opérateur `delete` restitue la mémoire dynamique. Si la valeur de `p` a été obtenue par un appel de `new`, on écrit

```
delete p;
```

dans le cas d'un objet qui n'est pas un tableau, et

```
delete [] p;
```

si ce que `p` pointe est un tableau. Les conséquences d'une utilisation de `delete` là où il aurait fallu utiliser `delete[]`, ou inversement, sont imprévisibles.

2 Classes

2.1 Classes et objets

Un objet est constitué par l'association d'une certaine quantité de mémoire, organisée en champs, et d'un ensemble de fonctions, destinées principalement à la consultation et la modification des valeurs de ces champs.

La définition d'un type objet s'appelle une classe. D'un point de vue syntaxique, cela ressemble beaucoup à une définition de structure, sauf que

– le mot réservé `class` remplace[5] le mot `struct`,

– certains champs de la classe sont des fonctions.

Par exemple, le programme suivant est une première version d'une classe `Point` destinée à représenter les points affichés dans une fenêtre graphique :

[5] En fait on peut aussi utiliser `struct`, voyez la section 2.2.5.

```
class Point {
public:
    void afficher() {
        cout << '(' << x << ',' << y << ')';
    }
    void placer(int a, int b) {
        validation des valeurs de a et b;
        x = a; y = b;
    }
private:
    int x, y;
};
```

Chaque objet de la classe `Point` comporte un peu de mémoire, composée de deux entiers `x` et `y`, et de deux fonctions : `afficher`, qui accède à `x` et `y` sans les modifier, et `placer`, qui change les valeurs de `x` et `y`.

L'association de membres et fonctions au sein d'une classe, avec la possibilité de rendre privés certains d'entre eux, s'appelle l'*encapsulation* des données. Intérêt de la démarche : puisqu'elles ont été déclarées privées, les coordonnées `x` et `y` d'un point ne peuvent être modifiées autrement que par un appel de la fonction `placer` sur ce point. Or, en prenant les précautions nécessaires lors de l'écriture de cette fonction (ce que nous avons noté « *validation des valeurs de* `a` *et* `b` ») le programmeur responsable de la classe `Point` peut garantir aux utilisateurs de cette classe que tous les objets crées auront toujours des coordonnées correctes. Autrement dit : *chaque objet peut prendre soin de sa propre cohérence interne.*

Autre avantage important : on pourra à tout moment changer l'implémentation (i.e. les détails internes) de la classe tout en ayant la certitude qu'il n'y aura rien à changer dans les programmes qui l'utilisent.

NOTE. Dans une classe, les déclarations des membres peuvent se trouver dans un ordre quelconque, même lorsque ces membres se référencent mutuellement. Dans l'exemple précédent, le membre `afficher` mentionne les membres `x` et `y`, dont la définition se trouve après celle de `afficher`.

JARGON. On appelle
- *objet* une donnée d'un type classe ou structure,
- *fonction membre* un membre d'une classe qui est une fonction[6],
- *donnée membre* un membre qui est une variable[7].

2.2 Accès aux membres

2.2.1 Accès aux membres d'un objet

On accède aux membres des objets en C++ comme on accède aux membres des structures en C. Par exemple, à la suite de la définition de la classe `Point` donnée précédemment on peut déclarer des variables de cette classe en écrivant[8] :

```
Point a, b, *pt;        // deux points et un pointeur de point
```

Dans un contexte où le droit de faire un tel accès est acquis (cf. section 2.2.3) l'accès aux membres du point `a` s'écrit :

```
a.x = 0;                // un accès bien écrit au membre x du point a
d = a.distance(b);      // un appel bien écrit de la fonction distance de l'objet a
```

Si on suppose que le pointeur `pt` a été initialisé, par exemple par une expression telle que

```
pt = new Point;         // allocation dynamique d'un point
```

alors des accès analogues aux précédents s'écrivent :

```
pt->x = 0;              // un accès bien écrit au membre x du point pointé par pt
d = pt->distance(b);    // un appel de la fonction distance de l'objet pointé par pt
```

A propos de l'accès à un membre d'un objet, deux questions se posent. Il faut comprendre qu'elles sont tout à fait indépendantes l'une de l'autre :
- l'accès est-il bien écrit ? c'est-à-dire, désigne-t-il bien le membre voulu de l'objet voulu ?
- cet accès est-il légitime ? c'est-à-dire, dans le contexte où il est écrit, a-t-on le droit d'accès sur le membre en question ? La question des droits d'accès est traitée à la section 2.2.3.

[6] Dans la plupart des langages orientés objets, les fonctions membres sont appelées *méthodes*.

[7] Dans beaucoup de langages orientés objets, les données membres sont appelées *variables d'instance* et aussi, sous certaines conditions, *propriétés*

[8] Notez que, une fois la classe déclarée, il n'est pas obligatoire d'écrire `class` devant `Point` pour y faire référence.

2.2.2 Accès à ses propres membres, accès à soi-même

Quand des membres d'un objet apparaissent dans une expression écrite dans une fonction *du même objet* on dit que ce dernier fait un *accès à ses propres membres.*

On a droit dans ce cas à une notation simplifiée : on écrit le membre tout seul, sans expliciter l'objet en question. C'est ce que nous avons fait dans les fonctions de la classe `Point` :

```
class Point {
    ...
    void afficher() {
        cout << '(' << x << ',' << y << ')';
    }
    ...
};
```

Dans la fonction `afficher`, les membres `x` et `y` dont il question sont ceux de l'objet à travers lequel on aura appelé cette fonction. Autrement dit, lors d'un appel comme

```
unPoint.afficher();
```

le corps de cette fonction sera équivalent à

```
cout << '(' << unPoint.x << ',' << unPoint.y << ')';
```

Accès à soi-même. Il arrive que dans une fonction membre d'un objet on doive faire référence à l'objet (tout entier) à travers lequel on a appelé la fonction. Il faut savoir que dans une fonction membre[9] on dispose de la pseudo variable `this` qui représente un pointeur vers l'objet en question.

Par exemple, la fonction afficher peut s'écrire de manière équivalente, *mais cela n'a aucun intérêt* :

```
void afficher() {
    cout << '(' << this->x << ',' << this->y << ')';
}
```

Pour voir un exemple plus utile d'utilisation de `this` imaginons qu'on nous demande d'ajouter à la classe `Point` deux fonctions booléennes, une pour dire si deux points sont égaux, une autre pour dire si deux points sont le même objet. Dans les deux cas le deuxième point est donné par un pointeur :

```
class Point {
    ...
    bool pointEgal(Point *pt) {
        return pt->x == x && pt->y == y;
    }
    bool memePoint(Point *pt) {
        return pt == this;
    }
    ...
};
```

2.2.3 Membres publics et privés

Par défaut, les membres des classes sont privés. Les mots clés `public` et `private` permettent de modifier les droits d'accès des membres :

```
class nom {
    les membres déclarés ici sont privés
public:
    les membres déclarés ici sont publics
private:
    les membres déclarés ici sont privés
etc.
};
```

[9] Sauf dans le cas d'une fonction membre *statique*, voir la section 2.8.

Les expressions `public:` et `private:` peuvent apparaître un nombre quelconque de fois dans une classe. Les membres déclarés après `private:` (resp. `public:`) sont privés (resp. publics) jusqu'à la fin de la classe, ou jusqu'à la rencontre d'une expression `public:` (resp. `private:`).

Un membre public d'une classe peut être accédé partout où il est visible ; un membre privé ne peut être accédé que depuis une fonction membre de la classe (les notions de membre *protégé*, cf. section 4.2.1, et de classes et fonctions *amies*, cf. section 2.9, nuanceront cette affirmation).

Si `p` est une expression de type `Point` :

– dans une fonction qui n'est pas membre ou amie de la classe `Point`, les expressions `p.x` ou `p.y` pourtant syntaxiquement correctes et sans ambiguïté, constituent des accès illégaux aux membres privés `x` et `y` de la classe `Point`,

– les expressions `p.afficher()` ou `p.placer(u, v)` sont des accès légaux aux membres publics `afficher` et `placer`, qui se résolvent en des accès parfaitement légaux aux membres `p.x` et `p.y`.

2.2.4 Encapsulation au niveau de la classe

Les fonctions membres d'une classe ont le droit d'accéder à tous les membres *de la classe* : deux objets de la même classe ne peuvent rien se cacher. Par exemple, le programme suivant montre notre classe `Point` augmentée d'une fonction pour calculer la distance d'un point à un autre :

```
class Point {
public:
    void afficher() {
        cout << '(' << x << ',' << y << ')';
    }
    void placer(int a, int b) {
        validation des valeurs de a et b;
        x = a; y = b;
    }
    double distance(Point autrePoint) {
        int dx = x - autrePoint.x;
        int dy = y - autrePoint.y;
        return sqrt(dx * dx + dy * dy);
    }
private:
    int x, y;
};
```

Lors d'un appel tel que `p.distance(q)` l'objet `p` accède aux membres privés `x` et `y` de l'objet `q`. On dit que *C++ pratique l'encapsulation au niveau de la classe*, non au niveau de l'objet.

On notera au passage que, contrairement à d'autres langages orientés objets, en C++ encapsuler n'est pas *cacher* mais *interdire*. Les usagers d'une classe voient les membres privés de cette dernière, mais ne peuvent pas les utiliser.

2.2.5 Structures

Une structure est la même chose qu'une classe mais, par défaut, les membres y sont publics. Sauf pour ce qui touche cette question, tout ce qui sera dit dans la suite à propos des classes s'appliquera donc aux structures :

```
struct nom {
    les membres déclarés ici sont publics
private:
    les membres déclarés ici sont privés
public:
    les membres déclarés ici sont publics
etc.
};
```

2.3 Définition des classes

2.3.1 Définition séparée et opérateur de résolution de portée

Tous les membres d'une classe doivent être au moins déclarés à l'intérieur de la formule `class`*nom*`{...};` qui constitue la déclaration de la classe.

Cependant, dans le cas des fonctions, aussi bien publiques que privées, on peut se limiter à n'écrire que leur en-tête à l'intérieur de la classe et définir le corps ailleurs, plus loin dans le même fichier ou bien dans un autre fichier.

Il faut alors un moyen pour indiquer qu'une définition de fonction, écrite en dehors de toute classe, est en réalité la définition d'une fonction membre d'une classe. Ce moyen est l'*opérateur de résolution de portée*, dont la syntaxe est

NomDeClasse`::`

Par exemple, voici notre classe `Point` avec la fonction `distance` définie séparément :

```
class Point {
public:
    ...
    double distance(Point autrePoint);
    ...
}
```

Il faut alors, plus loin dans le même fichier ou bien dans un autre fichier, donner la définition de la fonction « promise » dans la classe `Point`. Cela s'écrit :

```
double Point::distance(Point autrePoint) {
    int dx = x - autrePoint.x;
    int dy = y - autrePoint.y;
    return sqrt(dx * dx + dy * dy);
};
```

Définir les fonctions membres à l'extérieur de la classe allège la définition de cette dernière et la rend plus compacte. Mais la question n'est pas qu'esthétique, il y a une différence de taille : *les fonctions définies à l'intérieur d'une classe sont implicitement qualifiées « en ligne » (cf. section 1.4).*

Conséquence : la plupart des fonctions membres seront définies séparément. Seules les fonctions courtes, rapides et fréquemment appelées mériteront d'être définies dans la classe.

2.3.2 Fichier d'en-tête et fichier d'implémentation

En programmation orientée objets, « programmer » c'est définir des classes. Le plus souvent ces classes sont destinées à être utilisées dans plusieurs programmes, présents et à venir[10]. Se pose alors la question : comment disposer le code d'une classe pour faciliter son utilisation ?

Voici comment on procède généralement :

- les définitions des classes se trouvent dans des fichiers en-tête (fichiers « `.h` », « `.hpp` », etc.),
- chacun des ces fichiers en-tête contient la définition d'une seule classe ou d'un groupe de classes intimement liées ; par exemple, la définition de notre classe `Point` pourrait constituer un fichier `Point.h`
- les définitions des fonctions membres qui ne sont pas définies à l'intérieur de leurs classes sont écrites dans des fichiers sources (fichiers « `.cpp` » ou « `.cp` »),
- aux programmeurs utilisateurs de ces classes sont distribués :
  - les fichiers « `.h` »
  - le fichiers objets résultant de la compilation des fichiers « `.cpp` »

Par exemple, voici les fichiers correspondants à notre classe `Point` (toujours très modeste) :

Fichier `Point.h` :

```
class Point {
public:
    void placer(int a, int b) {
        validation de a et b
        x = a; y = b;
    }
    double distance(Point autrePoint);
private:
    int x, y;
};
```

[10] La réutilisabilité du code est une des motivations de la méthodologie orientée objets.

Fichier `Point.cpp` :

```
#include "Point.h"
#include <math.h>

double Point::distance(Point autrePoint) {
    int dx = x - autrePoint.x;
    int dy = y - autrePoint.y;
    return sqrt(dx * dx + dy * dy);
}
```

La compilation du fichier `Point.cpp` produira un fichier objet (nommé généralement `Point.o` ou `Point.obj`). Dans ces conditions, la « distribution » de la classe `Point` sera composée des deux fichiers `Point.h` et `Point.obj`, ce dernier ayant éventuellement été transformé en un fichier bibliothèque (nommé alors `Point.lib` ou quelque chose comme ça). Bien entendu, tout programme utilisateur de la classe `Point` devra comporter la directive

```
#include "Point.h"
```

et devra, une fois compilé, être relié au fichier `Point.obj` ou `Point.lib`.

2.4 Constructeurs

2.4.1 Définition de constructeurs

Un constructeur d'une classe est une fonction membre spéciale qui :
- a le même nom que la classe,
- n'indique pas de type de retour,
- ne contient pas d'instruction `return`.

Le rôle d'un constructeur est d'initialiser un objet, notamment en donnant des valeurs à ses données membres. Le constructeur n'a pas à s'occuper de trouver l'espace pour l'objet ; il est appelé (immédiatement) après que cet espace ait été obtenu, et cela quelle que soit la sorte d'allocation qui a été faite : statique, automatique ou dynamique, cela ne regarde pas le constructeur. Exemple :

```
class Point {
public:
    Point(int a, int b) {
        validation des valeurs de a et b
        x = a; y = b;
    }
    ... autres fonctions membres ...
private:
    int x, y;
};
```

Un constructeur de la classe est *toujours* appelé, explicitement (voir ci-dessous) ou implicitement, lorsqu'un objet de cette classe est créé, et en particulier chaque fois qu'une variable ayant cette classe pour type est définie.

C'est le couple *définition de la variable + appel du constructeur* qui constitue la réalisation en C++ du concept « création d'un objet ». L'intérêt pour le programmeur est évident : garantir que, dès leur introduction dans un programme, tous les objets sont garnis et cohérents, c'est-à-dire éviter les variables indéfinies, au contenu incertain.

Une classe peut posséder plusieurs constructeurs, qui doivent alors avoir des signatures différentes :

```
class Point {
public:
    Point(int a, int b) {
        validation de a et b
        x = a; y = b;
    }
    Point(int a) {
        validation de a
        x = a; y = 0;
    }
    Point() {
        x = y = 0;
    }
    ...
private:
    int x, y;
};
```

L'emploi de paramètres avec des valeurs par défaut permet de grouper des constructeurs. La classe suivante possède les mêmes constructeurs que la précédente :

```
class Point {
public:
    Point(int a = 0, int b = 0) {
        validation de a et b
        x = a; y = b;
    }
    ...
private:
    int x, y;
};
```

Comme les autres fonctions membres, les constructeurs peuvent être déclarés dans la classe et définis ailleurs. Ainsi, la classe précédente pourrait s'écrire également

```
class Point {
public:
    Point(int a = 0, int b = 0);
    ...
private:
    int x, y;
};
```

et, ailleurs :

```
Point::Point(int a, int b) {
    validation de a et b
    x = a; y = b;
}
```

Deux remarques générales. 1. Comme l'exemple ci-dessus le montre, lorsqu'une fonction fait l'objet d'une déclaration et d'une définition séparées, comme le constructeur `Point`, les éventuelles valeurs par défaut des argument concernent la déclaration, non la définition.

2. Lorsqu'une fonction fait l'objet d'une déclaration et d'une définition séparées, les noms des arguments ne sont utiles que pour la définition. Ainsi, la déclaration du constructeur `Point` ci-dessus peut s'écrire également :

```
class Point {
    ...
    Point(int = 0, int = 0);
    ...
};
```

2.4.2 Appel des constructeurs

Un constructeur est *toujours* appelé lorsqu'un objet est crée, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a(3, 4);
Point b = Point(5, 6);
```

Dans le cas d'un constructeur avec un seul paramètre, on peut aussi adopter une forme qui rappelle l'initialisation des variables de types primitifs (à ce propos voir aussi la section 3.3.1) :

```
Point e = 7;            // équivaut à : Point e = Point(7)
```

Un objet alloué dynamiquement est lui aussi toujours initialisé, au mois implicitement. Dans beaucoup de cas il peut, ou doit, être initialisé explicitement. Cela s'écrit :

```
Point *pt;
...
pt = new Point(1, 2);
```

Les constructeurs peuvent aussi être utilisés pour initialiser des objets temporaires, anonymes. En fait, chaque fois qu'un constructeur est appelé, un objet nouveau est crée, même si cela ne se passe pas à l'occasion de la définition d'une variable. Par exemple, deux objets sans nom, représentant les points (0,0) et (3,4), sont créés dans l'instruction suivante[11] :

```
cout << Point(0, 0).distance(Point(3, 4)) << "\n";
```

NOTE. L'appel d'un constructeur dans une expression comportant un signe = peut prêter à confusion, à cause de sa ressemblance avec une affectation. Or, en C++, l'initialisation et l'affectation sont deux opérations distinctes, du moins lorsqu'elles concernent des variables d'un type classe : l'initialisation consiste à donner une *première* valeur à une variable au moment où elle commence à exister ; l'affectation consiste à *remplacer* la valeur courante d'une variable par une autre valeur ; les opérations mises en œuvre par le compilateur, constructeur dans un cas, opérateur d'affectation dans l'autre, ne sont pas les mêmes.

Comment distinguer le « = » d'une affectation de celui d'une initialisation ? Grossièrement, lorsque l'expression commence par un type, il s'agit d'une définition et le signe = correspond à une initialisation. Exemple :

```
Point a = Point(1, 2);          // Initialisation de a
```

Cette expression crée la variable `a` et l'initialise en rangeant dans `a.x` et `a.y` les valeurs 1 et 2. En revanche, lorsque l'expression ne commence pas par un type, il s'agit d'une affectation. Exemple :

```
Point a;
...
a = Point(1, 2);               // Affectation de a
```

L'expression ci-dessus est une affectation ; elle crée un point anonyme de coordonnées (1,2) et le recopie sur la variable `a` en remplacement de la valeur courante de cette variable, construite peu avant. On arrive au même résultat que précédemment, mais au prix de deux initialisations et une affectation à la place d'une seule initialisation.

2.4.3 Constructeur par défaut

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètres : ou bien il n'en a pas, ou bien tous ses paramètres ont des valeurs par défaut. Il joue un rôle remarquable, car *il est appelé chaque fois qu'un objet est créé sans qu'il y ait appel explicite d'un constructeur*, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité :

```
Point x;                        // équivaut à : Point x = Point()
Point t[10];                    // produit 10 appels de Point()
Point *p = new Point;           // équivaut à : p = new Point()
Point *q = new Point[10];       // produit 10 appels de Point()
```

SYNTHÈSE D'UN CONSTRUCTEUR PAR DÉFAUT. Puisque tout objet doit être initialisé lors de sa création, si le programmeur écrit une classe *sans aucun constructeur*, alors le compilateur synthétise un constructeur par défaut comme ceci :

[11] Ces objets anonymes ne pouvant servir à rien d'autre dans ce programme, ils seront détruits lorsque cette instruction aura été exécutée.

– si la classe n'a ni objet membre, ni fonction virtuelle, ni classe de base (c'est le cas de notre classe `Point`), alors le constructeur synthétisé est le constructeur trivial, qui consiste à ne rien faire. Les données membres seront créées comme elles l'auraient été en C, c'est-à-dire initialisées par zéro s'il s'agit d'une variable globale, laissées indéterminées s'il s'agit d'une variable locale ou dynamique,
– si la classe a des objets membres ou des classes de base, alors le constructeur synthétisé produit l'appel du constructeur par défaut de chaque objet membre et de chaque classe de base.

Attention. Si au moins un constructeur est défini pour une classe, alors *aucun* constructeur par défaut n'est synthétisé par le compilateur. Par conséquent, ou bien l'un des constructeurs explicitement définis est un constructeur par défaut, ou bien toute création d'un objet devra expliciter des valeurs d'initialisation.

2.4.4 Constructeur par copie (clonage)

Le constructeur par copie d'une classe C est un constructeur dont le premier paramètre est de type « C `&` » (référence sur un C) ou « `const` C `&` » (référence sur un C constant) et dont les autres paramètres, s'ils existent, ont des valeurs par défaut. Ce constructeur est appelé lorsqu'un objet est initialisé en copiant un objet existant. Cela arrive parfois explicitement, mais souvent implicitement, notamment chaque fois qu'un objet est passé comme paramètre par valeur à une fonction ou rendu comme résultat par valeur (c.-à-d. autre qu'une référence) d'une fonction.

Si le programmeur n'a pas défini de constructeur de copie pour une classe, le compilateur synthétise un constructeur par copie consistant en la recopie de chaque membre d'un objet dans le membre correspondant de l'autre objet. Si ces membres sont de types primitifs ou des pointeurs, cela revient à faire la copie « bit à bit » d'un objet sur l'autre.

A titre d'exemple le programme suivant introduit une nouvelle variété de point ; à chacun est associée une étiquette qui est une chaîne de caractères :

```
class PointNomme {
public:
    PointNomme(int a, int b, char *s = "") {
        x = a; y = b;
        label = new char[strlen(s) + 1];
        strcpy(label, s);
    }
    ...
private:
    int x, y;
    char *label;
};
```

Fig. 1 – Un point avec étiquette

La figure 1 représente la structure de ces objets. Lorsqu'un objet comporte des pointeurs, comme ici, l'information qu'il représente (appelons-la l'« objet logique ») ne se trouve pas entièrement incluse dans l'espace contigu que le compilateur connaît (l'« objet technique »), car des morceaux d'information (dans notre exemple le texte de l'étiquette) se trouvent à d'autres endroits de la mémoire. Ainsi, la copie bit à bit que fait le compilateur peut être inadaptée à de tels objets.

La figure 2 montre le résultat de la copie bit à bit d'un objet `Point` telle que la produirait, avec l'actuelle définition de la classe `Point`, la construction d'un objet `b` telle que

```
Point b = a;
```

(`a` est une variable de type `Point` préalablement construite). Bien entendu, la copie du pointeur n'a pas dupliqué la chaîne pointée : les deux objets, l'original et la copie, partagent la même chaîne. Très souvent, ce partage n'est pas souhaitable, car difficile à gérer et dangereux : toute modification de l'étiquette d'un des deux points se répercutera immédiatement sur l'autre.

Pour résoudre ce problème il faut équiper notre classe d'un constructeur par copie :

FIG. 2 – Copie « superficielle » d'un objet

```
class PointNomme {
    ...
    PointNomme(PointNomme &p) {
        x = p.x; y = p.y;
        label = new char[strlen(p.label) + 1];
        strcpy(label, p.label);
    }
    ...
};
```

Maintenant, la copie d'un point entraînera la duplication effective de la chaîne de caractères pointée, comme sur la figure 3.

FIG. 3 – Copie « profonde »

2.5 Construction des objets membres

Lorsque des membres d'une classe sont à leur tour d'un type classe on dit que la classe a des *objets membres*. L'initialisation d'un objet de la classe nécessite alors l'initialisation de ces objets membres. Il en est toujours ainsi, indépendamment du fait que l'on emploie ou non un constructeur explicite, et qu'à son tour ce constructeur appelle ou non explicitement des constructeurs des objets membres.

Lorsque les objets membres n'ont pas de constructeurs par défaut, une syntaxe spéciale[12] permet de préciser les arguments des constructeurs des membres :

NomDeLaClasse`(`*paramètres*`)`
`:` *membre*`(`*paramètres*`),` ... *membre*`(`*paramètres*`) {`
corps du constructeur
`}`

A titre d'exemple, imaginons que notre classe `Point` ne possède pas de constructeur sans arguments, et qu'on doive définir une classe `Segment` ayant deux points pour membres (un segment est déterminé par deux points). Voici comment on devra écrire son constructeur :

```
class Segment {
    Point origine, extremite;
    int epaisseur;
public:
    Segment(int ox, int oy, int ex, int ey, int ep)
            : origine(ox, oy), extremite(ex, ey) {
        epaisseur = ep;
    }
    ...
};
```

NOTE. Si la classe `Point` avait eu un constructeur sans arguments, le *mauvais* constructeur suivant aurait quand-même fonctionné

[12] A titre d'exercice on vérifiera que, sans cette syntaxe spéciale, la construction des objets membres serait impossible.

```
class Segment {
    ...
    Segment(int ox, int oy, int ex, int ey, int ep) {
        origine = Point(ox, oy);              // Version erronée
        extremite = Point(ex, ey);
        epaisseur = ep;
    }
    ...
};
```

mais il faut comprendre que cette version est très maladroite, car faite de deux affectations (les deux lignes qui forment le corps du constructeur ne sont pas des déclarations). Ainsi, au lieu de se limiter à initialiser les membres `origine` et `extrémité`, on procède successivement à

- la construction de `origine` et `extrémité` en utilisant le constructeur sans arguments de la classe `Point`,
- la construction de deux points anonymes, initialisés avec les valeurs de `ox`, `oy`, `ex` et `ey`,
- l'écrasement des valeurs initiales de `origine` et `extrémité` par les deux points ainsi construits.

NOTE. La syntaxe spéciale pour l'initialisation des objets membres peut être utilisée aussi pour initialiser les données membres de types primitifs. Par exemple, le constructeur de `Segment` précédent peut aussi s'écrire :

```
class Segment {
    ...
    Segment(int ox, int oy, int ex, int ey, int ep)
            : origine(ox, oy), extremite(ex, ey), epaisseur(ep) {
    }
    ...
};
```

2.6 Destructeurs

De la même manière qu'il y a des choses à faire pour initialiser un objet qui commence à exister, il y a parfois des dispositions à prendre lorsqu'un objet va disparaître.

Un destructeur est une fonction membre spéciale. Il a le même nom que la classe, précédé du caractère `~`. Il n'a pas de paramètre, ni de type de retour. Il y a donc au plus un destructeur par classe.

Le destructeur d'une classe est appelé lorsqu'un objet de la classe est détruit, juste avant que la mémoire occupée par l'objet soit récupérée par le système.

Par exemple, voici le destructeur qu'il faut ajouter à notre classe `PointNommé`. Sans ce destructeur, la destruction d'un point n'entraînerait pas la libération de l'espace alloué pour son étiquette :

```
class PointNomme {
    ...
    ~PointNomme() {
        delete [] label;
    }
    ...
};
```

Notez que le destructeur n'a pas à s'inquiéter de restituer l'espace occupé par l'« objet technique » lui-même, formé, ici, par les variables `x`, `y` et `label` (le pointeur, non la chaîne pointée). Cette restitution dépend du type de mémoire que l'objet occupe, statique, automatique ou dynamique, et ne regarde pas le destructeur, de la même manière que son allocation ne regardait pas le constructeur qui a initialisé l'objet.

SYNTHÈSE DU DESTRUCTEUR. Si le programmeur n'a pas écrit de destructeur pour une classe, le compilateur en synthétise un, de la manière suivante :

- si la classe n'a ni objets membres ni classes de base (cf. section 4), alors il s'agit du destructeur trivial qui consiste à ne rien faire,
- si la classe a des classes de base ou des objets membres, le destructeur synthétisé consiste à appeler les destructeurs des données membres et des classes de base, dans l'ordre inverse de l'appel des constructeurs correspondants.

2.7 Membres constants

2.7.1 Données membres constantes

Une donnée membre d'une classe peut être qualifiée `const`. Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra par la suite plus être modifiée.

A titre d'exemple voici une nouvelle version de la classe `Segment`, dans laquelle chaque objet reçoit, lors de sa création, un « numéro de série » qui ne doit plus changer au cours de la vie de l'objet :

```
class Segment {
    Point origine, extremite;
    int epaisseur;
    const int numeroDeSerie;
public:
    Segment(int x1, int y1, int x2, int y2, int ep, int num);
    };
```

Constructeur, *version erronée* :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
        : origine(x1, y1), extremite(x2, y2) {
    epaisseur = ep;
    numeroDeSerie = num;       // ERREUR : tentative de modification d'une constante
}
```

Constructeur, version correcte, en utilisant la syntaxe de l'initialisation des objets membres :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
        : origine(x1, y1), extremite(x2, y2), numeroDeSerie(num) {
    epaisseur = ep;
}
```

2.7.2 Fonctions membres constantes

Le mot `const` placé à la fin de l'en-tête d'une fonction membre indique que l'état de l'objet à travers lequel la fonction est appelée n'est pas changé du fait de l'appel. C'est une manière de déclarer qu'il s'agit d'une fonction de *consultation* de l'objet, non d'une fonction de *modification* :

```
class Point {
    ...
    void placer(int a, int b);          // modifie l'objet
    float distance(Point p) const;      // ne modifie pas l'objet
    ...
};
```

A l'intérieur d'une fonction `const` d'une classe C le pointeur `this` est de type « `const` C `* const` » (pointeur constant vers un C *constant*) : l'objet pointé par `this` ne pourra pas être modifié. Cela permet au compilateur d'autoriser certains accès qui, sans cela, auraient été interdits. Par exemple, examinons la situation suivante :

```
void uneFonction(const Point a) {
    Point b;
    ...
    double d = a.distance(b);
    ...
}
```

la qualification `const` de la fonction `distance` est indispensable pour que l'expression précédente soit acceptée par le compilateur. C'est elle seule, en effet, qui garantit que le point `a`, contraint à rester constant, ne sera pas modifié par l'appel de `distance`.

Il est conseillé de qualifier `const` toute fonction qui peu l'être : comme l'exemple précédent le montre, cela élargit son champ d'application.

Coexistence des fonctions constantes et non constantes. La qualification `const` d'une fonction membre fait partie de sa signature. Ainsi, on peut surcharger une fonction membre non constante par une fonction membre constante ayant, à part cela, le même en-tête. La fonction non constante sera appelée sur les objets non constants, la fonction constante sur les objets constants.

On peut utiliser cette propriété pour écrire des fonctions qui n'effectuent pas le même traitement ou qui ne rendent pas le même type de résultat lorsqu'elles sont appelées sur un objet constant et lorsqu'elles sont appelées sur un objet non constant. Exemple (se rappeler que le résultat rendu par une fonction ne fait pas partie de sa signature) :

```
class Point {
    int x, y;
public:
    int  X() const { return x; }
    int  Y() const { return y; }
    int& X() { return x; }
    int& Y() { return y; }
    ...
};
```

Avec la déclaration précédente, les fonctions `X` et `Y` sont sécurisées : sur un objet constant elles ne permettent que la consultation, sur un objet non constant elles permettent la consultation et la modification :

```
const Point a(2, 3);
Point b(4,5);
int r;
...
r = a.X();          // Oui
a.X() = r;          // ERREUR (a.X() rend une valeur)
r = b.X();          // Oui
b.X() = r;          // Oui (b.X() rend une référence)
```

2.8 Membres statiques

Chaque objet d'une classe possède son propre exemplaire de chaque membre ordinaire (bientôt nous dirons membre non statique) de la classe :

- pour les données membres, cela signifie que de la mémoire nouvelle est allouée lors de la création de chaque objet ;
- pour les fonctions membres, cela veut dire qu'elles ne peuvent être appelées qu'en association avec un objet (on n'appelle pas « la fonction f » mais « la fonction f sur l'objet x »).

A l'opposé de cela, les membres *statiques*, signalés par la qualification `static` précédant leur déclaration, sont partagés par tous les objets de la classe. De chacun il n'existe qu'un seul exemplaire par classe, quel que soit le nombre d'objets de la classe.

Les données et fonctions membres non statiques sont donc ce que dans d'autres langages orientés objets on appelle *variables d'instance* et *méthodes d'instance*, tandis que les données et fonctions statiques sont appelées dans ces langages *variables de classe* et *méthodes de classe*.

La visibilité et les droits d'accès des membres statiques sont régis par les mêmes règles que les membres ordinaires.

2.8.1 Données membres statiques

```
class Point {
    int x, y;
public:
    static int nombreDePoints;
    Point(int a, int b) {
        x = a; y = b;
        nbrPoints++;
    }
};
```

Chaque objet `Point` possède ses propres exemplaires des membres `x` et `y` mais, quel que soit le nombre de points existants à un moment donné, il existe un seul exemplaire du membre `nombreDePoints`.

Initialisation. La ligne mentionnant `nombreDePoints` dans la classe `Point` est une simple « annonce », comme une déclaration `extern` du langage C. Il faut encore créer et initialiser cette donnée membre (ce qui, pour une donnée membre non statique, est fait par le constructeur lors de la création de chaque objet). Cela

se fait par une formule analogue à une définition de variable, écrite dans la portée globale, même s'il s'agit de membres privés :

```
int Point::nombreDePoints = 0;
```

(la ligne ci-dessus doit être écrite dans un fichier « `.cpp` », non dans un fichier « `.h` ») L'accès à un membre statique depuis une fonction membre de la même classe s'écrit comme l'accès à un membre ordinaire (voyez l'accès à `nombreDePoints` fait dans le constructeur `Point` ci-dessus).

L'accès à un membre statique depuis une fonction non membre peut se faire à travers un objet, n'importe lequel, de la classe :

```
Point a, b, c;
...
cout << a.nombreDePoints << "\n";
```

Mais, puisqu'il y a un seul exemplaire de chaque membre statique, l'accès peut s'écrire aussi indépendamment de tout objet, par une expression qui met bien en évidence l'aspect « variable de classe » des données membres statiques :

```
cout << Point::nombreDePoints << "\n";
```

2.8.2 Fonctions membres statiques

Une fonction membre statique n'est pas attachée à un objet. Par conséquent :
– elle ne dispose pas du pointeur `this`,
– de sa classe, elle ne peut référencer que les fonctions et les membres statiques.

Par exemple, voici la classe `Point` précédente, dans laquelle le membre `nombreDePoints` a été rendu privé pour en empêcher toute modification intempestive. Il faut donc fournir une fonction pour en consulter la valeur, nous l'avons appelée `combien` :

```
class Point {
    int x, y;
    static int nombreDePoints;
public:
    static int combien() {
        return nombreDePoints;
    }
    Point(int a, int b) {
        x = a; y = b;
        nbrPoints++;
    }
};
```

Pour afficher le nombre de points existants on devra maintenant écrire une expression comme (`a` étant de type `Point`) :

```
cout << a.combien() << "\n";
```

ou, encore mieux, une expression qui ne fait pas intervenir de point particulier :

```
cout << Point::combien() << "\n";
```

2.9 Amis

2.9.1 Fonctions amies

Une fonction amie d'une classe C est une fonction qui, sans être membre de cette classe, a le droit d'accéder à tous ses membres, aussi bien publics que privés.

Une fonction amie doit être déclarée ou définie dans la classe qui accorde le droit d'accès, précédée du mot réservé `friend`. Cette déclaration doit être écrite indifféremment parmi les membres publics ou parmi les membres privés :

```
class Tableau {
    int *tab, nbr;
    friend void afficher(const Tableau &);
public:
    Tableau(int nbrElements);
    ...
};
```

et, plus loin, ou bien dans un autre fichier :

```
void afficher(const Tableau &t) {
    cout << '[';
    for (int i = 0; i < t.nbr; i++)
        cout << ' ' << t.tab[i];
    cout << " ];"
}
```

NOTE. Notez cet effet de la qualification `friend` : bien que déclarée à l'intérieur de la classe `Tableau`, la fonction `afficher` n'est pas membre de cette classe ; en particulier, elle n'est pas attachée à un objet, et le pointeur `this` n'y est pas défini.

Les exemples précédents ne montrent pas l'utilité des fonctions amies, ce n'est pas une chose évidente. En effet, dans la plupart des cas, une fonction amie peut avantageusement être remplacée par une fonction membre :

```
class Tableau {
    int *tab, nbr;
public:
    void afficher() const; //maintenant c'est une fonction membre
    ...
};
```

avec la définition :

```
void Tableau::afficher() {
    cout << '[';
    for (int i = 0; i < nbr; i++)
        cout << ' ' << tab[i];
    cout << " ];"
}
```

Il y a cependant des cas de figure où une fonction doit être nécessairement écrite comme une amie d'une classe et non comme un membre ; un de ces cas est celui où la fonction doit, pour des raisons diverses, être membre d'une autre classe. Imaginons, par exemple, que la fonction `afficher` doive écrire les éléments d'un objet `Tableau` dans un certain objet `Fenêtre` :

```
class Tableau;              // ceci « promet » la classe Tableau

class Fenetre {
    ostream &fluxAssocie;
public:
    void afficher(const Tableau &);
    ...
};

class Tableau {
    int *tab, nbr;
public:
    friend void Fenetre::afficher(const Tableau&);
    ...
};
```

Maintenant, la fonction `afficher` est *membre* de la classe `Fenetre` et *amie* de la classe `Tableau` : elle a tous les droits sur les membres privés de ces deux classes, et son écriture s'en trouve facilitée :

```
void Fenetre::afficher(const Tableau &t) {
    for (int i = 0; i < t.nbr; i++)
        fluxAssocie << t.tab[i];
}
```

2.9.2 Classes amies

Une classe amie d'une classe C est une classe qui a le droit d'accéder à tous les membres de C. Une telle classe doit être déclarée dans la classe C (la classe qui accorde le droit d'accès), précédée du mot réservé `friend`, indifféremment parmi les membres privés ou parmi les membres publics de C.

Exemple : les deux classes `Maillon` et `Pile` suivantes implémentent la structure de données *pile* (structure « dernier entré premier sorti ») d'entiers :

```
class Maillon {
    int info;
    Maillon *suivant;
    Maillon(int i, Maillon *s) {
        info = i; suivant = s;
    }
    friend class Pile;
};

class Pile {
    Maillon *top;
public:
    Pile() {
        top = 0;
    }
    bool vide() {
        return top == 0;
    }
    void empiler(int x) {
        top = new Maillon(x, top);
    }
    int sommet() {
        return top->info;
    }
    void depiler() {
        Maillon *w = top;
        top = top->suivant;
        delete w;
    }
};
```

On notera la particularité de la classe `Maillon` ci-dessus : tous ses membres sont privés, et la classe `Pile` est son amie (on dit que `Maillon` est une classe « esclave » de la classe `Pile`). Autrement dit, seules les piles ont le droit de créer et de manipuler des maillons ; le reste du système n'utilise que les piles et leurs opérations publiques, et n'a même pas à connaître l'existence des maillons.

Exemple d'utilisation :

```
Pile p;
int x;

cout << "? "; cin >> x;
while (x >= 0) {
    p.empiler(x);
    cin >> x;
}

while (! p.vide()) {
    cout << p.sommet() << ' ';
    p.depiler();
}
```

La relation d'amitié n'est pas transitive, « les amis de mes amis *ne sont pas* mes amis ». A l'évidence, la notion d'amitié est une entorse aux règles qui régissent les droits d'accès ; elle doit être employée avec une grande modération, et uniquement pour permettre l'écriture de composants intimement associés d'un programme, comme les classes `Maillon` et `Pile` de notre exemple.

3 Surcharge des opérateurs

3.1 Principe

En C++ on peut redéfinir la sémantique des opérateurs du langage, soit pour les étendre à des objets, alors qui n'étaient initialement définis que sur des types primitifs, soit pour changer l'effet d'opérateurs prédéfinis sur des objets. Cela s'appelle *surcharger des opérateurs*.

Il n'est pas possible d'inventer de nouveaux opérateurs ; seuls des opérateurs déjà connus du compilateur peuvent être surchargés. Tous les opérateurs de C++ peuvent être surchargés, sauf les cinq suivants :

`.` `.*` `::` `?` `:` `sizeof`

Il n'est pas possible de surcharger un opérateur appliqué uniquement à des données de type standard : un opérande au moins doit être d'un type classe.

Une fois surchargés, les opérateurs gardent leur pluralité, leur priorité et leur associativité initiales. En revanche, ils perdent leur éventuelle commutativité et les éventuels liens sémantiques avec d'autres opérateurs. Par exemple, la sémantique d'une surcharge de `++` ou `<=` n'a pas à être liée avec celle de `+` ou `<`.

Surcharger un opérateur revient à définir une fonction ; tout ce qui a été dit à propos de la surcharge des fonctions (cf. section 1.6) s'applique donc à la surcharge des opérateurs.

Plus précisément, pour surcharger un opérateur • (ce signe représente un opérateur quelconque) il faut définir une fonction nommée `operator•`. Ce peut être une fonction membre d'une classe ou bien une fonction indépendante. Si elle n'est pas membre d'une classe, alors elle doit avoir au moins un paramètre d'un type classe.

3.1.1 Surcharge d'un opérateur par une fonction membre

Si la fonction `operator•` est membre d'une classe, elle doit comporter un paramètre de moins que la pluralité de l'opérateur : le premier opérande sera l'objet à travers lequel la fonction a été appelée. Ainsi, sauf quelques exceptions :

- « *obj*• » ou « •*obj* » équivalent à « *obj*`.operator•()` »
- « obj_1 • obj_2 » équivaut à « obj_1`.operator•(`obj_2`)` »

Exemple :

```
class Point {
    int x, y;
public:
    Point(int = 0, int = 0);
    int X() const { return x; }
    int Y() const { return y; }
    Point operator+(const Point) const;    //surcharge de + par une fonction membre
    ...
};

Point Point::operator+(const Point q) const {
    return Point(x + q.x, y + q.y);
}
```

Emploi :

```
Point p, q, r;
...
r = p + q;            // compris comme : r = p.operator+(q);
```

3.1.2 Surcharge d'un opérateur par une fonction non membre

Si la fonction `operator•` n'est pas membre d'une classe, alors elle doit avoir un nombre de paramètres égal à la pluralité de l'opérateur. Dans ce cas :

- « *obj*• » ou « •*obj* » équivalent à « `operator•(`*obj*`)` »
- « obj_1 • obj_2 » équivaut à « `operator•(`obj_1`,` obj_2`)` »

Exemple :

```
Point operator+(const Point p, const Point q) {    // surcharge de + par une
    return Point(p.X() + q.X(), p.Y() + q.Y());    // fonction non membre
}
```

Emploi :

```
Point p, q, r;
...
r = p + q;              // compris maintenant comme : r = operator+(p, q);
```

NOTE 1. A cause des conversions implicites (voir la section 3.3.1), la surcharge d'un opérateur binaire *symétrique* par une fonction non membre, comme la précédente, est en général préférable, car les deux opérandes y sont traités symétriquement. Exemple :

```
Point p, q, r;
int x, y;
```

Surcharge de l'opérateur + par une fonction membre :

```
r = p + q;              // Oui : r = p.operator+(q);
r = p + y;              // Oui : r = p.operator+(Point(y));
r = x + q;              // Erreur : x n'est pas un objet
```

Surcharge de l'opérateur + par une fonction non membre :

```
r = p + q;              // Oui : r = operator+(p, q);
r = p + y;              // Oui : r = operator+(p, Point(y));
r = x + q;              // Oui : r = operator+(Point(p), q);
```

NOTE 2. Lorsque la surcharge d'un opérateur est une fonction non membre, on a souvent intérêt, ou nécessité, à en faire une fonction amie. Par exemple, si la classe `Point` n'avait pas possédé les « accesseurs » publics `X()` et `Y()`, on aurait dû surcharger l'addition par une fonction amie :

```
class Point {
    int x, y;
public:
    Point(int = 0, int = 0);
    friend Point operator+(const Point, const Point);
    ...
};

Point operator+(const Point p, Point q) {
    return Point(p.x + q.x, p.y + q.y);
}
```

NOTE 3. Les deux surcharges de + comme celles montrées ci-dessus, par une fonction membre et par une fonction non membre, ne peuvent pas être définies en même temps dans un même programme ; si tel était le cas, une expression comme `p + q` serait trouvée ambiguë par le compilateur. On notera que cela est une particularité de la surcharge des opérateurs, un tel problème ne se pose pas pour les fonctions ordinaires (une fonction membre n'est jamais en compétition avec une fonction non membre).

NOTE 4. La surcharge d'un opérateur binaire par une fonction non membre est carrément obligatoire lorsque le premier opérande est d'un type standard ou d'un type classe défini par ailleurs, que le programmeur ne peut plus étendre (pour un exemple, voyez la section 3.2.1.

3.2 Quelques exemples

3.2.1 Injection et extraction de données dans les flux

L'opérateur `<<`[13] peut être utilisé pour « injecter » des données dans un flux de sortie, ou `ostream` (cf. section 1.8). D'une part, l'auteur de la classe `ostream` a défini des surcharges de `<<`, probablement par des fonctions membres, pour les types connus lors du développement de la bibliothèque standard :

[13] Appliqué à des données de types primitifs, l'opérateur `<<` exprime, en C++ comme en C, l'opération de décalage de bits vers la gauche.

```
class ostream {
    ...
public:
    ...
    ostream& operator<<(int);
    ostream& operator<<(unsigned int);
    ostream& operator<<(long);
    ostream& operator<<(unsigned long);
    ostream& operator<<(double);
    ostream& operator<<(long double);
    etc.
    ...
};
```

D'autre part, le programmeur qui souhaite étendre `<<` aux objets d'une classe qu'il est en train de développer ne peut plus ajouter des membres à la classe `ostream`. Il doit donc écrire une fonction non membre :

```
ostream& operator<<(ostream& o, const Point p) {
    return o << '(' << p.X() << ',' << p.Y() << ')';
}
```

Parfois (ce n'est pas le cas ici) l'écriture de l'opérateur non membre est plus simple si on en fait une fonction amie des classes des opérandes :

```
class Point {
    ...
    friend ostream& operator<<(ostream&, const Point);
};

ostream& operator<<(ostream &o, const Point p) {
    return o << '(' << p.x << ',' << p.y << ')';
}
```

Avec cette surcharge de `<<` nos points s'écrivent sur un flux de sortie comme les données primitives :

```
Point p;
...
cout << "le point trouvé est : " << p << "\n";
```

On peut de manière analogue surcharger l'opérateur `>>` afin d'obtenir une moyen simple pour lire des points. Par exemple, si on impose que les points soient donnés sous la forme (x, y), c'est-à-dire par deux nombres séparés par une virgule et encadrés par des parenthèses :

```
class Point {
    ...
    friend ostream& operator<<(ostream&, const Point);
    friend istream& operator>>(istream&, Point&);
};

istream& operator>>(istream& i, Point& p) {
    char c;
    i >> c;
    if (c == '(') {
        cin >> p.x >> c;
        if (c == ',') {
            cin >> p.y >> c;
            if (c == ')')
                return i;
        }
    }
    cerr << "Erreur de lecture. Programme avorté\n";
    exit(-1);
}
```

Exemple d'utilisation :

```
void main() {
    Point p;
    cout << "donne un point : ";
    cin >> p;
    cout << "le point donné est : " << p << "\n";
}
```

Essai de ce programme :

```
donne un point : ( 2 , 3 )
le point donné est : (2,3)
```

3.2.2 Affectation

L'affectation entre objets est une opération prédéfinie qui peut être surchargée. Si le programmeur ne le fait pas, tout se passe comme si le compilateur avait synthétisé une opération d'affectation consistant en la recopie membre à membre des objets. S'il s'agit d'une classe sans objets membres, sans classe de base et sans fonction virtuelle, cela donne l'opérateur d'affectation trivial, consistant en la copie bit à bit d'une portion de mémoire sur une autre, comme pour la copie des structures du langage C.

Les raisons qui poussent à surcharger l'opérateur d'affectation pour une classe sont les mêmes que celles qui poussent à écrire un constructeur par copie (cf. section 2.4.4). Très souvent, c'est que la classe possède des « bouts dehors », c'est-à-dire des membres de type pointeur, et qu'on ne peut pas se contenter d'une copie superficielle.

L'opérateur d'affectation doit être surchargé par une fonction membre (en effet, dans une affectation on ne souhaite pas que les opérandes jouent des rôles symétriques).

Reprenons les points munis d'étiquettes qui nous ont déjà servi d'exemple :

```
class PointNomme {
public:
    PointNomme(int = 0, int = 0, char * = "");
    PointNomme(const PointNomme&);
    ~PointNomme();
private:
    int x, y;
    char *label;
};

PointNomme::PointNomme(int a, int b, char *s) {
    x = a; y = b;
    label = new char[strlen(s) + 1];
    strcpy(label, s);
}

PointNomme::PointNomme(const PointNomme& p) {
    cout << "PointNomme(const PointNomme&)\n";
    x = p.x; y = p.y;
    label = new char[strlen(p.label) + 1];
    strcpy(label, p.label);
}

PointNomme::~PointNomme() {
    cout << "~PointNomme()\n";
    delete [] label;
}
```

Avec une classe `PointNomme` définie comme ci-dessus, un programme aussi « inoffensif » que le suivant est erroné (et explosif) :

```
void main() {
    PointNomme p(0, 0, "Origine"), q;
    q = p;
}
```

En effet, l'affectation n'ayant pas été surchargée, l'instruction « `q = p ;` » ne fait qu'une recopie bit à bit de l'objet `p` dans l'objet `q`, c'est à dire une copie superficielle (voyez la figure 2 à la page 17). A la fin du programme, les objets `p` et `q` sont détruits, l'un après l'autre. Or, la destruction d'un de ces objets libère la chaîne `label` et rend l'autre objet incohérent, ce qui provoque une erreur fatale lors de la restitution du second objet.

Voici la surcharge de l'opérateur d'affectation qui résout ce problème. Comme il fallait s'y attendre, cela se résume à une destruction de la valeur courante (sauf dans le cas vicieux où on essaierait d'affecter un objet par lui même) suivie d'une copie :

```
class PointNomme {
public:
    PointNomme(int = 0, int = 0, char * = "");
    PointNomme(const PointNomme&);
    PointNomme& operator=(const PointNomme&);
    ~PointNomme();
    ...
};

PointNomme& PointNomme::operator=(const PointNomme& p) {
    if (&p != this) {
        delete [] label;
        x = p.x;
        y = p.y;
        label = new char[strlen(p.label) + 1];
        strcpy(label, p.label);
    }
    return *this;
}
```

3.3 Opérateurs de conversion

3.3.1 Conversion vers un type classe

Pour convertir une donnée de n'importe quel type, primitif ou classe, vers un type classe il suffit d'employer un *constructeur de conversion*. Un tel constructeur n'est pas une notion nouvelle, mais simplement un constructeur qui peut être appelé avec un seul argument. Exemple :

```
class Point {
    ...
public:
    Point(int a) {
      x = y = a;
    }
    ...
};
```

Le programmeur peut appeler explicitement ce constructeur de trois manières différentes :

```
Point p = Point(2);
Point q(3);             // compris comme Point q = Point(3);
Point r = 4;            // compris comme Point r = Point(4);
```

La troisième expression ci-dessus fait bien apparaître l'aspect « conversion » de ce procédé. Il faut savoir qu'un tel constructeur sera aussi appelé implicitement, car le compilateur s'en servira à chaque endroit où, un `Point` étant requis, il trouvera un nombre. Exemple (vu à la section 3.1.2) :

```
Point p;
...
r = p + 5;              // compris comme : r = operator+(p, Point(5));
```

NOTE. Si on les trouve trop dangereuses, on peut empêcher que le compilateur fasse de telles utilisations implicites d'un constructeur à un argument. Il suffit pour cela de le qualifier `explicit` :

```
class Point {
    ...
public:
    explicit Point(int a) {
      x = y = a;
    }
    ...
};
```

Nouvel essai :

```
Point p = Point(1);     // Ok
Point q = 2;            // ERREUR
```

3.3.2 Conversion d'un objet vers un type primitif

C étant une classe et T un type, primitif ou non, on définit la conversion de C vers T par une fonction membre C`::operator`T`()`.

Par exemple, les classes `Point` et `Segment` suivantes sont munies de conversions `int` ↔ `Point` et `Point` ↔ `Segment` :

```
struct Point {
    int x, y;
    ...
    Point(int a) {                      // conversion int → Point
        x = a; y = 0;
    }
    operator int() {                    // conversion Point → int
        return abs(x) + abs(y);         // par exemple...
    }
};

struct Segment {
    Point orig, extr;
    ...
    Segment(Point p)                    // conversion Point → Segment
        : orig(p), extr(p) { }
    operator Point() {                  // conversion Segment → Point
        return Point((orig.x + extr.x) / 2, (orig.y + extr.y) / 2);
    }
};
```

4 Héritage

4.1 Classes de base et classes dérivées

Le mécanisme de l'héritage consiste en la définition d'une classe par réunion des membres d'une ou plusieurs classes préexistantes, dites *classes de base directes*, et d'un ensemble de membres spécifiques de la classe nouvelle, appelée alors *classe dérivée*. La syntaxe est :

```
class classe : dérivation classe, dérivation classe, ... dérivation classe {
    déclarations et définitions des membres spécifiques de la nouvelle classe
}
```

où *dérivation* est un des mots-clés `private`, `protected` ou `public` (cf. section 4.2.2).

Par exemple, voici une classe `Tableau` (tableau « amélioré », en ce sens que la valeur de l'indice est contrôlée lors de chaque accès) et une classe `Pile` qui ajoute à la classe `Tableau` une donnée membre exprimant le niveau de remplissage de la pile et trois fonctions membres qui encapsulent le comportement particulier (« dernier entré premier sorti ») des piles :

```
class Tableau {
    int *tab;
    int maxTab;
public:
    int &operator[](int i) {
        contrôle de la valeur de i
        return tab[i];
    }
    int taille() { return maxTab; }
    ...
};
class Pile : private Tableau {            // Une Pile est un Tableau
    int niveau;                           // avec des choses en plus
public:
    void empiler(int) {
        (*this)[niveau++] = x;
    }
    int depiler() {
        return (*this)[--niveau];
    }
    int taille() { return niveau; }
    ...
};
```

Étant donnée une classe C, une *classe de base* de C est soit une classe de base directe de C, soit une classe de base directe d'une classe de base de C.

L'héritage est appelé *simple* s'il y a une seule classe de base directe, il est dit *multiple* sinon. En C++, l'héritage peut être multiple.

Encombrement. Dans la notion d'héritage il y a celle de réunion de membres. Ainsi, du point de vue de l'occupation de la mémoire, chaque objet de la classe dérivée contient un objet de la classe de base :

Fig. 4 – Un objet `Tableau` et un objet `Pile`

Pour parler de l'ensemble des membres hérités (par exemple, `tab` et `maxTab`) d'une classe de base B qui se trouvent dans une classe dérivée D on dit souvent le *sous-objet* B de l'objet D.

Visibilité. Pour la visibilité des membres (qui n'est pas l'accessibilité, expliquée dans les sections suivantes), il faut savoir que la classe dérivée détermine une portée imbriquée dans la portée de la classe de base. Ainsi, les noms des membres de la classe dérivée masquent les noms des membres de la classe de base.

Ainsi, si `unePile` est un objet `Pile`, dans un appel comme

```
unePile.taille()            // la valeur de unePile.niveau
```

la fonction `taille()` de la classe `Tableau` et celle de la classe `Pile` (c'est-à-dire les fonctions `Tableau::taille()` et `Pile::taille()`) ne sont pas en compétition pour la surcharge, car la deuxième rend tout simplement la première invisible. L'opérateur de résolution de portée permet de remédier à ce masquage (sous réserve qu'on ait le droit d'accès au membre `taille` de la classe `Tableau`) :

```
unePile.Tableau::taille()   // la valeur de unePile.nbr
```

4.2 Héritage et accessibilité des membres

4.2.1 Membres protégés

En plus des membres publics et privés, une classe C peut avoir des membres *protégés*. Annoncés par le mot clé `protected`, ils représentent une accessibilité intermédiaire car ils sont accessibles par les fonctions membres et amies de C et aussi par *les fonctions membres et amies des classes directement dérivées de* C.

Les membres protégés sont donc des membres qui ne font pas partie de l'interface de la classe, mais dont on a jugé que le droit d'accès serait nécessaire ou utile aux concepteurs des classes dérivées.

Imaginons, par exemple, qu'on veuille comptabiliser le nombre d'accès faits aux objets de nos classes `Tableau` et `Pile`. Il faudra leur ajouter un compteur, qui n'aura pas à être public (ces décomptes ne regardent pas les utilisateurs de ces classes) mais qui devra être accessible aux membres de la classe `Pile`, si on veut que les accès aux piles qui ne mettent en œuvre aucun membre des tableaux, comme dans `vide()`, soient bien comptés.

```
class Tableau {
    int *tab;
    int maxTab;
protected:
    int nbrAcces;
public:
    Tableau(int t) {
        nbrAcces = 0;
        tab = new int[maxTab = t];
    }
    int &operator[](int i) {
        contrôle de la valeur de i
        nbrAcces++;
        return tab[i];
    }
};

class Pile : private Tableau {
    int niveau;
public:
    Pile(int t)
        : Tableau(t), niveau(0) { }
    bool vide() {
        nbrAcces++; return niveau == 0;
    }
    void empiler(int x) {
        (*this)[niveau++] = x;
    }
    int depiler() {
        return (*this)[--niveau];
    }
};
```

4.2.2 Héritage privé, protégé, public

En choisissant quel mot-clé indique la dérivation, parmi `private`, `protected` ou `public`, le programmeur détermine l'accessibilité dans la classe dérivée des membres de la classe de base. On notera que :

- cela concerne l'accessibilité des membres, non leur présence (les objets de la classe dérivée contiennent toujours tous les membres de toutes les classes de base),
- la dérivation ne peut jamais servir à *augmenter* l'accessibilité d'un membre.

Conséquence de ces deux points, les objets des classes dérivées ont généralement des membres inaccessibles : les membres privés de la classe de base sont présents dans les objets de la classe dérivée, mais il n'y a aucun moyen d'y faire référence.

Héritage privé. Syntaxe :

`class` *classeDérivée* `:` $\texttt{private}_{optionnel}$ *classeDeBase* `{ ... }`

Le mot clé `private` est optionnel car l'héritage privé est l'héritage par défaut. C'est la forme la plus restrictive d'héritage : voyez la figure 5.

Dans l'héritage privé, l'interface de la classe de base disparaît (l'ensemble des membres publics cessent d'être publics). Autrement dit, on utilise la classe de base pour réaliser l'implémentation de la classe dérivée, mais on s'oblige à écrire une nouvelle interface pour la classe dérivée.

Cela se voit dans l'exemple déjà donné des classes `Tableau` et `Pile` (cf. §). Il s'agit d'héritage privé, car les tableaux ne fournissent que l'implémentation des piles, non leur comportement.

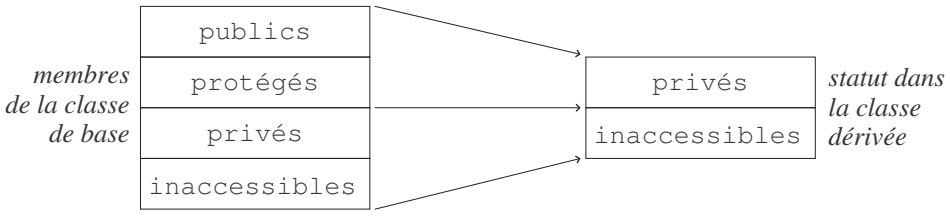

FIG. 5 – Héritage privé

Exemple d'emploi d'un tableau :

```
Tableau t(taille max souhaitée);
...
t[i] = x;
etc.
```

Emploi d'une pile :

```
Pile p(taille max souhaitée);
p[i] = x;                          // ERREUR : l'opérateur [] est inaccessible
p.empiler(x);                      // Oui
...
cout << t[i];                      // ERREUR : l'opérateur [] est inaccessible
cout << p.depiler();               // Oui
etc.
```

A retenir : si D dérive de manière privée de B alors un objet D est une sorte de B, mais les utilisateurs de ces classes n'ont pas à le savoir. Ou encore : ce qu'on peut demander à un B, on ne peut pas forcément le demander à un D.

HÉRITAGE PROTÉGÉ. Syntaxe :

`class` *classeDérivée* `: protected` *classeDeBase* `{ ... }`

Cette forme d'héritage, moins souvent utilisée que les deux autres, est similaire à l'héritage privé (la classe de base fournit l'implémentation de la classe dérivée, non son interface) mais on considère ici que les détails de l'implémentation, c.-à-d. les membres publics et protégés de la classe de base, doivent rester accessibles aux concepteurs d'éventuelles classes dérivées de la classe dérivée.

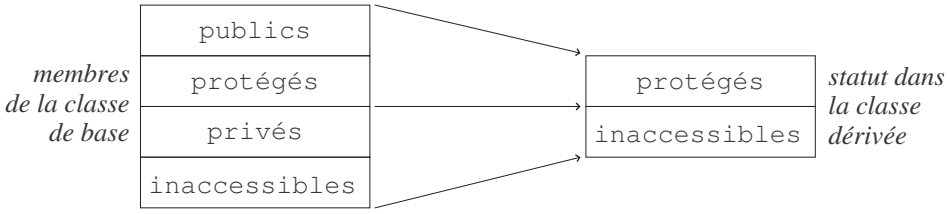

FIG. 6 – Héritage protégé

HÉRITAGE PUBLIC. Syntaxe :

`class` *classeDérivée* `: public` *classeDeBase* `{ ... }`

Dans l'héritage public (voyez la figure 7) l'interface est conservé : tous les éléments du comportement public de la classe de base font partie du comportement de la classe dérivée. C'est la forme d'héritage la plus fréquemment utilisée, car la plus utile et la plus facile à comprendre : dire que D dérive publiquement de B c'est dire que, vu de l'extérieur, tout D est une sorte de B, ou encore que *tout ce qu'on peut demander à un B on peut aussi le demander à un D.*

La plupart des exemples d'héritage que nous examinerons seront des cas de dérivation publique.

4.3 Redéfinition des fonctions membres

Lorsqu'un membre spécifique d'une classe dérivée a le même nom qu'un membre d'une classe de base, le premier *masque* le second, et cela quels que soient les rôles syntaxiques (constante, type, variable, fonction, etc.) de ces membres.

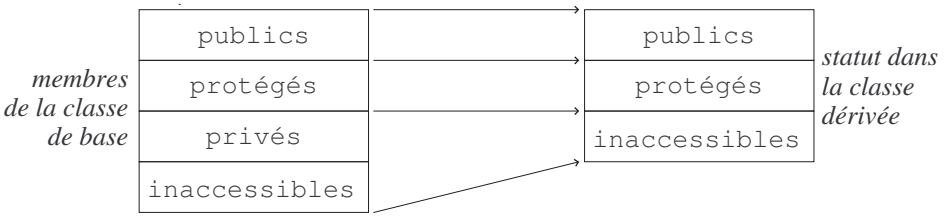

FIG. 7 – Héritage public

La signification de ce masquage dépend de la situation :
– il peut s'agir des noms de deux fonctions membres de même signature,
– il peut s'agir de fonctions de signatures différentes, ou de membres dont l'un n'est pas une fonction.

S'il ne s'agit pas de deux fonctions de même signature, on a une situation maladroite, génératrice de confusion, dont on ne retire en général aucun bénéfice. En particulier, le masquage d'une donnée membre par une autre donnée membre ne fait pas économiser la mémoire, puisque les deux membres existent dans chaque objet de la classe dérivée.

En revanche, le masquage d'une fonction membre de la classe de base par une fonction membre de même signature est une démarche utile et justifiée. On appelle cela une *redéfinition* de la fonction membre.

La justification est la suivante : si la classe D dérive publiquement de la classe B, tout D peut être vu comme une sorte de B, c'est-à-dire un B « amélioré » (augmenté des membres d'autres classes de base, ou de membres spécifiques) ; il est donc naturel qu'un D réponde aux requêtes qu'on peut soumettre à un B, et qu'il y réponde de façon améliorée. D'où l'intérêt de la redéfinition : la classe dérivée donne des versions analogues mais enrichies des fonctions de la classe de base.

Souvent, cet enrichissement concerne le fait que les fonctions redéfinies accèdent à ce que la classe dérivée a de plus que la classe de base. Exemple : un `Pixel` est un point amélioré (c.-à-d. augmenté d'un membre supplémentaire, sa couleur). L'affichage d'un pixel consiste à l'afficher en tant que point, avec des informations additionnelles. Le code suivant reflète tout cela :

```
class Point {
    int x, y;
public:
    Point(int, int);
    void afficher() {                          // affichage d'un Point
        cout << '(' << x << ',' << y << ')';
    }
    ...
};
class Pixel : public Point {
    char *couleur;
public:
    Pixel(int, int, char *);
    void afficher() {                          // affichage d'un Pixel
        cout << '[';
        Point::afficher();                     // affichage en tant que Point
        cout << ';' << couleur << ']';
    }
    ...
};
```

Utilisation :

```
Point pt(2, 3);
Pixel px(4, 5, "rouge");
...
pt.afficher();                    // affichage obtenu : (2,3)
px.afficher();                    // affichage obtenu : [(4,5);rouge]
```

4.4 Création et destruction des objets dérivés

Lors de la construction d'un objet, ses sous-objets hérités sont initialisés selon les constructeurs des classes de base[14] correspondantes. S'il n'y a pas d'autre indication, il s'agit des constructeurs par défaut.

Si des arguments sont requis, il faut les signaler dans le constructeur de l'objet selon une syntaxe voisine de celle qui sert à l'initialisation des objets membres (cf. section 2.5) :

classe`(`*paramètres*`)`
`:` *classeDeBase*`(`*paramètres*`),` ... *classeDeBase*`(`*paramètres*`) {`
corps du constructeur
`}`

De la même manière, lors de la destruction d'un objet, les destructeurs des classes de base directes sont toujours appelés. Il n'y a rien à écrire, cet appel est toujours implicite. Exemple :

```
class Tableau {
    int *table;
    int nombre;
public:
    Tableau(int n) {
        table = new int[nombre = n];
    }
    ~Tableau() {
        delete [] table;
    }
    ...
};

class Pile : private Tableau {
    int niveau;
public:
    Pile(int max)
            : Tableau(max) {
        niveau = 0;
    }
    ~Pile() {
        if (niveau > 0)
            erreur("Destruction d'une pile non vide");
        }                                  // ceci est suivi d'un appel de ~Tableau()
    ...
};
```

4.5 Récapitulation sur la création et destruction des objets

Ayant examiné le cas de l'héritage nous pouvons donner ici, sous une forme résumée, la totalité de ce qu'il y a à dire à propos de la construction et le destruction des objets dans le cas le plus général.

4.5.1 Construction

1. Sauf pour les objets qui sont les valeurs de variables globales, le constructeur d'un objet est appelé immédiatement après l'obtention de l'espace pour l'objet.
   On peut considérer que l'espace pour les variables globales existe dès que le programme est chargé. Les constructeurs de ces variables sont appelés, *dans l'ordre des déclarations de ces variables*, juste avant l'activation de la fonction principale du programme (en principe `main`).
2. L'activation d'un constructeur commence par les appels des constructeurs de chacune de ses classes de base directes, *dans l'ordre de déclaration de ces classes de base directes.*
3. Sont appelés ensuite les constructeurs de chacun des objets membres, *dans l'ordre de déclaration de ces objets membres.*
4. Enfin, les instructions qui composent le corps du constructeur sont exécutées.

[14] Il s'agit ici, tout d'abord, des classes de base directes ; mais les constructeurs de ces classes de base directes appelleront à leur tour les constructeurs de leurs classes de base directes, et ainsi de suite. En définitive, les constructeurs de *toutes* les classes de base auront été appelés.

Sauf indication contraire, les constructeurs dont il est question aux points 2 et 3 sont les constructeurs sans argument des classes de base et des classes des données membres. Si des arguments doivent être précisés (par exemple parce que certaines de ces classes n'ont pas de constructeur sans argument) cela se fait selon la syntaxe :

```
classe(parametres)
        : nomDeMembreOuDeClasseDeBase(paramètres),
          ...
          nomDeMembreOuDeClasseDeBase(paramètres) {
    corps du constructeur
    }
```

Attention. Notez bien que, lorsque les constructeurs des classes de base et des objets membres sont explicitement appelés dans le constructeur de la classe dérivée, ces appels ne sont pas effectués dans l'ordre où le programmeur les a écrits : comme il a été dit aux points 2 et 3 ci-dessus, les appels de ces constructeurs sont toujours faits dans l'ordre des déclarations des classes de base et des objets membres.

4.5.2 Destruction

Grosso modo, le principe général est celui-ci : les objets « contemporains » (attachés à un même contexte, crées par une même déclaration, etc.) sont détruits dans l'ordre inverse de leur création. Par conséquent

1. L'exécution du destructeur d'un objet commence par l'exécution des instructions qui en composent le corps.
2. Elle continue par les appels des destructeurs des classes des objets membres, dans l'ordre inverse de leurs déclarations.
3. Sont appelés ensuite les destructeurs des classes de base directes, dans l'ordre inverse des déclarations de ces classes de base.
4. Enfin, l'espace occupé par l'objet est libéré.

A quel moment les objets sont-ils détruits ? Les objets qui sont les valeurs de variables globales sont détruits immédiatement après la terminaison de la fonction principale (en principe `main`), dans l'ordre inverse de la déclaration des variables globales.

Les objets qui ont été alloués dynamiquement ne sont détruits que lors d'un appel de `delete` les concernant.

Les objets qui sont des valeurs de variables automatiques (locales) sont détruits lorsque le contrôle quitte le bloc auquel ces variables sont rattachées ou, au plus tard, lorsque la fonction contenant leur définition se termine.

Objets temporaires. Les objets temporaires ont les vies les plus courtes possibles. En particulier, les objets temporaires créés lors de l'évaluation d'une expression sont détruits avant l'exécution de l'instruction suivant celle qui contient l'expression.

Les objets temporaires créés pour initialiser une référence persistent jusqu'à la destruction de cette dernière. Exemple :

```
{
Point &r = Point(1, 2);      // l'objet Point temporaire créé ici vit autant que r,
                             // c'est-à-dire au moins jusqu'à la fin du bloc
...
}
```

Attention, les pointeurs ne sont pas traités avec autant de soin. Il ne faut jamais initialiser un pointeur avec l'adresse d'un objet temporaire (l'opérateur `new` est fait pour cela) :

```
{
Point *p = & Point(3, 4);        // ERREUR
Point *q = new Point(5, 6);      // Oui
```

ici, le pointeur `p` *est invalide : le point de coordonnées (3,4) a déjà été détruit*

```
delete q;                        // Oui
delete p;                        // ERREUR (pour plusieurs raisons)
}
```

4.6 Polymorphisme

4.6.1 Conversion standard vers une classe de base

Si la classe D dérive publiquement de la classe B alors les membres de B sont membres de D. Autrement dit, les membres publics de B peuvent être atteints à travers un objet D. Ou encore : *tous les services offerts*

par un B sont offerts par un D.

Par conséquent, là où un B est prévu, on doit pouvoir mettre un D. C'est la raison pour laquelle la conversion (explicite ou implicite) d'un objet de type D vers le type B, une classe de base accessible de D, est définie, et a le statut de conversion standard. Exemple :

```
class Point {                          // point « géométrique »
    int x, y;
public:
    Point(int, int);
    ...
    };

class Pixel : public Point {           // pixel = point coloré
    char *couleur;
public:
    Pixel(int, int, char *);
    ...
    };
```

Utilisation :

```
Pixel px(1, 2, "rouge");
Point pt = px;                         // un pixel a été mis là où un point était
                                       // attendu : il y a conversion implicite
```

De manière interne, la conversion d'un D vers un B est traitée comme l'appel d'une fonction membre de D qui serait publique, protégée ou privée selon le mode (ici : `public`) dont D dérive de B.

Ainsi, la conversion d'une classe dérivée vers une classe de base privée ou protégée existe mais n'est pas utilisable ailleurs que depuis l'intérieur de la classe dérivée. Bien entendu, le cas le plus intéressant est celui de la dérivation publique. Sauf mention contraire, dans la suite de cette section nous supposerons être dans ce cas.

Selon qu'elle s'applique à des objets ou à des pointeurs (ou des références) sur des objets, la conversion d'un objet de la classe dérivée vers la classe de base recouvre deux réalités très différentes :

1. Convertir un objet D vers le type B c'est lui enlever tous les membres qui ne font pas partie de B (les membres spécifiques et ceux hérités d'autres classes). Dans cette conversion il y a perte effective d'information : voyez la figure 8.

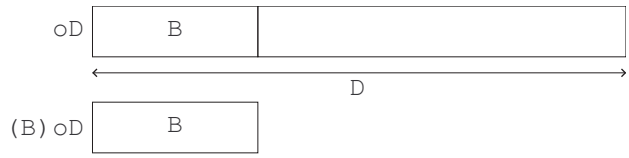

FIG. 8 – Conversion de `oD`, un objet D, en un objet B

2. Au contraire, convertir un $D*$ en un $B*$ (c'est-à-dire un pointeur sur D en un pointeur sur B) ne fait perdre aucune information. Pointé à travers une expression de type $B*$, l'objet n'offre que les services de la classe B, mais il ne cesse pas d'être un D avec tous ses membres : voyez la figure 9.

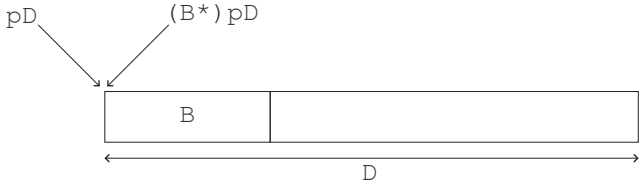

FIG. 9 – Conversion de `pD`, un pointeur sur un D, en un pointeur sur un B

Par exemple, supposons posséder deux fonctions de prototypes :

```
void fon1(Point pt);
void fon2(Point *pt);
```

et supposons qu'elles sont appelées de la manière suivante :

```
Pixel pix = Pixel(2, 3, "rouge")
...
fon1(pix);
fon2(&pix);
```

Le statut de l'objet `pix` en tant que paramètre de ces deux fonctions est tout à fait différent. L'objet passé à `fon1` comme argument est une version tronquée de `pix`, alors que le pointeur passé à `fon2` est l'adresse de l'objet `pix` tout entier :

```
void fon1(Point pt) {
    La valeur de pt est un Point ; il n'a pas de couleur. Il y avait peut-être
    à l'origine un Pixel mais, ici, aucune conversion (autre que définie
    par l'utilisateur) ne peut faire un Pixel à partir de pt.
}

void fon2(Point *pt) {
    Il n'est rien arrivé à l'objet dont l'adresse a servi à initialiser pt lors de
    l'appel de cette fonction. Si on peut garantir que cet objet est un Pixel,
    l'expression suivante (placée sous la responsabilité du programmeur) a
    un sens :
                ((Pixel *) pt)->couleur ...
}
```

Pour ce qui nous occupe ici, les références (pointeurs gérés de manière interne par le compilateur) doivent être considérées comme des pointeurs. Nous pouvons donc ajouter un troisième cas :

```
void fon3(Point &rf) {
    Il n'est rien arrivé à l'objet qui a servi à initialiser rf lors de l'appel de
    cette fonction. Si on peut garantir que cet objet est un Pixel, l'expression
    suivante (placée sous la responsabilité du programmeur) a un sens :
                ((Pixel &) rf).couleur ...
}
```

4.6.2 Type statique, type dynamique, généralisation

Considérons la situation suivante :

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

D unD;
...
B unB = unD;
B *ptrB = &unD;
B &refB = unD;
```

Les trois affectations ci-dessus sont légitimes ; elles font jouer la conversion standard d'une classe vers une classe de base. Le type de `unB` ne soulève aucune question (c'est un `B`), mais les choses sont moins claires pour les types de `ptrB` et `refB`. Ainsi `ptrB`, par exemple, pointe-t-il un `B` ou un `D` ?

- on dit que le *type statique* de `*ptrB` (ce que `ptB` pointe) et de `refB` est `B`, car c'est ainsi que `ptB` et `refB` ont été *déclarées* ;
- on dit que le *type dynamique* de `*ptrB` et de `refB` est `D`, car tel est le type des *valeurs effectives* de ces variables.

Le type statique d'une expression découle de l'analyse du texte du programme ; il est connu à la compilation. Le type dynamique, au contraire, est déterminé par la valeur courante de l'expression, il peut changer durant l'exécution du programme.

La notion de type dynamique va nous amener assez loin. Pour commencer, remarquons ceci : grâce aux

conversions implicites de pointeurs et références vers des pointeurs et références sur des classes de base, tout objet peut être momentanément tenu pour plus général qu'il n'est. Cela permet les traitements génériques, comme dans l'exemple suivant, qui présente un ensemble de classes conçues pour gérer le stock d'un supermarché[15] :

```
struct Article {
    char *denom;                              // denomination
    int quant;                                // quantité disponible
    double prix;                              // prix HT
    Article(char *d, int q, double p)
        : denom(d), quant(q), prix(p) { }
    ...
};

struct Perissable : public Article {
    Date limite;
    Perissable(char *d, int q, double p, Date l)
        : Article(d, q, p), limite(l) { }
    ...
};

struct Habillement : public Article {
    int taille;
    Habillement(char *d, int q, double p, int t)
        : Article(d, q, p), taille(t) { }
    ...
};

struct Informatique : public Article {
    char garantie;
    Informatique (char *d, int q, double p, char g)
        : Article(d, q, p), garantie(g) { }
    ...
};
```

Introduisons quelques structures de données (des tableaux de pointeurs) pour mémoriser les diverses classes d'articles présents dans le stock :

```
Perissable   *aliment[maxAliment];
Habillement  *vetemts[maxVetemts];
Informatique *info[maxInfo];
```
etc.

Définissons également une structure pour mémoriser *tous* les articles du stock :

```
Article *stock[maxStock];
```

Initialisons tout cela (notez que chaque adresse fournie par un appel de `new` est d'abord copiée dans un tableau spécifique et ensuite généralisée et copiée dans le tableau générique) :

```
nStock = nAliment = nVetemts = nInfo = 0;
stock[nStock++] = aliment[nAliment++] =
            new Perissable("Foie gras Shell 1Kg", 1000, 450.0, Date(15,1,2000));
stock[nStock++] = vetemts[nVetemts] =
            new Habillement("Tenue gendarme fluo rose", 250, 200.0, 52);
stock[nStock++] = info[nInfo++] =
            new Informatique("Pentium III 800 Mhz Diesel HDI", 500, 1000.0, 'C');
```
etc.

En tant qu'éléments du tableau `stock`, les objets des classes `Perissable`, `Habillement`, etc., sont tenus pour plus généraux qu'ils ne sont, ce qui est très commode pour faire sur chacun une opération générique. Par exemple, le calcul de la valeur totale du stock n'a pas besoin des spécificités de chaque article :

[15] Nous déclarons ici des structures (une structure est une classe dont les membres sont par défaut publiques) pour ne pas nous encombrer avec des problèmes de droits d'accès.

```
double valeurStock = 0;
for (int i = 0; i < nStock; i++)
    valeurStock += stock[i]->quant * stock[i]->prix;
```

Dans cet exemple nous avons utilisé la généralisation des pointeurs pour accéder à des membres dont les objets héritent sans les modifier : les notions de quantité disponible et de prix unitaire sont les mêmes pour toutes les sortes d'articles du stock.

La question devient beaucoup plus intéressante lorsqu'on accède à des membres de la classe de base qui sont redéfinis dans les classes dérivées. Voici un autre exemple, classique : la classe `Figure` est destinée à représenter les éléments communs à tout un ensemble de figures géométriques, telles que des triangles, des ellipses, des rectangles, etc. :

```
class Figure {
    ...
};
class Triangle : public Figure {
    ...
};
class Ellipse : public Figure {
    ...
};

Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < n; i++)
    image[i]->seDessiner();
```

Si on suppose que la fonction `seDessiner` (qui doit nécessairement être déclarée dans la classe `Figure`, sinon la ligne ci-dessus serait trouvée incorrecte à la compilation) est redéfinie dans les classes `Triangle`, `Ellipse`, etc., il est intéressant de se demander quelle est la fonction qui sera effectivement appelée par l'expression `image[i]->seDessiner()` écrite ci-dessus.

Par défaut, la détermination du membre accédé à travers une expression comme la précédente se fait durant la compilation, c'est-à-dire d'après le *type statique* du pointeur ou de la référence. Ainsi, dans l'expression

```
image[i]->seDessiner();
```

c'est la fonction `seDessiner` de la classe `Figure` (probablement une fonction bouche-trou) qui est appelée. Cette réalité décevante obéit à des considérations d'efficacité et de compatibilité avec le langage C ; elle semble limiter considérablement l'intérêt de la généralisation des pointeurs...

...heureusement, la section suivante présente un comportement beaucoup plus intéressant !

4.7 Fonctions virtuelles

Soit f une fonction membre d'une classe C. Si les conditions suivantes sont réunies :
- f est redéfinie dans des classes dérivées (directement ou indirectement) de C,
- f est souvent appelée à travers des pointeurs ou des références sur des objets de C ou de classes dérivées de C,

alors f mérite d'être une *fonction virtuelle*. On exprime cela en faisant précéder sa déclaration du mot réservé `virtual`. L'important service obtenu est celui-ci : si f est appelée à travers un pointeur ou une référence sur un objet de C, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de f, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite *classe polymorphe*. La qualification `virtual` devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Exemple :

```
class Figure {
public:
    virtual void seDessiner() {
        fonction passe-partout, probablement sans grand intérêt
    }
    ...
};
class Triangle : public Figure {
public:
    void seDessiner() {
        implémentation du tracé d'un triangle
    }
    ...
};

class Ellipse : public Figure {
public:
    void seDessiner() {
        implémentation du tracé d'une ellipse
    ...
};

Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < N ; i++)
    image[i]->seDessiner();
```

Maintenant, l'appel `image[i]->seDessiner()` active la fonction redéfinie dans la classe de l'objet effectivement pointé par `image[i]`. Cela constitue, bien entendu, le comportement intéressant, celui qu'on recherche presque toujours. On notera que cet appel est légal parce que la fonction `seDessiner` a été déclarée une première fois dans la classe qui correspond au type statique de `image[i]`.

En règle générale, les diverses redéfinitions d'une fonction peuvent être vues comme des versions de plus en plus spécialisées d'un traitement spécifié de manière générique dans la classe de base C. Dans ces conditions, appeler une fonction virtuelle f à travers un pointeur p sur un objet de C c'est demander l'exécution de la version de f la plus spécialisée qui s'applique à ce que p pointe en fait à l'instant où l'appel est exécuté.

Contraintes En concurrence avec le mécanisme du masquage, la redéfinition des fonctions virtuelles subit des contraintes fortes, sur les types des paramètres et le type du résultat :

1. Tout d'abord, la signature (liste des types des arguments, sans le type du résultat) de la redéfinition doit être strictement la même que celle de la première définition, sinon la deuxième définition n'est pas une redéfinition mais un masquage pur et simple de la première.

2. La contrainte sur le résultat garantit que l'appel d'une fonction virtuelle est correct et sans perte d'information quoi qu'il arrive. Supposons que f soit une fonction virtuelle d'une classe C rendant un résultat de type T_1, et qu'une redéfinition de f dans une classe dérivée de C rende un résultat de type T_2. Durant la compilation, une expression telle que (`p` est de type C`*`) :

```
p->f()
```

est vue comme de type T_1 ; le compilateur vérifiera que ce type correspond à ce que le contexte de cette expression requiert. Or, à l'exécution, cette expression peut renvoyer un objet de type T_2. Pour que cela soit cohérent et efficace, il faut que le type T_2 puisse être converti dans le type T_1, et cela sans avoir à modifier (tronquer) la valeur renvoyée. Il faut donc :

- soit $T_1 = T_2$,
- soit T_1 est un pointeur [resp. une référence] sur un objet d'une classe C_1, T_2 est un pointeur [resp. une référence] sur un objet d'une classe C_2 et C_1 est une classe de base accessible de C_2.

Autrement dit, si une fonction virtuelle f rend l'adresse d'un T, toute redéfinition de f doit rendre l'adresse *d'une sorte de* T ; si f rend une référence sur un T, toute redéfinition de f doit rendre une référence sur une

sorte de T ; enfin, si f ne rend ni un pointeur ni une référence, toute redéfinition de f doit rendre la même chose que f.

Exemple :

```
class Figure {
public:
    virtual Figure *symetrique(Droite *d);
    ...
};
class Triangle : public Figure {
public:
    Triangle *symetrique(Droite *d);
    ...
};
```

Exemple d'utilisation (notez que la contrainte sur le résultat assure la correction de l'initialisation de `f2`, quel que soit le type dynamique de `f1`) :

```
Figure *f1 = new nomDuneClasseDérivéeDeFigure(...);
Figure *f2 = f1->symetrique(axe);
```

Les deux contraintes indiquées sont examinées une après l'autre : si la contrainte sur la signature (ici `(Droite *)`) est satisfaite, le compilateur détecte un cas de redéfinition de fonction virtuelle et, alors seulement, il vérifie la contrainte sur le résultat (ici la contrainte est satisfaite car le résultat est un pointeur dans les deux cas et `Figure` est une classe de base accessible de `Triangle`). Si cette dernière n'est pas satisfaite, il annonce une erreur.

4.8 Classes abstraites

Les fonctions virtuelles sont souvent introduites dans des classes placées à des niveaux si élevés de la hiérarchie (d'héritage) qu'on ne peut pas leur donner une implémentation utile ni même vraisemblable.

Une première solution consiste à en faire des fonctions vides :

```
class Figure {
public:
    virtual void seDessiner() {
    }              // sera définie dans les classes dérivées
    ...
};
```

En fait, si la fonction `seDessiner` peut être vide c'est parce que :
- la classe `Figure` ne doit pas avoir d'objets,
- la classe `Figure` doit avoir des classes dérivées (qui, elles, auront des objets),
- la fonction `seDessiner` doit être redéfinie dans les classes dérivées.

Bref, la classe `Figure` est une abstraction. Un programme ne créera pas d'objets `Figure`, mais des objets `Rectangle`, `Ellipse`, etc., de classes dérivées de `Figure`. Le rôle de la classe `Figure` n'est pas de produire des objets, mais de représenter ce que les objets rectangles, ellipses, etc., ont en commun ; or, s'il y a une chose que ces objets ont en commun, c'est bien la faculté de se dessiner. Nous dirons que `Figure` est une *classe abstraite*.

On remarque que la fonction `seDessiner` introduite au niveau de la classe `Figure` n'est pas un service rendu aux programmeurs des futures classes dérivées de `Figure`, mais une contrainte : son rôle n'est pas de dire ce qu'est le dessin d'une figure, mais d'obliger les futures classes dérivées de `Figure` à le dire. On peut traduire cet aspect de la chose en faisant que cette fonction déclenche une erreur. En effet, si elle est appelée, c'est que ou bien on a créé un objet de la classe `Figure`, ou bien on a créé un objet d'une classe dérivée de `Figure` dans laquelle on n'a pas redéfini la fonction `seDessiner` :

```
class Figure {
public:
    virtual void seDessiner() {
        erreur("on a oublié de définir la fonction seDessiner");
    }
    ...
};
```

Cette manière de faire est peu pratique, car l'erreur ne sera déclenchée qu'à l'exécution (c'est le programmeur qui a commis l'oubli mais c'est l'utilisateur qui se fera réprimander !). En C++ on a une meilleure solution : définir une *fonction virtuelle pure*, par la syntaxe :

```
class Figure {
public:
    virtual void seDessiner() = 0;          // Une fonction virtuelle pure
    ...
};
```

D'une part, cela « officialise » le fait que la fonction ne peut pas, à ce niveau, posséder une implémentation. D'autre part, cela crée, pour le programmeur, l'obligation de définir cette implémentation dans une classe dérivée ultérieure, et permet de vérifier *pendant la compilation* que cela a été fait. Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition (autre que « `= 0` »).

Pour le compilateur, une *classe abstraite* est une classe qui a des fonctions virtuelles pures. Tenter de créer des objets d'une classe abstraite est une erreur, qu'il signale :

```
class Figure {
public:
    virtual void seDessiner() = 0;
    ...
    };

class Rectangle : public Figure {
    int x1, y1, x2, y2;
public:
    void seDessiner() {
        trait(x1, y1, x2, y1);
        trait(x2, y1, x2, y2);
        trait(x2, y2, x1, y2);
        trait(x1, y2, x1, y1);
    }
    ...
};
```

Utilisation :

```
Figure f;                   // ERREUR : Création d'une instance de la
                            //          classe abstraite Figure
Figure *pt;                 // Oui : c'est un pointeur
pt = new Figure;            // ERREUR : Création d'une instance de la
                            //          classe abstraite Figure
Rectangle r;                // Oui : la classe Rectangle n'est pas abstraite
pt = new Rectangle;         // Oui
```

4.9 Identification dynamique du type

Le coût du polymorphisme, en espace mémoire, est d'un pointeur par objet, quel que soit le nombre de fonctions virtuelles de la classe[16]. Chaque objet d'une classe polymorphe comporte un membre de plus que ceux que le programmeur voit : un pointeur vers une table qui donne les adresses qu'ont les fonctions virtuelles pour les objets de la classe en question.

[16]En temps, le coût du polymorphisme est celui d'une indirection supplémentaire pour chaque appel d'une fonction virtuelle, puisqu'il faut aller chercher dans une table l'adresse effective de la fonction.

D'où l'idée de profiter de l'existence de ce pointeur pour ajouter au langage, sans coût supplémentaire, une gestion des types dynamiques qu'il faudrait sinon écrire au coup par coup (probablement à l'aide de fonctions virtuelles). Fondamentalement, ce mécanisme se compose des deux opérateurs `dynamic_cast` et `typeid`.

4.9.1 L'opérateur `dynamic_cast`

Syntaxe

`dynamic_cast<`*type*`>(`*expression*`)`

Il s'agit d'effectuer la conversion d'une expression d'un type pointeur vers un autre type pointeur, les types pointés étant deux classes polymorphes dont l'une, B, est une classe de base accessible de l'autre, D.

S'il s'agit de généralisation (conversion dans le sens $D* \rightarrow B*$), cet opérateur ne fait rien de plus que la conversion standard.

Le cas intéressant est $B* \rightarrow D*$, conversion qui n'a de sens que si l'objet pointé par l'expression de type $B*$ est en réalité un objet de la classe D ou d'une classe dérivée de D.

L'opérateur `dynamic_cast` fait ce travail de manière sûre et portable, et rend l'adresse de l'objet lorsque c'est possible, 0 dans le cas contraire :

```
class Animal {
    ...
    virtual void uneFonction() {        // il faut au moins une fonction virtuelle
    ...                                 // (car il faut des classes polymorphes)
    }
};

class Mammifere : public Animal {
    ...
};

class Chien : public Mammifere {
    ...
};

class Caniche : public Chien {
    ...
};

class Chat : public Mammifere {
    ...
};

class Reverbere {
    ...
};

Chien medor;
Animal *ptr = &medor;
...

Mammifere *p0 = ptr;
            // ERREUR (à la compilation) : un Animal n'est pas forcément un Mammifère
Mammifere *p1 = dynamic_cast<Mammifere *>(ptr);
            // OK : p1 reçoit une bonne adresse, car Médor est un mammifère
Caniche *p2 = dynamic_cast<Caniche *>(ptr);
            // OK, mais p2 reçoit 0, car Médor n'est pas un caniche
Chat *p3 = dynamic_cast<Chat *>(ptr);
            // OK, mais p3 reçoit 0, car Médor n'est pas un chat non plus
Reverbere *p4 = dynamic_cast<Reverbere *>(ptr);
            // OK, mais p4 reçoit 0, car Médor n'est pas un reverbère
```

L'opérateur `dynamic_cast` s'applique également aux références. L'explication est la même, sauf qu'en cas d'impossibilité d'effectuer la conversion, cet opérateur lance l'exception `bad_cast` :

```
Animal &ref = medor;
Mammifere &r1 = dynamic_cast<Mammifere &>(ref);     // Oui
Caniche &r2 = dynamic_cast<Caniche &>(ref);         // exception bad_cast lancée ;
                                                    // non attrapée, elle est fatale
```

4.9.2 L'opérateur `typeid`

Syntaxes :

`typeid(`*type*`)`

ou

`typeid(`*expression*`)`

Le résultat est une valeur de type `const type_info&`, où `type_info` est une classe de la bibliothèque standard comportant au moins les membres suivants :

```
class type_info {
public:
    const char *name() const;
    int operator==(const type_info&) const;
    int operator!=(const type_info&) const;
    int before(const type_info&) const;
    ...
};
```

On a donc la possibilité de comparer les types dynamiques et d'en obtenir l'écriture des noms. Exemple :

```
Animal *ptr = new Caniche;
cout << typeid(ptr).name() << '\n';
cout << typeid(*ptr).name() << '\n';
cout << "L'animal pointé par ptr "
    << (typeid(*ptr) == typeid(Chien) ? "est" : "n'est pas")
    << " un chien\n";
cout << "L'animal pointé par ptr est un "
    << typeid(*ptr).name() << "\n";
```

affichage obtenu :

```
Animal *
Chien
L'animal pointé par ptr n'est pas un chien
L'animal pointé par ptr est un Caniche
```

5 Modèles (*templates*)

Un modèle définit une famille de fonctions ou de classes paramétrée par une liste d'identificateurs qui représentent des valeurs et des types. Les valeurs sont indiquées par leurs types respectifs, les types par le mot réservé `class`. Le tout est préfixé par le mot réservé `template`. Exemple :

```
template<class T, class Q, int N>
    ici figure la déclaration ou la définition d'une fonction ou d'une classe
    dans laquelle T et Q apparaissent comme types et N comme constante
```

Le mot `class` ne signifie pas que `T` et `Q` sont nécessairement des classes, mais des types. Depuis la norme ISO on peut également utiliser le mot réservé `typename`, qui est certainement un terme plus heureux.

La production effective d'un élément de la famille définie par un modèle s'appelle l'*instanciation*[17] du modèle. Lors de cette opération, les paramètres qui représentent des types doivent recevoir pour valeur des types ; les autres paramètres doivent recevoir des expressions dont la valeur est connue durant la compilation.

[17] Le choix du mot n'est pas très heureux. Dans tous les autres langages à objets, l'instanciation est l'opération de création d'un objet, et on dit qu'un objet est instance de sa classe. En C++ le mot instanciation ne s'utilise qu'en rapport avec les modèles.

5.1 Modèles de fonctions

Il n'y a pas en C++ un opérateur ou un mot réservé expressément destiné à produire des instances des modèles. L'instanciation d'un modèle de fonction

```
template<param_1, ... param_k>
    type nom ( argfor_1, ... argfor_n )...
```

est commandée implicitement par l'appel d'une des fonctions que le modèle définit. Un tel appel peut être écrit sous la forme :

```
nom <arg_1, ... arg_k> (argeff_1, ... argeff_n)
```

dans laquelle les derniers arguments du modèle, arg_i ... arg_k, éventuellement tous, peuvent être omis si leurs valeurs peuvent être déduites sans ambiguïté des arguments effectifs de la fonction, $argeff_i$, ... $argeff_n$.

Exemple : déclaration du modèle « recherche de la valeur minimale d'un tableau (ayant au moins un élément) » :

```
template<class T> T min(T tab[], int n) {
    T min = tab[0];
    for (int i = 1; i < n; i++)
        if (tab[i] < min) min = tab[i];
    return min;
}
```

Voici un programme qui produit et utilise deux instances de ce modèle :

```
int t[] = { 10, 5, 8, 14, 20, 3, 19, 7 };
...
cout << min<int>(t, 8);                          // T = int
cout << min<char>("BKEFYFFLKRNFAJDQKXJD", 20);  // T = char
```

Puisque dans les deux cas l'argument du modèle peut se déduire de l'appel, ce programme s'écrit aussi :

```
cout << min(t, 8);                               // T = int
cout << min("BKEFYFFLKRNFAJDQKXJD", 20);        // T = char
```

♣ Un modèle de fonction peut coexister avec une ou plusieurs spécialisations. Par exemple, la fonction suivante est une spécialisation du modèle `min` qu'on ne peut pas obtenir par instanciation de ce modèle, car à la place de l'opérateur `<` figure un appel de la fonction `strcmp` :

```
char *min(char *tab[], int n) {
    char *min = tab[0];
    for (int i = 1; i < n; i++)
        if (strcmp(tab[i], min) < 0) min = tab[i];
    return min;
}
```

Pour le mécanisme de la surcharge des fonctions, les instances de modèles sont souvent en concurrence avec d'autres instances ou des fonctions ordinaires. Il faut savoir que les fonctions ordinaires sont préférées aux instances de modèles et les instances de modèles plus spécialisés sont préférées aux instances de modèles moins spécialisés.

Par exemple, donnons-nous un programme un peu plus compliqué que le précédent, en écrivant une fonction qui recherche l'indice du minimum d'un tableau, avec un deuxième tableau pour « critère secondaire ». Nous définissons :

1. Un modèle avec deux paramètres-types :

```
template<class T1, class T2>
    int imin(T1 ta[], T2 tb[], int n) {
        cout << "imin(T1 [], T2 [], int) : ";
        int m = 0;
        for (int i = 1; i < n; i++)
            if (ta[i] < ta[m] || ta[i] == ta[m] && tb[i] < tb[m])
                m = i;
        return m;
    }
```

2. Un modèle qui en est une spécialisation partielle (le premier tableau est un tableau de chaînes, il n'y a donc plus qu'un paramètre-type) :

```
template<class T>
    int imin(char *ta[], T tb[], int n) {
        cout << "imin(char *[], T [], int) : ";
        int m = 0, r;
        for (int i = 1; i < n; i++) {
            r = strcmp(ta[i], ta[m]);
            if (r < 0 || r == 0 && tb[i] < tb[m])
                m = i;
            }
        return m;
    }
```

3. Une fonction qui est une spécialisation complète du modèle (deux tableaux de chaînes, plus aucun paramètre-type) :

```
int imin(char *ta[], char *tb[], int n) {
    cout << "imin(char *[], char *[], int) : ";
    int m = 0, r;
    for (int i = 1; i < n; i++) {
        r = strcmp(ta[i], ta[m]);
        if (r < 0 || r == 0 && strcmp(tb[i], tb[m]) < 0)
            m = i;
        }
    return m;
}
```

Essayons tout cela :

```
int ti1[5] = { 10,  5, 15,  5, 18 };
int ti2[5] = { 20,  8, 20, 10, 20 };
char *ts1[5] = { "Andre", "Denis", "Charles", "Andre", "Bernard" };
char *ts2[5] = { "Duchamp", "Dubois", "Dumont", "Dupont", "Durand" };
cout << imin(ti1, ti2, 5) << '\n';
cout << imin(ts1, ti2, 5) << '\n';
cout << imin(ts1, ts2, 5) << '\n';
```

L'affichage obtenu :

```
imin(T1 [], T2 [], int) : 1
imin(char *[], T [], int) : 3
imin(char *[], char *[], int) : 0
```

confirme que les modèles les plus spécialisés sont préférés aux moins spécialisés.

5.2 Modèles de classes

Un modèle de classe est un type paramétré, par d'autres types et par des valeurs connues lors de la compilation.

Par exemple, définissons un modèle pour représenter des tableaux dans lesquels l'indice est vérifié lors de chaque accès. Le type des éléments de ces tables n'est pas connu, c'est un paramètre du modèle :

```
template<class TElement> class Table {
    TElement *tab;
    int nbr;
public:
    Table(int n) {
        tab = new TElement[nbr = n];
    }
    ~Table() {
        delete [] tab;
    }
    TElement &operator[](int i) {
        contrôle de la validité de i
        return tab[i];
    }
```

```
    TElement operator[](int i) const {
        contrôle de la validité de i
        return tab[i];
    }
};
```

On obtient l'instanciation d'un modèle de classe en en créant des objets. Ici, contrairement à ce qui se passe pour les fonctions, les arguments du modèle doivent être toujours explicités :

```
Table<char> message(80);
Table<Point> ligne(100);
```

Une fois déclarés, ces tableaux s'utilisent comme les tableaux du langage (avec en plus la certitude que si l'indice déborde on en sera prévenu tout de suite) :

```
message[0] = ' ';
ligne[i] = Point(j, k);
```

Tout se passe comme si la définition de l'objet message écrite ci-dessus produisait le texte obtenu en substituant `TElement` par `char` dans le modèle `Table` et le présentait au compilateur ; un peu plus tard, la définition de `ligne` produit le texte obtenu en substituant `TElement` par `Point` et le présente également au compilateur.

C'est un traitement apparenté au développement des macros[18] mais bien plus complexe, aussi bien sur le plan formel (les arguments d'un modèle peuvent être des instances d'autres modèles, les modalités de l'instanciation d'un modèle de fonction peuvent être automatiquement déduites de l'appel de la fonction, etc.) que sur le plan pratique (nécessité de ne pas instancier un modèle qui l'a déjà été dans la même unité de compilation, nécessité de reconnaître que des instances du même modèle produites et compilées dans des unités de compilation séparées sont la même entité, etc.).

5.2.1 Fonctions membres d'un modèle

Les fonctions membres d'un modèle de classe sont des modèles de fonctions avec les mêmes paramètres que le modèle de la classe. Cela est implicite pour les déclarations et définitions écrites à l'intérieur du modèle de classe, mais doit être explicité pour les définitions écrites à l'extérieur.

Par exemple, voici notre modèle de classe `Table` avec des fonctions membres séparées :

```
template<class TElement> class Table {
    TElement *tab;
    int nbr;
public:
    explicit Table(int);
    ~Table();
    TElement &operator[](int);
    TElement operator[](int) const;
};
template<class TElement>
    Table<TElement>::Table(int n) {
        tab = new TElement[nbr = n];
    }
template<class TElement>
    Table<TElement>::~Table() {
        delete [] tab;
    }
template<class TElement>
    TElement &Table<TElement>::operator[](int i) {
        contrôle de la validité de i
        return tab[i];
    }
template<class TElement>
    TElement Table<TElement>::operator[](int i) const {
        contrôle de la validité de i
        return tab[i];
    }
```

[18] Dans les premières versions de C++ les modèles étaient définis comme des macros et pris en charge par le préprocesseur.

Les paramètres des modèles de classes peuvent avoir des valeurs par défaut. Voici, par exemple, une autre version de nos tables, dans laquelle l'allocation du tableau n'est pas dynamique ; par défaut, ce sont des tables de 10 entiers :

```
template<class TElement = int, int N = 10>
    class TableFixe {
        TElement tab[N];
    public:
        TElement & operator[](int i) {
            IndiceValide(i, N);
            return tab[i];
        }
        TElement operator[](int i) const {
            IndiceValide(i, N);
            return tab[i];
        }
    };

TableFixe<float, 10> t1;        // Oui, table de 10 float
TableFixe<float> t2;            // Oui, table de 100 float
TableFixe<> t3;                 // Oui, table de 100 int
TableFixe t4;                   // ERREUR
```

Comme une classe, un modèle peut être « annoncé », c'est-à-dire, déclaré et non défini. Il pourra alors être utilisé dans toute situation ne nécessitant pas de connaître sa taille ou ses détails internes :

```
template<class TElement> class Table;
Table<float> table1;            // ERREUR
Table<float> *ptr;              // Oui
ptr = new Table<float>;         // ERREUR
extern Table<float> table2;     // Oui
```

6 Exceptions

6.1 Principe et syntaxe

Il y a parfois un problème de communication au sein des grands logiciels. En effet, le concepteur d'une bibliothèque de bas niveau peut programmer la détection d'une situation anormale produite durant l'exécution d'une de ses fonctions, mais ne sait pas quelle conduite doit adopter alors la fonction de haut niveau qui l'a appelée. L'auteur des fonctions de haut niveau connaît la manière de réagir aux anomalies, mais ne peut pas les détecter.

Le mécanisme des exceptions est destiné à permettre aux fonctions profondes d'une bibliothèque de notifier la survenue d'une erreur aux fonctions hautes qui utilisent la bibliothèque.

Les points-clés de ce mécanisme sont les suivants :
- la fonction qui détecte un événement exceptionnel construit une exception et la « lance » (*throw*) vers la fonction qui l'a appelée ;
- l'exception est un nombre, une chaîne ou, mieux, un objet d'une classe spécialement définie dans ce but (souvent une classe dérivée de la classe `exception`) comportant diverses informations utiles à la caractérisation de l'événement à signaler ;
- une fois lancée, l'exception traverse la fonction qui l'a lancée, sa fonction appelante, la fonction appelante de la fonction appelante, etc., jusqu'à atteindre une fonction active (c.-à-d. une fonction commencée et non encore terminée) qui a prévu d'« attraper » (*catch*) ce type d'exception ;
- lors du lancement d'une exception, la fonction qui l'a lancée et les fonctions que l'exception traverse sont immédiatement terminées : les instructions qui restaient à exécuter dans chacune de ces fonctions sont abandonnées ; malgré son caractère prématuré, cette terminaison prend le temps de détruire les objets locaux de chacune des fonctions ainsi avortées ;
- si une exception arrive à traverser toutes les fonctions actives, car aucune de ces fonctions n'a prévu de l'attraper, alors elle produit la terminaison du programme.

Une fonction indique qu'elle s'intéresse aux exceptions qui peuvent survenir durant l'exécution d'une certaine séquence d'instructions par une expression qui d'une part délimite cette séquence et qui d'autre part associe un

traitement spécifique aux divers types d'exception que la fonction intercepte. Cela prend la forme d'un « bloc `try` » suivi d'un ensemble de gestionnaires d'interception ou « gestionnaires `catch` » :

```
try {
    instructions susceptibles de provoquer, soit directement soit dans des
    fonctions appelées, le lancement d'une exception
}
catch(déclarationParamètre1) {
    instructions pour traiter les exceptions correspondant au type de paramètre1
}
catch(déclarationParamètre2) {
    instructions pour traiter les exceptions, non attrapées par le
    gestionnaire précédent, correspondant au type de paramètre2
}
etc.
catch(...) {
    instructions pour traiter toutes les exceptions
    non attrapées par les gestionnaires précédents
}
```

Un bloc `try` doit être immédiatement suivi par au moins un gestionnaire `catch`. Un gestionnaire `catch` doit se trouver immédiatement après un bloc `try` ou immédiatement après un autre gestionnaire `catch`.

Chaque gestionnaire `catch` comporte un en-tête avec la déclaration d'un argument formel qui sera initialisé, à la manière d'un argument d'une fonction, avec l'exception ayant activé le gestionnaire. Exemple :

```
catch(bad_alloc e) {
    cerr << "ERREUR : " << e.what() << '\n';
}
```

Si le corps du gestionnaire ne référence pas l'exception en question, l'en-tête peut se limiter à spécifier le type de l'exception :

```
catch(bad_alloc) {
    cout << "ERREUR : Allocation impossible";
}
```

Un programme lance une exception par l'instruction

`throw` *expression*`;`

La valeur de *expression* est supposée décrire l'exception produite. Elle est propagée aux fonctions actives, jusqu'à trouver un gestionnaire `catch` dont le paramètre indique un type compatible avec celui de l'expression. Les fonctions sont explorées de la plus récemment appelée vers la plus anciennement appelée (c.-à-d. du sommet vers la base de la pile d'exécution) ; à l'intérieur d'une fonction, les gestionnaires catch sont explorés dans l'ordre où ils sont écrits. Le gestionnaire[19]

```
catch(...)
```

attrape *toutes* les exceptions. Il n'est donc pas toujours présent et, quand il l'est, il est le dernier de son groupe.

Dès l'entrée dans un gestionnaire `catch`, l'exception est considérée traitée (il n'y a plus de « balle en l'air »). A la fin de l'exécution d'un gestionnaire, le contrôle est passé à l'instruction qui suit le dernier gestionnaire de son groupe. Les instructions restant à exécuter dans la fonction qui a lancé l'exception et dans les fonctions entre celle-là et celle qui contient le gestionnaire qui a attrapé l'exception sont abandonnées. Il est important de noter que les objets locaux attachés aux blocs brusquement abandonnés à cause du lancement d'une exception sont proprement détruits.

Exemple. A plusieurs reprises nous avons vu des classes représentant des tableaux dans lesquels les indices étaieznt contrôles. Voici comment cela pourrait s'écrire[20] :

```
void IndiceValide(int i, int n) {
    if (i < 0 || i >= n)
        throw "indice hors bornes";
}
```

[19] Notez que, pour une fois, les trois points ... font partie de la syntaxe du langage.

[20] C'est une version naïve. Lancer, comme ici, une exception de type chaîne de caractères est vite fait, mais lancer une exception objet d'une classe, dérivée de la classe `exception`, expressément définie dans ce but, serait plus puissant et utile.

```
class Vecteur {
    double *tab;
    int nbr;
public:
    explicit Vecteur(int n) {
        tab = new double[nbr = n];
        }
    double &operator[](int i) {
        IndiceValide(i, nbr);
        return tab[i];
        }
    ...
};
```

Utilisation :

```
try {
    cout << "n ? "; cin >> n;
    Vecteur t(n);
    ...
    cout << "i, t[i] ? "; cin >> i >> x;
    t[i] = x;
    cout << "t[" << i << "] = " << x << '\n';
    ...
}
catch (char *message) {     // origine : IndiceValide
    cout << "Problème: " << message << '\n';
}
catch (bad_alloc) {         // origine : new
    cout << "Problème d'allocation de mémoire\n";
}
catch (...) {               // toutes les autres exceptions
    cout << "Problème indeterminé\n";
}
cout << " - Terminé\n";
```

Exécution :

```
i, t[i] ?  5  0.125
t[5] = 0.125
...
i, t[i] ?  25  0.250
Problème: indice hors bornes
Terminé
```

6.2 Attraper une exception

Lorsque l'exécution d'une instruction lance une exception, un gestionnaire `catch` ayant un argument *compatible* avec l'exception est recherché dans les fonctions actives (commencées et non encore terminées), de la plus récemment appelée vers la plus anciennement appelée.

Un argument d'un gestionnaire `catch` de type T_1, `const` T_1, T_1`&` ou `const` T_1`&` est compatible avec une exception de type T_2 si et seulement si :

- T_1 et T_2 sont le même type, ou
- T_1 est une classe de base accessible de T_2, ou
- T_1 et T_2 sont de la forme $B*$ et $D*$ respectivement, et B est une classe de base accessible de D.

Notez que les conversions standard sur des types qui ne sont pas des pointeurs ne sont pas effectuées. Ainsi, par exemple, un gestionnaire `catch(float)` n'attrape pas des exceptions `int`.

NOTE. Lorsque le contrôle entre dans un gestionnaire `catch`, son argument est initialisé (par l'exception effectivement lancée) selon un mécanisme analogue à celui de l'initialisation des arguments d'un appel de fonction.

C'est donc la même sorte de considérations que celles que l'on fait pour un appel de fonction qui permettent de choisir le mode (valeur, adresse ou référence) de l'argument du gestionnaire. On notera en particulier que

la déclaration comme pointeur ou référence est indispensable si on souhaite faire jouer le polymorphisme. Par exemple, le programme suivant est maladroit :

```
catch(exception e) {                    // maladroit
    cout << "Problème: " << e.what();
}
```

car on y appelle la fonction `what` de la classe `exception`, et l'affichage obtenu sera trop général, dans le genre : « `Problème : Unknown exception` ». En principe, l'exception effectivement lancée appartient à une classe dérivée de `exception`, dans laquelle la fonction virtuelle `what` a été redéfinie pour donner un message intéressant. Pour obtenir l'affichage de ce message il faut écrire :

```
catch(exception &e) {
    cout << "Exception: " << e.what();
}
```

ou bien (mais dans ce cas il faut que l'instruction `throw` lance l'exception par adresse) :

```
catch(exception *e) {
    cout << "Exception: " << e->what();
}
```

Relance d'une exception. A l'intérieur d'un gestionnaire `catch` on peut écrire l'instruction

```
throw;
```

elle indique que l'exception, qui était tenue pour traitée en entrant dans le gestionnaire, doit être relancée comme si elle n'avait pas encore été attrapée.

6.3 Déclaration des exceptions qu'une fonction laisse échapper

Lorsqu'une exception est lancée à l'intérieur d'une fonction dans laquelle elle n'est pas attrapée on dit que la fonction « laisse échapper » l'exception en question.

Une fonction peut indiquer les types des exceptions qu'elle laisse échapper. Cela définit avec précision, à l'intention des utilisateurs de la fonction, les risques entraînés par un appel de cette fonction. La syntaxe est :

en-tête_de_la_fonction `throw(`*type*$_1$`,` *type*$_2$`, ...` *type*$_k$`)`

Exemple :

```
void IndiceValide(int i, int n) throw(char *) {
    if (i < 0 || i >= n)
        throw "indice hors bornes";
}
class Vecteur {
    ...
    double &operator[](int i) throw(char *) {
        IndiceValide(i, nbr);
        return tab[i];
    }
    ....
};
```

Plus généralement, d'une fonction dont l'en-tête se présente ainsi

en-tête_de_la_fonction `throw(`T_1`,` T_2`*)`

ne peuvent provenir que des exceptions dont le type est T_1 ou un type ayant T_1 pour classe de base accessible ou T_2`*` ou un type pointeur vers une classe ayant T_2 pour classe de base accessible. L'expression

en-tête_de_la_fonction `throw()`

indique qu'un appel d'une telle fonction ne peut lancer *aucune* exception.

En outre, on considère qu'une fonction dont l'en-tête ne comporte pas de clause `throw` laisse échapper *toutes* les exceptions.

De telles déclarations ne font pas partie de la signature : deux fonctions dont les en-têtes ne différent qu'en cela ne sont pas assez différentes pour faire jouer le mécanisme de la surcharge.

Pour les exceptions explicitement lancées par la fonction, cette indication permet une vérification dès la compilation, avec l'émission d'éventuels messages de mise en garde.

Qu'il y ait eu une mise en garde durant la compilation ou pas, lorsqu'une exception non attrapée se produit dans une fonction qui ne la laisse pas échapper, la fonction prédéfinie `unexpected()` est appelée. Par défaut la fonction `unexpected()` se réduit à l'appel de la fonction `terminate()`.

6.4 La classe exception

Une exception est une valeur quelconque, d'un type prédéfini ou bien d'un type classe défini à cet effet. Cependant, les exceptions lancées par les fonctions de la bibliothèque standard sont toutes des objets de classes dérivées d'une classe définie spécialement à cet effet, la classe `exception`, qui comporte au minimum les membres suivants :

```
class exception {
public:
    exception() throw();
    exception(const exception &e) throw();
    exception &operator=(const exception &e) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

Les quatre premiers membres correspondent aux besoins internes de l'implémentation des exceptions. La fonction `what` renvoie une chaîne de caractères qui est une description informelle de l'exception.

On notera que, comme le montre la déclaration de cette classe, aucune exception ne peut être lancée durant l'exécution d'un membre de la classe `exception`.

Bien que la question concerne plus la bibliothèque standard que le langage lui-même, voici les principales classes dérivées directement ou indirectement de exception :

`exception`

- `bad_exception` : cette exception, en rapport avec la définition par l'utilisateur de la fonction `unexpected`, signale l'émission par une fonction d'une exception qui n'est pas déclarée dans sa clause `throw`.
- `bad_cast` : cette exception signale l'exécution d'une expression `dynamic_cast` invalide.
- `bad_typeid` : indique la présence d'un pointeur `p` nul dans une expression `typeid(*p)`
- `logic_error` : ces exceptions signalent des erreurs provenant de la structure logique interne du programme (en théorie, on aurait pu les prévoir en étudiant le programme) :
  - `domain_error` : erreur de domaine,
  - `invalid_argument` : argument invalide,
  - `length_error` : tentative de création d'un objet de taille supérieure à la taille maximum autorisée (dont le sens précis dépend du contexte),
  - `out_of_range` : arguments en dehors des bornes.
- `bad_alloc` : cette exception correspond à l'échec d'une allocation de mémoire.
- `runtime_error` : exceptions signalant des erreurs, autres que des erreurs d'allocation de mémoire, qui ne peuvent être détectées que durant l'exécution du programme :
  - `range_error` : erreur de rang,
  - `overflow_error` : débordement arithmétique (par le haut),
  - `underflow_error` : débordement arithmétique (par le bas).

Remarque. Les classes précédentes sont surtout des emplacements réservés pour des exceptions que les fonctions de la bibliothèque standard (actuelle ou future) peuvent éventuellement lancer ; rien ne dit que c'est actuellement le cas.

Références

B. Stroustrup
Le langage C++ (3ème édition)
CampusPress, 1999

H. Garreta
Le langage et la bibliothèque C++ Norme ISO
Ellipses, 2000

J. Charbonnel
Langage C++, la proposition de standard ANSI/ISO expliquée
Masson, 1996

A. Géron, F. Tawbi
Pour mieux développer avec C++
InterEditions, 1999

♠ ♠ ♠