LE LANGAGE C++

TODO :

- Flux d'entrées/sorties
- 5.1.4 : déclaration d'accès (à revoir)

v1.5.4.3 – 14/02/2014
peignotc(at)arqendra(dot)net / peignotc(at)gmail(dot)com

TABLE DES MATIÈRES

TABLE DES ANNEXES

TABLE DES ILLUSTRATIONS

0 PROLOGUE

Le contenu du présent document se focalise principalement sur les aspects de Programmation Orientée Objet appliqués au langage C++ et tient pour acquis tous les éléments de base du langage, qui ne sont ni plus ni moins que ceux issus du langage C (nda : voir le cours *Le langage C* du même auteur).

La maîtrise de notions telles que les principes de « mot-clef », d'« instruction », et de « variable », les structures de contrôle, les tableaux, les fonctions, les pointeurs et les types de variables complexes – principalement les structures – est impérative pour appréhender les nouveaux éléments amenés.

À défaut, la bonne connaissance des éléments d'un langage proche en termes de fondamentaux et de syntaxe, comme Java ou C# par exemple, doit permettre de saisir le contenu sans trop de difficultés ; dans certains cas, on veillera néanmoins à compulser des documents dédiés au langage C ou C++, car certaines notions demeurent spécifiques à ces langages (les pointeurs n'existent pas en Java et sont peu utilisés en C# par exemple).

1 INTRODUCTION AU LANGAGE C++

Le langage C++ est un langage de programmation orienté objet qui a été conçu au début des années 80 sur les bases du langage C [1]. L'idée était de rajouter l'approche POO [2] au langage C, langage ayant connu un franc succès, et de profiter ainsi de la popularité de ce dernier pour permettre un passage facilité à la programmation objet [3].

Le C++, bâti sur le C, reprend donc la quasi-intégralité des concepts que celui-ci met en œuvre, le même jeu d'instructions de base, la même syntaxe, ainsi que l'accès aux librairies standard du C. Le C++ peut ainsi être considéré comme un sur-ensemble du C [4].

Outre les aspects de POO, le C++ permet aussi, entre autres, une gestion plus efficace des affichages et saisies, apporte de nouveaux opérateurs, et plus de souplesse dans la déclaration des variables, tout en étant un langage plus fortement typé que le C [5].

Néanmoins, le C++ souffre des défauts de sa conception, et parce qu'il a été le premier langage orienté objet largement utilisé, tout comme le fait qu'il soit basé sur un langage qui lui ne l'est pas, reste un langage laborieux, parfois confus, qui demande une très bonne maîtrise du C et des concepts objets.

Les avantages du langage C++ sont :

- continuité du langage C ;
- possibilité de travailler en objet (C++) ou en séquentiel (C) ;
- portable (recompilation nécessaire tout de même) ;
- rapidité d'exécution ;
- largement diffusé et utilisé.

Les inconvénients du langage C++ sont :

- plus complexe que d'autres langages objet ;
- confusion possible avec le langage C.

Le langage C++ est le langage objet « historique » et reste une référence incontournable de la programmation dans le milieu industriel. Par ailleurs, nonobstant ses lourdeurs, l'étude du C++ est une bonne passerelle pour la programmation objet avec des langages de plus haut niveau ; et par rapport auxquels l'exécution d'un programme écrit en C++ est souvent plus rapide.

---

[1] Le concepteur du langage C++ est Bjarne Stroustrup.
[2] Programmation Orientée Objet.
[3] Le premier nom du langage C++ était ainsi *C with classes* (« C avec des classes »), la « classe » étant l'élément sémantique fondamental en programmation objet.
[4] On peut dire que le *C++ est un C « incrémenté »* (++).
[5] Vérification plus rigoureuse du type d'une variable.

2 NOTIONS DE BASE

2.1 INTRODUCTION

Le langage C++ est donc pleinement basé sur le langage C, et est identique sur tous les points importants :

- développement : langage compilé, chaîne de compilation, sources sous forme de fichiers texte, structure des fichiers source (l'extension n'est plus *.c* bien sûr mais *.cpp*) ;
- commentaires : multi-lignes et mono-ligne ;
- variables : types, manipulation et opérateurs ;
- entrées/sorties : affichages et saisies ;
- structures de contrôles : opérateurs, structures alternatives et itératives ;
- tableaux et chaînes de caractères : principes et manipulation ;
- fonctions : librairies standard du C, fonctions utilisateur ;
- pointeurs : principes, manipulation et application ;
- types de variables complexes : structures, etc. ;
- entrées/sorties sur fichiers : lecture et écriture ;
- …

Néanmoins, mis à part les aspects de programmation objet, C++ propose aussi certaines améliorations et spécificités par rapport au C.

2.2 AMÉLIORATIONS PAR RAPPORT AU LANGAGE C

2.2.1 Entrées/sorties

2.2.1.1 Entrées/sorties standard

L'interfaçage entre une application et le système d'exploitation, qui est une entité externe à l'application, met en œuvre des flux d'entrées/sorties [1]. Comme ces flux sont les plus utilisés (quasiment constamment lors de l'utilisation de la machine), on les appelle **flux standard** [2] ou flux d'entrées/sorties standard :

- entrée standard : par défaut le clavier, appelé *stdin* (fonctions `scanf()`) ;
- sortie standard : par défaut la console (l'écran), appelé *stdout* (fonctions `printf()`) ;
- erreur standard : par défaut est aussi la console, appelé *stderr*.

2.2.1.2 Entrées/sorties standard par flux

Pour effectuer des entrées/sorties standard, on peut utiliser les fonctions `scanf()` et `printf()` de la librairie standard du C, mais aussi les **entrées/sorties standard par flux** de la bibliothèque *iostream.h* :

- entrée standard : flux `cin` ;
- sortie standard : flux `cout` ;
- erreur standard : flux `cerr` ;
- erreur standard bufferisée : flux `clog`.

On manipule ces flux à l'aide d'opérateurs spécifiques :

- lecture de valeur d'un flux d'entrée : opérateur >> (macro pour la fonction `istream& operator<<()`) ;
- écriture de valeur dans un flux de sortie : opérateur << (macro pour la fonction `ostream& operator>>()`).

[1] Un flux est un moyen de communication sériel : les données sont traitées dans l'ordre dans lequel elles sont envoyées.
[2] Les flux standard sont énormément utilisés dans le monde Unix, au niveau du shell.

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

int main(void)
{
   int i;
   float f = 1.23;
   char ch[20] = "bonjour\n";

   cout << "saisir un entier ";
   cin >> i;     // saisie d'une valeur et stockage dans la variable i
   cout << "i=" << i << "\nf=" << f << "\nch=" << ch;   // affichage de valeurs
   return 0;
}
```

L'utilisation des entrées/sorties standard par flux est plus rapide [1], et la taille mémoire créée à la compilation est réduite [2] ; de plus, le type est vérifié automatiquement, limitant ainsi les risques d'affichage erroné.

On peut modifier la manière dont les données sont lues ou écrite dans le flux, en écrivant un ou plusieurs manipulateurs dans le flux, définis dans la bibliothèque *iomanip.h* :

- `dec` : lecture/écriture d'un entier en décimal ;
- `oct` : lecture/écriture d'un entier en octal ;
- `hex` : lecture/écriture d'un entier en hexadécimal ;
- `endl` : insertion d'un saut de ligne et vidage des buffers ;
- `setprecision(int)` : affichage de la valeur avec un nombre de chiffres précisé (virgule décimale non incluse) [3] ;
- `setw(int)` : affichage formaté d'une longueur du nombre de caractères précisé [3] (remplissage avec le caractère défini par `setfill()`, ou bien par défaut par le caractère espace) ;
- `setfill(char)` : définition du caractère de remplissage utilisé avec `setw()` [3] ;
- `flush` : vidage des buffers après écriture.

Ex. :
```
#include <iostream.h>
#include <iomanip.h>

int main(void)
{
   int i = 1234;
   float f = 54.6553;
   cout << setw(10) << setfill('*') << i << endl;    // affichage: "******1234"
   cout << setw(8) << setprecision(5) << f << endl;  // affichage: "**54.655"
   return 0;
}
```

2.2.2 Variables

2.2.2.1 Déclaration

La **déclaration** d'une variable peut être réalisée n'importe où dans le code, et plus seulement en début de bloc. La portée reste identique, c'est-à-dire qu'elle reste définie dans le bloc de code de déclaration.

Ceci est notamment utile pour les compteurs de boucle, afin d'éviter que leur déclaration ne perturbe la lisibilité.

Ex. :
```
int i;
cin >> i;
int k = 5;
for ( int j=0 ; j<5 ; j++ ) cout << i+j << " "; // j n'existe que dans le for ()
```

2.2.2.2 Type booléen

Le C++ inclut un nouveau type de variables, le **type booléen**, défini par le mot-clef `bool`. Ce type ne peut donc prendre que 2 valeurs : `true` ou `false`.

[1] L'analyse est réalisée à la compilation, au lieu d'être faite à l'exécution.
[2] Seul le code nécessaire est compilé, au lieu d'inclure le code nécessaire à tous les formats disponibles.
[3] La définition d'une valeur avec ce manipulateur reste valide jusqu'à la fin du programme ou bien jusqu'à la définition suivante.

Ce type est utilisé partout où, en langage C, était utilisée une variable entière de valeur 0 ou 1, comme :

- le résultat d'une opération logique (`&&`, `||`, etc.) ;
- l'évaluation d'une expression (tests, etc.) [1].

Ex. :
```
bool flag // définition d'une variable booléenne
if ( flag ) { ... }     // identique à if ( flag == true )
if ( !flag ) { ... }    // identique à if ( flag == false )
```

2.2.2.3 Constantes

On peut déclarer une **variable constante**, sans utiliser la directive préprocesseur `#define`, avec le modificateur `const` à utiliser suivant la syntaxe `const type_variable nom_variable = valeur`.

Ex. :
```
const int i = 2;    // initialisation obligatoire de la valeur pour une constante
```

Nb : Définir un *pointeur constant* (`type* const ptr`) et un *pointeur sur une constante* (`const type* ptr`) sont deux choses différentes. Dans le premier cas, l'adresse ne peut pas être modifiée (le pointeur n'est pas « navigable ») ; dans le second, la valeur pointée ne peut pas être modifiée. Il est d'ailleurs possible de définir *un pointeur constant sur une constante* (`const type* const ptr`) ; ni l'adresse ni la valeur ne peuvent être modifiées.

2.2.2.4 Structures

Lors de la définition d'une **structure**, il n'est plus nécessaire de créer un type synonyme (avec le mot-clef `typedef`) pour ainsi s'affranchir de la syntaxe `struct nom_structure`. Cette définition alloue automatiquement le nom donné au nouveau type.

Ex. :
```
struct Personne {   // définition d'un nouveau type utilisateur
   char initiale;
   int age;
};

int main(void)
{
   Personne eleve1;  // utilisation du type utilisateur
   ...
}
```

2.2.2.5 Énumérations

Tout comme les structures, lors de la définition d'une **énumération**, il n'est plus nécessaire de créer un type synonyme (avec le mot-clef `typedef`) pour s'affranchir de la syntaxe `enum nom_enumeration`. Cette définition alloue automatiquement le nom donné au nouveau type.

Ex. :
```
enum semaine {lun, mar, mer, jeu, ven, sam, dim};
semaine jour;
jour = mar;
```

Au contrario du langage C, une énumération n'est pas une liste d'entiers, et une variable énumérée ne peut prendre pour valeur que l'une des valeurs du type énuméré auquel elle appartient. Il est malgré tout possible de lire une interprétation de la valeur entière de la variable énumérée.

Ex. :
```
int i = sam;        // conversion implicite en int, i vaut 5
cout << jour << endl;   // affichage: "1" (mar)
```

2.2.2.6 Conversion de type

La **conversion de type** (transtypage) peut être réalisée syntaxiquement de 2 manières :

- `(nouveau_type) expression_ou_variable` ;
- `nouveau_type (expression_ou_variable)`.

Ex. :
```
double d = 2.42;
int i, j;
i = (int) d;    // transtypage classique
j = int (d);
```

[1] Avec une valeur entière, il est toujours possible d'écrire `if ( x ) { ... }` mais il est vivement conseillé d'écrire explicitement `if ( x != 0 ) { ... }`.

Nb : La conversion explicite de type est souvent utilisée avec les pointeurs du type `void*`.

2.2.2.7 Opérateur de résolution de portée

On peut accéder à une variable déclarée en dehors du bloc de code courant en utilisant l'**opérateur de résolution de portée** `::` suivant la syntaxe `::nomVariable`. Cet opérateur permet généralement d'accéder ainsi à une variable globale possédant un homonyme en local.

Ex. :
```
int i = 4;

int main(void)
{
   int i = 10;

   cout << i << endl;      // affichage: "10"
   cout << ::i << endl;    // affichage: "4"
   return 0;
}
```

Nb : Malgré l'existence de cet opérateur, et comme toujours, il est conseillé d'utiliser des noms de variables différents pour des variables différentes, et donc il ne faut l'utiliser qu'en cas de réel besoin.

2.2.2.8 Références

Avec les pointeurs on manipule des adresses de manière explicite ; on peut aussi manipuler des adresses de manière implicite, en laissant l'ordinateur géré le principe contenant/contenu. En C++, on dispose pour cela du type **référence** noté `type&`, à utiliser suivant la syntaxe `type &reference`.

Ainsi, hormis lors de l'initialisation, on manipule un *contenu* et pas un contenant (à l'instar des pointeurs). Ce qui implique qu'une variable référence doit être initialisée à la déclaration, et que cette initialisation correspond le plus souvent à une variable déjà déclarée ; les 2 variables utilisent alors le même espace mémoire.

Ce type permet notamment de créer une variable « synonyme » d'une autre : la modification de l'une d'elles modifie automatiquement l'autre.

Ex. :
```
int i = 6;
int& j = i;     // définition d'une variable référence

i = 3;
cout << j << endl;   // affichage: "3"
j = 11;
cout << i << endl;   // affichage: "11"
```

2.2.2.9 Allocation de mémoire

Venant supplanter `malloc()` et `free()`, on dispose de deux nouveaux opérateurs afin de réserver et de libérer la mémoire. Il s'agit des opérateurs `new` et `delete`, à utiliser suivant les syntaxes `pointeur = new type_pointeur` / `delete pointeur` pour une variable, ou `pointeur = new type_pointeur[...]` / `delete[] pointeur` lorsque l'on déclare plusieurs espaces mémoires contigus.

Ex. :
```
int* ptr;
float* ptr2;
ptr = new int;     // réservation de l'espace mémoire pour 1 int
ptr2 = new float[5];    // réservation de l'espace mémoire pour 5 float
...
delete ptr;        // libération de l'espace mémoire réservé pour 1 int
delete[] ptr2;        // libération de l'espace mémoire réservé pour 5 float
```

Tout comme `malloc()`, l'opérateur `new` retourne l'adresse de début du pointeur.

Nb : Les opérateurs `new` et `delete` servent également pour la gestion de l'espace mémoire nécessaire aux objets.

2.2.3 Fonctions

2.2.3.1 Paramètres par défaut

Pour une fonction, il est possible de définir des **paramètres par défaut** ; c'est-à-dire, spécifier une valeur par défaut pour un ou plusieurs paramètres d'entrée. Il suffit pour cela d'inclure, dans le prototype de la fonction, l'initialisation du ou des paramètres à leur valeur respective par défaut.

Pour affecter au paramètre sa valeur par défaut, il suffit de l'omettre dans l'appel de la fonction ; sinon c'est la valeur passée en paramètre qui est utilisée. L'appel de la procédure peut alors être réalisée de plusieurs manières.

Ex. :
```
void AfficherValeurs(int val1, int val2 = 5)
{
   cout << val1 << " " << val2 << endl;
}

int main(void)
{
   AfficherValeurs(11, 3);    // affichage: "11 3"
   AfficherValeurs(11);       // affichage: "11 5"
   return 0;
}
```

Les paramètres ayant une valeur par défaut doivent être positionnés à la fin de la liste des paramètres du prototype.

2.2.3.2 Fonctions inline

Pour une fonction de petite « taille », devant être appelée souvent, on peut définir une pseudo-fonction (/macro), appelée **fonction inline**, afin d'optimiser son exécution. Pour cela, on utilise le modificateur `inline` dans la définition du prototype de la fonction.

Ex. :
```
inline int Carre(int n)
{
   return n*n;
}

int main(void)
{
   cout << Carre(3) << endl;       // identique à cout << 3*3 << endl;
   return 0;
}
```

Une fonction inline n'est pas compilée : le préprocesseur remplace l'appel de la fonction par son code source avant la compilation, évitant ainsi le branchement et la mise en pile réalisés normalement lors de l'appel d'une procédure, d'où le gain d'exécution.

Nb : Lors de la compilation, le compilateur a toute liberté de décider si effectivement les fonctions déclarées `inline` seront traitées comme telles, ou bien comme des fonctions classiques.

2.2.3.3 Utilisation des références

En plus du passage par valeur, et par adresse (pointeur), on dispose en C++ du **passage par référence**. L'intérêt est de pouvoir travailler dans une fonction avec exactement la même variable que celle passée en paramètre, en utilisant pour cela une référence qui crée une variable occupant le même espace mémoire.

Ex. : Une fonction effectuant la permutation de deux nombres passés en paramètres.

```
void Permut(int& nb1, int& nb2)
{
   int tmp = nb1;
   nb1 = nb2;
   nb2 = tmp;
}
```

```
int main(void)
{
   int i = 2, j = 9;
   Permut(i, j);
   cout << i << " - " << j << endl;     // affichage: "9 - 2"
   return 0;
}
```

Nb : Si le passage par référence est choisi (fortement conseillé pour les paramètres de grande taille comme les objets), mais que l'on désire éviter toute modification, on précise alors le modificateur `const` pour le paramètre passé par référence dans la définition du prototype.

Une fonction peut retourner une **référence à la valeur de retour**, plutôt que la valeur elle-même. L'intérêt est qu'il est alors possible, à l'extérieur de la fonction, de modifier cette valeur de retour [1].

Ex. : `int t[5];`

```
int& Nieme(int i)
{
   return t[i];
}

int main(void)
{
   Nieme(0) = 12;
   Nieme(1) = 4;
   cout << t[0] << " - " << t[1] << endl;    // affichage: "12 - 4"
   return 0;
}
```

2.2.3.4 Surcharge

La **surcharge** d'une fonction correspond à la définition de fonctions homonymes mais dont la signature est différente ; la signature d'une fonction est caractérisée par le nom, le nombre et le type des paramètres [2]. L'intérêt est de pouvoir utiliser le même nom de fonction, avec des paramètres différents, pour définir des comportements de principe similaire mais pas identiques en tout point.

Ex. : Une fonction effectuant la somme de plusieurs nombres passés en paramètres.

```
int Somme(int i1, int i2)
{
   return i1 + i2;
}

int Somme(int i1, int i2, int i3)
{
   return i1 + i2 + i3;
}

double Somme(double d1, double d2)
{
   return d1 + d2;
}
```

[1] Apparaîssent ici les notions d'entités de type LValue (Left Value (eng) ≡ Valeur de Gauche (fr)) et RValue (Right Value (eng) ≡ Valeur de Droite (fr)) qui précisent si ce qui représente une valeur (variable ou fonction) peut être positionné à gauche ou à droite du signe '=' dans une expression d'affectation. Ainsi, une variable peut être LValue (ex. : `var = 1;`) ou RValue (ex. : `autrevar = var;`) ; une fonction retournant une valeur ou un pointeur peut être RValue (ex. : `x = fonction();`), mais ne peut en aucune manière être LValue (ex. : `int fonction(); fonction() = 1; // erreur d'exécution`) ; en revanche, une fonction retournant une référence peut être RValue, mais aussi LValue (ex. : `int& fonction(); fonction() = 1; // pas d'erreur d'exécution`).

[2] Rappels : le prototype d'une fonction est caractérisée par le nom, le nombre et le type des paramètres, ainsi que le type de valeur de retour ; donc, prototype ≡ signature + type de valeur de retour.

```
int main(void)
{
   int i = 2, j = 4, k = 8;
   double x = 4.32, y = -51.3;

   cout << Somme(i, j) << endl;
   cout << Somme(x, y) << endl;
   cout << Somme(i, j, k) << endl;
   return 0;
}
```

C'est le compilateur qui se charge de sélectionner la fonction à exécuter en comparant les signatures de la fonction appelée et des différentes fonctions surchargées existantes.

---

3 LES OBJETS ET LES CLASSES

3.1 LA PROGRAMMATION ORIENTÉE OBJET

3.1.1 L'approche objet

La programmation procédurale, dite aussi fonctionnelle [1], se base sur l'analyse descendante des fonctions attendues du système avec un processus récursif de découpage en sous-fonctions de plus en plus faciles à appréhender, et ce, jusqu'à atteindre un niveau de codage réalisable. L'analyse est donc uniquement axée sur *ce que fait le système*.

L'**approche objet** consiste à considérer le système comme un ensemble d'éléments organisés qui interagissent afin de constituer un tout : le système est fait d'objets qui collaborent par l'intermédiaire de messages. L'analyse est axée sur *ce qu'est le système* et *comment est organisé le système*, en plus de *ce que fait le système*. Ce type d'analyse, appelée aussi approche systémique, a pour but de représenter au mieux le monde réel dans lequel évoluent les systèmes informatiques et répondre ainsi plus aisément aux besoins.

La Programmation Orientée Objet, qui met en œuvre l'approche objet, date des années 60, et de nombreux langages objets ou orientés objet ont été créés [2] afin de répondre au mieux à ses différents principes. Elle est en plein essor depuis une quinzaine d'années, et est devenue incontournable dans l'industrie de développement de produits logiciels, pour les nombreux avantages qu'elle propose.

3.1.2 Atouts

L'approche objet, de par ses principes, s'insère dans une démarche de *qualité*, dans le but de fournir des méthodes et outils pour créer des produits logiciels plus performants suivant différents axes : maniabilité, maintenabilité, testabilité, lisibilité, modularité, interopérabilité, portabilité, robustesse, extensibilité, évolutivité, réutilisabilité.

Elle est adaptée à quasiment tous les champs d'investigation de la programmation, notamment car elle propose une technologie plus proche de la réalité, ainsi qu'une méthodologie accrûe ; en revanche, elle demande un effort supplémentaire d'abstraction et de conceptualisation.

3.2 L'OBJET

3.2.1 Introduction : rappels sur les structures

Une **structure** est un type de données *utilisateur* qui permet de regrouper des variables de différents types en un seul ensemble. Chaque variable au sein de la structure, appelée *champ*, est accessible indépendamment des autres.

Une structure définit un nouveau type, c'est-à-dire un modèle de variables, utilisable comme les types de variables « classiques ». Ce modèle de variables permet ainsi de créer des variables possédant les champs définis (nom et type).

Ex. : Soit une structure, constituée de 2 champs, représentant une personne, nommée `Personne`.

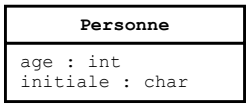

Figure 3.1 : exemple de structure

[1] Langage C, par exemple.

[2] Sans ordre particulier : C++, Java, SmallTalk (l'un des tout premiers), ADA, PHP5, Delphi, C#, …

Lorsque l'on fait référence à l'un des champs d'une structure, il faut obligatoirement préfixer le nom du champ par le nom de la structure à laquelle il appartient. Pour cela, on utilise l'opérateur '`.`' (point) ou '`->`' (tiret + supérieur), suivant la syntaxe `variable_structuree.champ` / `variable_structuree->champ`.

Une fois déclarée, une variable structurée peut être utilisée de diverses manières :

- On utilise l'un des champs de la structure : il se manipule comme une variable classique, est du type défini lors de la définition de la structure, et doit être préfixé du nom de la variable structurée.

  Ex. : Soit `eleve` une variable structurée de type `Personne` ; pour accéder à chacun des champs `initiale` et `age`, on écrira respectivement `eleve.initiale` et `eleve.age`.

- On utilise la variable structurée dans son ensemble (ce qui sous-entend aussi la totalité des champs de la structure) : elle se manipule comme une variable classique, est du type du nom de la structure, et peut être passée comme paramètre d'entrée d'une fonction ou comme type de retour d'une fonction ; tous les champs et leur valeur sont ainsi « inclus ».

  Ex. : Soit `eleve` une variable structurée de type `Personne` ; pour manipuler la variable structurée et donc ainsi l'ensemble des champs, on écrit simplement `eleve`.

3.2.2 Définitions

Selon le dictionnaire en ligne *Mediadico* [1], la définition du mot *objet* est la suivante :
Objet (n.m.) *i. Ce qui affecte les sens.*
ii. Ce qui se présente à l'esprit.
iii. Chose dans un sens indéterminé.

En informatique, le terme **objet** est tout aussi large, et désigne une entité informatique qui représente de manière abstraite une entité concrète ou abstraite du monde réel ou virtuel [2], dans le but de la piloter ou de la simuler. Cette représentation abstraite est une image simplifiée de l'objet originel effectivement représenté, en ne conservant que les composantes intrinsèques nécessaires à une problématique donnée [3].

Un objet peut ainsi être défini par trois caractéristiques :

- un *état* : *ce qu'est l'objet*, ensemble des valeurs instantanées de toutes les informations qualifiant l'objet (appelées **attributs**) ; c'est l'aspect statique ;
- un *comportement* : *ce que peut faire l'objet*, ensemble des compétences de l'objet ainsi que de ses actions et réactions (appelées **méthodes**) ; c'est l'aspect dynamique ;
- une *identité* : concept caractérisant l'existence propre d'un objet hors de toute considération d'*état* et de *comportement*.

On pourrait ainsi écrire : *objet* ≡ *état* + *comportement* + *identité*, en insistant sur le fait que l'objet ainsi défini est plus que la somme de ses 3 caractéristiques [4].

Ex. : Soit un objet représentant un véhicule du monde réel.
L'*état* du véhicule peut être défini par diverses informations : marque, modèle, couleur, …
Le *comportement* du véhicule peut être défini par diverses compétences et actions/réactions : démarrer, arrêter, accélérer, freiner, …
L'*identité* du véhicule permet de faire la distinction entre 2 véhicules de mêmes marque, modèle et couleur mais qui ont pourtant 2 existences propres (même s'ils sont pareils en tout point, chacun d'eux est unique ; par exemple, si l'un est abîmé lors d'un accident, cela n'aura aucune répercussion sur l'autre).

3.2.3 Principes de « classe » et d'« instance de classe »

Tout objet appartient à une famille d'objets, appelée classe. Une **classe** symbolise un ensemble d'objets ayant un *état* similaire (mêmes informations, mais pas forcément de même valeur) et un *comportement* identique.
Ce terme désigne donc un modèle d'objets, tel un type de variables classique ou bien une structure, qui permet de créer des objets d'*identité* différente, mais d'*état* similaire, et de *comportement* identique.

[1] http://www.mediadico.com.
[2] La notion d'« objet informatique » est donc aussi vague que la notion usuelle d'« objet ».
[3] La représentation informatique d'un même objet (au sens usuel du terme) peut ainsi être très différente d'une application à une autre.
[4] À l'image de la maxime « le tout vaut plus que la somme des parties ».

L'*état* et le *comportement* de tout objet sont donc basés sur ceux de la classe dont il provient : l'objet est une **instance** d'une classe, i.e. une copie fabriquée sur le modèle d'une classe, avec une *identité* propre, un *état* de même nature mais avec ses valeurs spécifiques, et un *comportement* commun à tous les objets d'une classe.

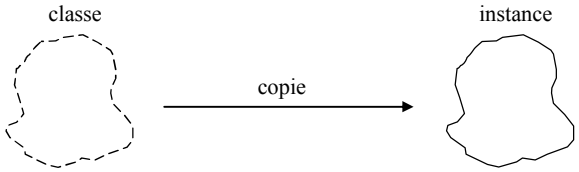

Figure 3.2 : classe et instance de classe

L'objet (ou instance) est manipulable alors que la classe reste un modèle immuable. De fait, un objet peut être créé, utilisé et détruit, alors que la classe ne peut être qu'utilisée comme modèle.

Pour représenter une classe, on utilise généralement une représentation type UML sous forme de cadre avec plusieurs séparateurs pour préciser : nom de la classe, liste des attributs (état), liste des méthodes (comportement).

Figure 3.3 : représentation UML d'une classe

Ex. : Soit la classe `Vehicule` représentant la notion de véhicule du monde réel.

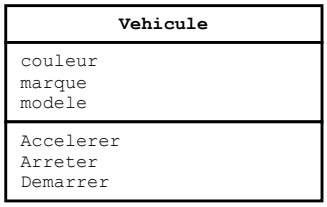

Figure 3.4 : exemple d'une classe

Sur ce modèle, pourront être créés divers objets `Vehicule` : une voiture de couleur bleue de marque SuperCar et de modèle Sonic, un vélo de couleur rouge de marque KoolBÿk et de modèle Mud, un camion de couleur noire de marque BigTruck et de modèle 40Tons, etc. ; ces véhicules peuvent accélérer, démarrer, et s'arrêter ; ces véhicules sont différents l'un de l'autre. Par conséquent, ils ont chacun leur *identité* propre, un *état* parfaitement défini, et un *comportement* identique.

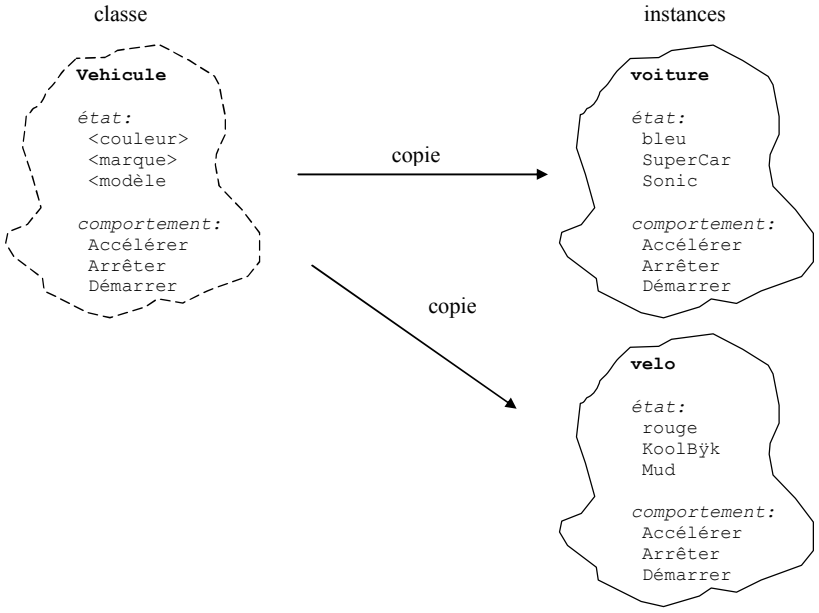

Figure 3.5 : exemple de création d'instances différentes à partir d'une même classe

En passant à la représentation informatique d'une famille d'objets, les attributs d'une classe (*état*) sont représentés par des variables, et les méthodes d'une classe (*comportement*) par des fonctions. Le terme « classe » désigne donc une entité informatique spécifique regroupant des variables associées et des fonctions associées en un seul ensemble ; le terme de **membres** désigne l'ensemble des attributs et des méthodes de cette classe.

Ex. : Soit la classe `Vehicule` représentant la notion de véhicule du monde réel.

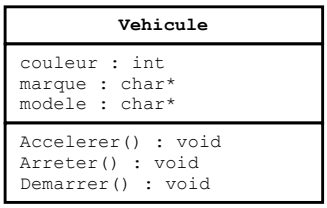

Figure 3.6 : exemple d'une classe avec détail des attributs et méthodes

Par la suite, l'accès aux divers membres d'une instance d'une classe s'opère de manière identique aux structures :

- opérateur `.` (point) pour une instanciation « classique » (statique) (Ex. : `monVehicule.couleur` et `monVehicule.Demarrer()`) ;
- opérateur `->` (tiret + supérieur) pour une instanciation à l'aide d'un pointeur (dynamique) (Ex. : `monVehicule->couleur` et `monVehicule->Demarrer()`).

Il est à noter qu'un membre n'a de sens que lorsqu'il est associé à un objet de la classe à laquelle il appartient. Par conséquent, deux membres ayant même nom mais appartenant à deux classes distinctes n'ont aucun lien entre eux [1].

3.2.4 Principe d'« encapsulation »

L'approche objet amène de la cohérence et de la précision dans la définition des modèles [2] utilisés. La cohésion interne d'un objet est notamment très forte, et doit être protégée de l'extérieur ; cependant, l'objet doit proposer un ensemble de services afin d'interagir avec son environnement : c'est ce qu'on appelle l'**encapsulation** [3].

Il convient donc de préserver la valeur des attributs d'un objet, en leur donnant le caractère (littéralement) *privé*, ce qui interdit tout accès de l'extérieur de la classe, que ce soit en lecture ou en écriture.

En revanche, il faut permettre à l'objet de collaborer avec d'autres objets, et on donne alors aux méthodes le caractère *public*, ce qui autorise leur appel de l'intérieur comme de l'extérieur de la classe.

S'il est nécessaire de permettre l'accès aux attributs de l'extérieur de la classe, il faut donc utiliser les méthodes de la classe, qui assureront ainsi des accès contrôlés par la classe elle-même.

Ce caractère public/privé, appelé **visibilité** [4] ou *droit d'accès*, peut être adapté si besoin ; néanmoins, par défaut, afin de respecter l'encapsulation, les attributs doivent donc être privés, et les méthodes publiques.

Dans la représentation de la classe (représentation UML), un caractère public est symbolisé par l'ajout d'un + (plus) devant le nom du membre, alors qu'un caractère privé est symbolisé par l'ajout d'un − (moins).

Ex. : Soit la classe `Vehicule` représentant la notion de véhicule du monde réel.

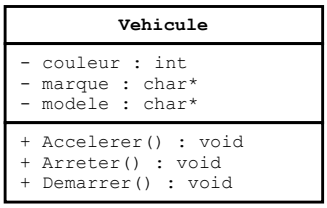

Figure 3.7 : exemple d'une classe avec détail des attributs et méthodes, et de leur visibilité

L'ensemble des membres publics est appelé *interface* de la classe, car vu de l'extérieur, ils en représentent la partie émergée, alors que la partie immergée (les membres non publics [5]) reste totalement inconnue.

Nb : Les membres non-publics ne sont protégés que de l'extérieur de la classe ; cette protection n'a pas cours pour deux objets distincts de la même classe. On parle d'*encapsulation de classe* et non d'*encapsulation d'objet*.

[1] Exemple : La méthode `Demarrer()` de la classe `Vehicule` fait tout autre chose que la méthode `Demarrer()` de la classe `Ordinateur`.
[2] Les classes.
[3] L'encapsulation est avec l'*héritage* et le *polymorphisme* l'un des principes définissant la Programmation Orientée Objet (cf. annexe D).
[4] Il existe un troisième caractère de visibilité, *protégé*, qui définit un accès restreint à la classe et aux classes héritées (cf. 5.1.1).
[5] Membres privés et protégés.

3.3 ÉCRITURE D'UNE CLASSE

3.3.1 Déclaration d'une classe

Avant d'utiliser un objet, il faut donner la déclaration de la classe à laquelle appartient cet objet. Cette déclaration correspond à la description de l'état et du comportement de la classe :

- *état* : nom, type, et visibilité de chaque attribut ;
- *comportement* : prototype et visibilité de chaque méthode.

On déclare alors la classe en utilisant le mot-clef `class` en regroupant ses différents membres par visibilité [1] (membres publics : `public`, membres privés : `private`), suivant la syntaxe :

```
class NomClasse // nom de la classe débute par une majuscule
{
   public:
      type_méthode1 NomMethode1(type_paramètre1-1, type_paramètre1-2, ...);
      type_méthode2 NomMethode2(type_paramètre2-1, type_paramètre2-2, ...);
      ...
   private:
      type_attribut1 nomAttribut1;
      type_attribut2 nomAttribut2;
      ...
}; // attention à ne pas oublier le ; final !
```

Nb : Si un membre est déclaré hors d'une catégorie de visibilité (au tout début), il est considéré comme étant privé.

Ex. : Soit une classe représentant une personne.

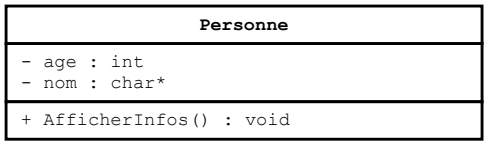

Figure 3.8 : exemple de déclaration d'une classe

```
class Personne
{
   public:
      void AfficherInfos();
   private:
      int age;
      char* nom;
};
```

Nb : La déclaration d'une classe est généralement réalisée dans un fichier d'en-tête, portant l'extension *.h*, dont le nom est le nom exact de la classe (Ex. : *Personne.h*).

3.3.2 Implémentation des méthodes

Les méthodes peuvent être définies dans la déclaration de la classe elle-même [2], telles de simples fonctions, mais en règle générale leur **implémentation** (écriture du code de la méthode) est donnée à l'extérieur. Pour savoir que la méthode appartient à la définition d'une classe, et à quelle classe [3], on préfixe alors le nom de la méthode avec le nom de la classe, suivi de l'opérateur de résolution de portée `::` (2x deux-points), suivant la syntaxe `type_méthode NomClasse::NomMethode(type_paramètre1 nomParametre1, ...)`.

Ex. : La méthode `AfficherInfos()` de la classe `Personne`.

```
void Personne::AfficherInfos()  // nom d'une méthode débute par une majuscule
{
   ...
}
```

[1] On ordonne généralement les catégories de visibilité en commençant par la catégorie 'public'.
[2] Elles sont alors considérées par le compilateur comme des fonctions inline (cf. 2.2.3.2).
[3] Et n'est donc pas une simple fonction.

Le corps de définition d'une méthode appartient à la classe. En conséquence, on accède librement à n'importe quel membre de la classe, même les membres privés ; de plus, il est inutile de préfixer leur nom.

Ex. : Les méthodes de la classe `Personne`.

```
void Personne::AfficherInfos()
{
   cout << "age " << age << endl;
   cout << "nom " << nom << endl;
}
```

Nb : L'implémentation des méthodes membres est généralement réalisée dans un fichier séparé du fichier d'en-tête, portant l'extension *.cpp*, dont le nom est le nom exact de la classe (Ex. : *Personne.cpp*).

3.3.3 Principe de « constructeur »

Un objet est l'instance d'une classe, c'est-à-dire une entité créée sur le modèle d'une classe, qui possède une existence propre (*identité*) et un *état* défini, en sus du *comportement* commun à tous les objets de la classe.

Lors de la déclaration d'un objet (`NomClasse nomObjet;`), l'identité est créée (`nomObjet` est fabriqué sur le modèle `NomClasse`), et les méthodes sont pleinement définies (reprises directement de la définition de la classe).
En revanche, même si les attributs sont définis, ils ne sont pas initialisés et ne possèdent pas de valeurs spécifiques à l'objet représenté. Il faut donc mettre en place un mécanisme d'initialisation des attributs.

Pour réaliser cela, on va utiliser une méthode, appelée **constructeur**, qui a en charge d'initialiser tous les attributs de l'objet. Cette méthode spécifique a pour particularités :

- nom exactement identique au nom de la classe (`NomClasse(...)`) ;
- méthode publique (pour pouvoir être appelée de l'extérieur de la classe [1]) ;
- aucun type de retour (pas même `void`) ;
- a ou pas des paramètres d'entrée, pouvant servir à initialiser les attributs à des valeurs externes à la méthode.

Lorsque le constructeur peut être appelé sans paramètre (constructeur sans aucun paramètre, ou dont chaque paramètre possède une valeur par défaut), on parle de **constructeur par défaut**.

Nb : Au-delà de l'initialisation des attributs, le constructeur peut aussi effectuer toutes sortes d'initialisations nécessaires à la création d'un objet.

Ex. : Le constructeur par défaut de la classe `Personne`.

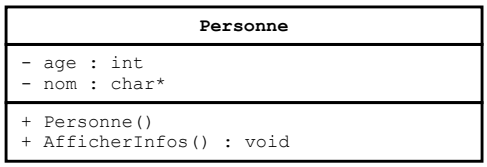

Figure 3.9 : exemple d'une classe avec un constructeur par défaut

```
class Personne
{
   public:
      Personne();  // constructeur par défaut
      void AfficherInfos();
   private:
      int age;
      char* nom;
};

Personne::Personne()
{
   age = 0;
   nom = new char[10];
   nom = "johndoe";   // nda : John Doe est l'équivalent anglais de M. Dupont ou
}                     // Durand pour désigner une personne de nom inconnu
```

[1] Puisque pour appeler une méthode – comme le constructeur – de l'intérieur de la classe, il faut disposer d'un objet complet, donc ayant un état pleinement défini, lequel ne peut être réalisé que grâce au… constructeur ! (nda : cf. paradoxe de l'œuf et de la poule).

En ce cas, lors de la simple déclaration d'un objet, un appel implicite au constructeur par défaut est réalisé. L'objet est donc parfaitement instancié.

Ex. : `Personne eleve1;  // un objet eleve1 de la classe Personne est déclaré`
`// et instancié par appel implicite au constructeur par défaut`

Le terme d'**instanciation** induit donc en réalité les actions suivantes :

- déclarer un nouvel objet : nom de l'objet + nom de sa classe (/type) ;
- réserver l'espace mémoire nécessaire au stockage de l'objet (soit implicitement lors d'une déclaration en statique, soit explicitement lors d'une déclaration dynamique) ;
- initialiser les attributs de l'objet : appel au constructeur.

Nb : Un constructeur ne peut être appelé que lors de l'instanciation d'un objet, et ne peut donc être appelé si l'objet est déjà instancié.

Puisque le constructeur peut avoir ou pas des paramètres d'entrée, il peut donc être surchargé comme toute méthode.

Ex. : Surcharges du constructeur de la classe `Personne`.

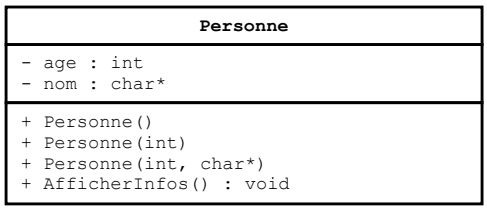

Figure 3.10 : exemple d'une classe avec plusieurs constructeurs

```
class Personne
{
   public:
      Personne();  // constructeur par défaut
      Personne(int);  // surcharge du constructeur
      Personne(int, char*);   // autre surcharge du constructeur
      void AfficherInfos();
   private:
      int age;
      char* nom;
};

Personne::Personne()
{
   age = 0;
   nom = new char[10];
   nom = "johndoe";
}

Personne::Personne(int a)
{
   age = a;
   nom = new char[10];     // réservation de l'espace mémoire nécessaire
   nom = "johndoe";
}

Personne::Personne(int a, char* n)
{
   age = a;
   nom = new char[strlen(n)+1];    // réservation de l'espace mémoire nécessaire
   strcpy(nom, n);      // recopie de la chaîne passée en paramètre dans l'attribut
}
```

Pour instancier un objet, on dispose alors de plusieurs constructeurs, et de plusieurs formes (appel implicite / explicite du constructeur).

Ex. :
```
Personne eleve1;                  // appel implicite au constructeur par défaut
Personne eleve2 = Personne();     // appel explicite au constructeur par défaut [1]
Personne eleve3(20);    // appel implicite au constructeur à 1 paramètre (int)
Personne eleve4 = Personne(19);   // appel explicite au constructeur à 1 paramètre
Personne eleve5(30, "joe");     //appel implicite au constructeur à 2 paramètres
Personne eleve6 = Personne(22, "bob");  // appel explicite au constructeur à 2
                                        // paramètres
```

Dans le cas de la déclaration d'un pointeur sur un objet, le constructeur ne peut jamais être appelé implicitement, il faut donc le faire explicitement pour réaliser une instanciation complète.

Ex. :
```
Personne* eleve7;          // déclaration d'un objet de la classe Personne
Personne* eleve8;          // déclaration d'un objet de la classe Personne
Personne* eleve9;          // déclaration d'un objet de la classe Personne

eleve7 = new Personne;       // appel explicite au constructeur par défaut
eleve8 = new Personne();     // appel explicite au constructeur par défaut
eleve9 = new Personne(15);   // appel explicite au constructeur à 1 paramètre

Personne* eleve10 = new Personne(16);  // instanciation d'un objet Personne
Personne* eleve11 = new Personne(28, "ken"); // instanciation d'un objet Personne
```

Si on ne spécifie aucun constructeur dans la définition de la classe, le compilateur en génère automatiquement un [2], qui est le constructeur par défaut, initialisant ainsi tous les attributs de la classe.

En revanche, dès qu'un constructeur est défini, que celui-ci soit *par défaut* ou non, le compilateur n'en génère aucun [3] ; il faut donc écrire explicitement le constructeur par défaut si l'on en a besoin.

Le constructeur par défaut est notamment nécessaire pour toute déclaration d'un tableau d'objets. Néanmoins, dans le cas d'un tableau statique, il est possible de spécifier explicitement l'initialisation de chacun des objets du tableau via un appel à l'une des surcharges du constructeur (nda : solution inadéquate pour des tableaux de grande taille).

Ex. :
```
Personne groupe1Eleves[2];       // appel implicite au constructeur par défaut
Personne groupe2Eleves[2] = { Personne(30, "joe"),   // appels explicites au
                              Personne(22) };        // constructeur
Personne* groupe3Eleves = new Personne[2];   // appel explicite au constructeur
                                             // par défaut
```

Nb : L'usage des valeurs par défaut pour les paramètres des constructeurs permet de faire correspondre plusieurs surcharges à 1 seul constructeur. Si on spécifie un constructeur avec une valeur par défaut pour chacun des paramètres, celui-ci correspond donc, entre autres, au constructeur par défaut.

Ex. : Un constructeur de la classe `Personne` avec des valeurs par défaut pour tous les paramètres.

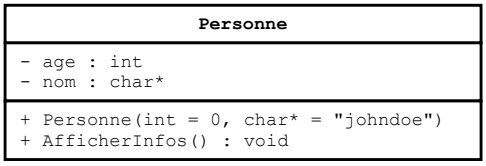

Figure 3.11 : exemple d'une classe avec un constructeur avec des valeurs par défaut pour tous les paramètres

[1] On distingue les écritures (1) : `MaClasse unObjet = MaClasse();` et (2) : `MaClasse unObjet; nouvelObjet = MaClasse();`. En (1), on déclare un nouvel objet et on l'initialise en effectuant un appel explicite au constructeur (1 construction) ; en (2), on déclare un nouvel objet, qui est construit grâce à un appel implicite au constructeur par défaut, puis on lui affecte une valeur en utilisant un objet anonyme (cf. 3.4.4) construit en effectuant un appel explicite au constructeur par défaut (2 constructions + 1 affectation). Les résultats obtenus suivant l'une ou l'autre des 2 méthodes sont les mêmes mais la méthode (2) est bien plus coûteuse en ressources.

[2] Celui-ci n'est pas très évolué, et s'il permet d'instancier un objet, il est peut-être inadéquat, voire même peu pertinent lorsque certains attributs sont déclarés en dynamique (cf. 3.4.1).

[3] Estimant, à raison, que vous n'avez donc pas oublié d'écrire un constructeur.

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");     // valeurs par défaut
      void AfficherInfos();
   private:
      int age;
      char* nom;
};
...

Personne::Personne(int a, char* n)   // constructeur correspondant à Personne(),
{                                    // Personne(int) et Personne(int, char*)
   age = a;
   nom = new char[strlen(n)+1];    // réservation de l'espace mémoire nécessaire
   strcpy(nom, n);      // recopie de la chaîne passée en paramètre dans l'attribut
}
```

3.3.4 Principe de « destructeur »

De la même manière qu'un mécanisme est mis en œuvre à la création d'un objet afin de l'initialiser (constructeur), un mécanisme spécifique est mis en œuvre à la destruction de l'objet.

Pour cela on dispose d'une méthode, appelée **destructeur**, qui a en charge de réaliser diverses opérations avant la destruction finale de l'objet (vidage des buffers, libération mémoire, etc.). Cette méthode spécifique a pour particularités :

- nom presque identique au nom de la classe, mais préfixée de '~' (tilde) (`~NomClasse()`) ;
- méthode publique (pour pouvoir être appelée de l'extérieur de la classe) ;
- aucun type de retour (pas même `void`) ;
- aucun paramètre d'entrée (ne peut donc être surchargée).

Le destructeur est appelé implicitement lors de la destruction d'un objet (fin du bloc de code de déclaration pour une instanciation statique, utilisation de l'opérateur `delete` [1] en cas d'instanciation dynamique). Même si cela est possible, il n'est généralement pas appelé explicitement.

Là aussi, si on ne spécifie aucun destructeur dans la définition de la classe, le compilateur en génère automatiquement un.

Ex. : Le destructeur de la classe `Personne`.

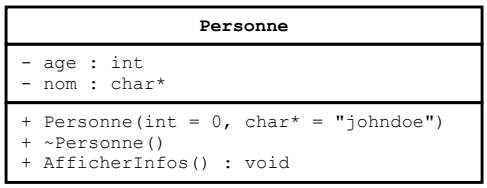

Figure 3.12 : exemple d'une classe avec destructeur

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();    // destructeur
      void AfficherInfos();
   private:
      int age;
      char* nom;
};
...
```

[1] Ou accessoirement, fin de l'exécution du programme si on fait confiance au système d'exploitation et au gestionnaire de mémoire.

```
Personne::~Personne()
{
   delete[] this->nom;
}
```

3.3.5 Le pointeur this

Tout membre (attribut ou méthode) appartient à un objet, donc lorsque l'on désire faire appel à un membre d'une classe, il faut impérativement préciser le nom de l'objet auquel appartient le membre désigné.

Or, dans l'implémentation d'une méthode, l'accès aux attributs, ainsi qu'aux autres méthodes de la classe, peut se faire directement, sans préfixer avec le nom de l'objet ; en effet, le nom de l'objet est induit car on se trouve « dans » la classe [1].

Cependant, il est possible, et même conseiller, de spécifier explicitement que le membre utilisé appartient bien à la classe, et n'est donc pas qu'une simple variable ou fonction.

Pour cela, il faut utiliser un paramètre caché disponible dans toute méthode de la classe, le **pointeur this**, qui fait référence à l'objet lui-même [2]. Il est implicitement déclaré `NomClasse* const this`, et s'utilise suivant la syntaxe `this->nom_membre` [3] pour accéder au membre.

Ex. : Un constructeur de la classe `Personne`.

```
Personne::Personne(int a, char* n)
{
   this->age = a;
   this->nom = new char[strlen(n)+1];
   strcpy(this->nom, n);
   this->AfficherInfos();     // affichage des informations
}
```

3.3.6 Les méthodes accesseurs

Les attributs sont théoriquement protégés grâce à l'encapsulation qui empêche tout accès de l'extérieur de la classe. Néanmoins, il peut parfois être utile d'augmenter les capacités de la classe à interagir avec son environnement. Pour réaliser cela, on dispose d'un mécanisme spécifique via des méthodes particulières.

Les méthodes **accesseurs** sont des méthodes permettant d'accéder aux attributs d'une classe à partir de l'extérieur, sans violer l'encapsulation. À chaque attribut, on peut ainsi faire correspondre 2 méthodes accesseurs :

- l'accesseur en lecture : méthode permettant de connaître la valeur d'un attribut à partir de l'extérieur de la classe, de prototype usuel `type_attribut GetAttribut()` [4] ;
- l'accesseur en écriture : méthode permettant de modifier la valeur d'un attribut à partir de l'extérieur de la classe, de prototype usuel `void SetAttribut(type_attribut)` [5].

Pour chaque attribut, on écrit donc les accesseurs nécessaires (lecture et/ou écriture) en fonction des besoins.

Nb : Les accesseurs lecture/écriture peuvent aussi être appelés *accesseurs*/*mutateurs* ou *getteurs*/*setteurs*.

Ex. : Les accesseurs de l'attribut `age`.

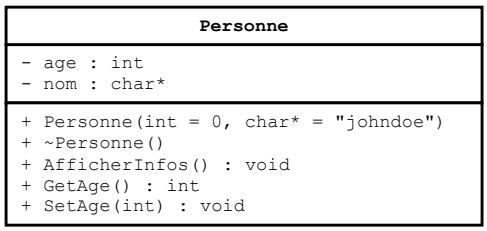

Figure 3.13 : exemple d'une classe avec les accesseurs d'un attribut

[1] Par ailleurs, il serait impossible de spécifier le nom de l'objet, car plusieurs objets dont on ne peut connaître le nom à l'avance seront certainement créés à partir de cette classe.
[2] This (eng) ≡ celui-ci (fr).
[3] `this` étant un pointeur de type `NomClasse*`, il faut donc utiliser '`->`' (tiret+flèche) pour accéder au membre, et non pas '`.`' (point).
[4] To get (eng) ≡ obtenir (fr).
[5] To set (eng) ≡ régler (fr).

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
      void AfficherInfos();
      int GetAge();             // accesseur en lecture pour l'attribut age
      void SetAge(int);         // accesseur en écriture pour l'attribut age
   private:
      int age;
      char* nom;
};
...

int Personne::GetAge()
{
   return this->age;
}
void Personne::SetAge(int a)
{
   this->age = a;
}
```

Nb : Lorsqu'un attribut possède un getteur et un setteur associés, celui-ci est alors pleinement accessible de l'extérieur de la classe. Néanmoins, cela demeure différent d'une visibilité publique car les accesseurs peuvent contrôler ses accès et ses modifications (Ex. : éviter une valeur d'écriture erronée, transmettre une valeur de lecture mise à l'échelle, etc.).

Ex. : La méthode `SetAge()` de la classe `Personne`.

```
void Personne::SetAge(int a)
{
   if ( a >= 0 ) this->age = a;    // vérification sur l'affectation de l'attribut
}
```

3.4 COMPLÉMENTS

3.4.1 Constructeur de recopie

Lors de l'instanciation d'un objet, on peut se baser sur un objet existant de la même classe pour en créer un nouveau, tel une recopie. La valeur de chaque attribut est alors simplement copiée d'un objet à l'autre. On dispose pour cela des 2 syntaxes `NomClasse nouvelObjet = ancienObjet` ou `NomClasse nouvelObjet(ancienObjet)`.

Ex. :
```
Personne* eleve10 = new Personne(16);   // instanciation d'un objet
Personne eleve12 = *eleve10;       // instanciation d'un objet à partir d'un autre
Personne eleve13(*eleve10);        // idem (syntaxe raccourcie)
```

Si ces syntaxes fonctionnent très bien pour de nombreuses classes, ce n'est pas le cas de classes possédant des attributs alloués dynamiquement ; dans ce cas, ce n'est pas la valeur pointée qui est recopiée, mais la valeur de l'adresse. Les objets ainsi recopiés, voulus indépendants, ont des attributs « communs » – ils possèdent chacun un attribut distinct alloué en dynamique pointant sur un même espace mémoire –, ce qui sous-entend que toute modification sur la valeur de cet attribut à partir d'un objet, se répercute sur l'autre objet.

Cet effet de bord est rarement désiré [1], d'autant plus que si l'un des objets est détruit, tous ses attributs aussi ; l'autre objet possède donc un attribut pointant sur un espace mémoire qui a été libéré (nda : erreur d'exécution en prévision !).

Ex. : Copie d'un objet `Personne` à partir d'un autre objet `Personne`.

[1] Pour disposer d'un attribut commun à plusieurs objets, on se tournera alors éventuellement vers la définition d'un attribut statique (cf. 3.4.3).

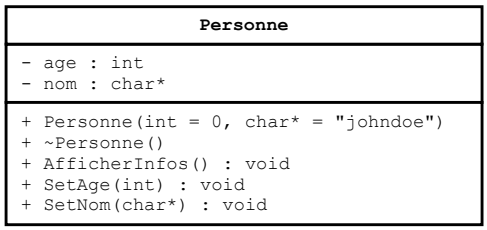

Figure 3.14 : exemple d'une classe sans constructeur de recopie

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
      void AfficherInfos();
      void SetAge(int);
      void SetNom(char*);
   private:
      int age;
      char* nom;
};
...


void Personne::SetNom(char* n)
{
   delete[] this->nom;     // destruction de l'ancienne version de l'attribut
   this->nom = new char[strlen(n)+1];
   strcpy(this->nom, n);
}


int main(void)
{
   Personne* eleve10 = new Personne(16);
   Personne eleve12 = *eleve10; // eleve12 est une copie de eleve10
   eleve10->AfficherInfos(); // affichage: "16" | "johndoe"
   eleve12.AfficherInfos();  // affichage: "16" | "johndoe"

   eleve10->SetAge(20);
   eleve10->SetNom("tim"); // modification de l'attribut nom de l'objet eleve10
   eleve10->AfficherInfos();    // affichage: "20" | "tim"
   eleve12.AfficherInfos();  // affichage: "16" | "tim" -> eleve12 a été modifié!
   // la modification sur l'objet eleve10 modifie aussi l'objet eleve12

   delete eleve10; // destruction de l'objet eleve10
   eleve12.AfficherInfos();  // affichage: "16" | "" -> eleve12 a été modifié!
   // la destruction de l'objet eleve10 a détruit l'un des attributs de eleve12

   return 0;
}
```

En réalité, pour l'instanciation d'un objet à partir d'un autre déjà instancié [1], le compilateur utilise bien un constructeur comme pour toute instanciation, mais un constructeur spécifique au clonage inter-objets, appelé **constructeur de recopie**.

À l'instar du constructeur par défaut, si un constructeur de recopie n'est pas explicitement écrit, alors le compilateur en génère automatiquement un, qui, comme il a été observé, trouve rapidement ses limites.

Le constructeur de recopie est ainsi appelé lors de l'instanciation d'un nouvel objet à partir d'un objet déjà instancié ; mais il est aussi utilisé de manière cachée dans 2 cas très importants lors de l'appel de toute méthode :

- lors d'un passage par valeur d'un paramètre de type objet (le paramètre-objet est cloné) [2] ;
- lors d'un renvoi par valeur d'un objet [1] (l'objet renvoyé est cloné) [2] [1].

[1] À l'aide de la syntaxe `NomClasse nouvelObjet = ancienObjet` ou `NomClasse nouvelObjet(ancienObjet)`.

[2] Ce qui explique notamment que l'objet dans la méthode et l'objet correspondant à l'extérieur de la méthode soient différents : l'un a été cloné pour obtenir l'autre.

Pour définir la manière dont un objet est copié à partir d'un autre, il suffit donc d'écrire le constructeur de recopie qui a comme prototype générique : `NomClasse(const NomClasse&)` [2].

Ex. : Le constructeur de recopie de la classe `Personne`.

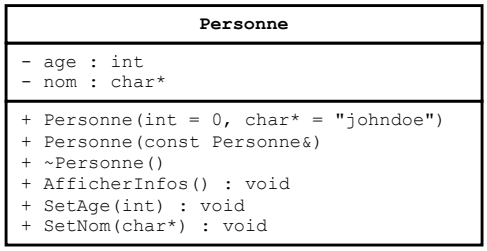

Figure 3.15 : exemple d'une classe avec constructeur de recopie

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      Personne(const Personne&);   // constructeur de recopie
      ~Personne();
      void AfficherInfos();
      void SetAge(int);
      void SetNom(char*);
   private:
      int age;
      char* nom;
};
...

Personne::Personne(const Personne& p)
{
   this->age = p.age;   // recopie de la valeur
   this->nom = new int[strlen(p.nom)+1];  // nouvelle réservation d'espace mémoire
   strcpy(this->nom, p.nom);    // recopie de la chaîne
}


int main(void)
{
   Personne* eleve10 = new Personne(16);
   Personne eleve12 = *eleve10; // eleve12 est une copie de eleve10
   eleve10->AfficherInfos(); // affichage: "16" | "johndoe"
   eleve12.AfficherInfos();  // affichage: "16" | "johndoe"

   eleve10->SetAge(20);
   eleve10->SetNom("tim");   // modification de l'attribut nom de l'objet eleve10
   eleve10->AfficherInfos();    // affichage: "20" | "tim"
   eleve12.AfficherInfos();  // affichage: "16" | "johndoe" -> eleve12 inchangé
   // la modification sur l'objet eleve10 ne modifie pas l'objet eleve12

   delete eleve10; // destruction de l'objet eleve10
   eleve12.AfficherInfos();  // affichage: "16" | "johndoe" -> eleve12 inchangé
   // la destruction de l'objet eleve10 ne change rien à eleve12 et ses attributs

   return 0;
}
```

Nb : Dans le constructeur de recopie, on peut noter que l'objet `this` peut accéder sans problème aux attributs de l'objet passé en paramètre, ce qui est possible car les 2 objets font partie de la même classe, et l'encapsulation en C++ est réalisée au niveau de la classe, et non au niveau de l'objet [3].

[1] Par souci d'optimisation, certains compilateurs contournent parfois cet appel au constructeur de recopie : la valeur retournée est positionnée directement en zone de mémoire temporaire accessible au contexte appelant plutôt que dans un segment de mémoire spécifique à la méthode pour être ensuite recopié dans une zone de mémoire temporaire lorsque la méthode se termine (cf. E.2.2).

[2] La syntaxe `NomClasse nouvelObjet(ancienObjet)`, permettant de créer un nouvel objet à partir d'un autre déjà existant, devient alors évidente.

[3] cf. 3.2.4.

3.4.2 Instanciation et destruction d'un attribut-objet

Le rôle d'un constructeur est d'initialiser les attributs d'un objet afin de réaliser une instanciation complète. Lorsqu'une classe comprend un attribut de type objet [1], celui-ci doit être pleinement instancié pour pouvoir instancier un objet de la classe. Autrement dit, l'instanciation d'un objet d'une classe suppose et impose que ses attributs-objets [2] soient eux aussi instanciés ; cette instanciation est réalisée juste avant l'instanciation de l'objet.

De la même manière, le destructeur, qui a en charge d'effectuer proprement la destruction des attributs, s'occupe de détruire les attributs-objets tout de suite après la destruction de l'objet lui-même.

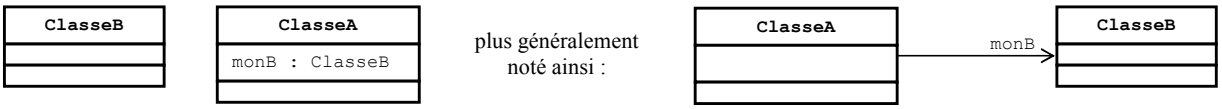

Figure 3.16 : représentation d'une classe avec un attribut de type objet

Ces mécanismes liés à la construction et à la destruction des attributs-objets peuvent être réalisés de manière invisible, en faisant implicitement appel au constructeur par défaut des attributs-objets ainsi qu'à leur destructeur.

Ex. : La classe `Personne` a un attribut-objet de la classe `Vehicule`, qui propose un constructeur par défaut.

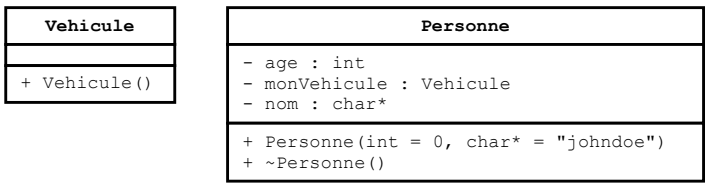

Figure 3.17 : exemple d'une classe possédant un attribut de type objet

```
class Vehicule
{
   public:
      Vehicule();     // constructeur par défaut
};
...


class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
   private:
      int age;
      Vehicule monVehicule;
};
...

Personne::Personne(int a, char* n)      // l'attribut monVehicule est initialisé
                                        // via un appel implicite à Vehicule()
                                        // réalisé avant les autres initialisations
{                                       // définies dans Personne()
   this->age = a;
   this->nom = new char[strlen(n)+1];
   strcpy(this->nom, n);
}

Personne::~Personne()
{
   delete[] this->nom;     // l'attribut monVehicule est détruit via un appel
                           // implicite à ~Vehicule() réalisé après les autres
}                          // destructions définies dans ~Personne()
```

[1] En modélisation UML, on parle d'*association* (cf. D.2).

[2] Aussi appelés *objets membres*.

Dans le cas de la construction, l'appel implicite au constructeur n'est en revanche plus possible lorsque la classe de l'attribut-objet ne propose pas de constructeur par défaut ; en effet, l'attribut-objet manque alors d'informations pour initialiser correctement l'ensemble de ses propres attributs.

Pour pallier ce problème, il suffit de définir une **liste d'initialisation** associée au constructeur, en spécifiant, pour chaque attribut-objet pour lequel cela s'avère nécessaire, l'appel au constructeur avec les paramètres adéquats, selon la syntaxe :

```
NomClasse::NomClasse(...)
: nomAttributObjet1(paramètres_constructeur_attributObjet1)
, nomAttributObjet2(paramètres_constructeur_attributObjet2)
, ...
{ /* code du constructeur */ }
```

Nb : La syntaxe `nomAttributObjet(paramètres_constructeur_attributObjet)` est à rapprocher de la syntaxe d'instanciation d'un objet utilisant une surcharge du constructeur : `Classe objet(paramètres);`.

Ex. : La classe `Personne` possède un attribut-objet de type `Adresse` ne possédant pas de constructeur par défaut (nda : par souci de lisibilité, l'attribut `nom` a été provisoirement omis de la déclaration de la classe `Personne` pour tous les exemples d'instanciation d'un attribut-objet).

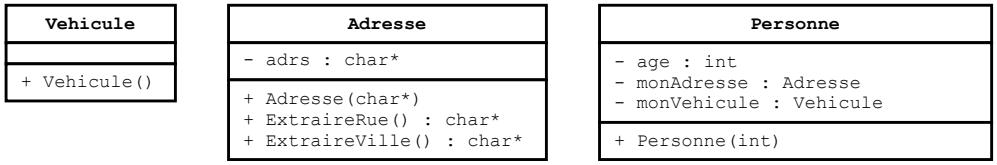

Figure 3.18 : exemple de déclaration d'une classe possédant un attribut-objet sans constructeur par défaut

```
class Adresse
{
   public:
      Adresse(char*);    // pas de constructeur par défaut
      char* ExtraireRue();
      char* ExtraireVille();
   private:
      char* adrs;
};
...


class Personne
{
   public:
      Personne(int = 0);
   private:
      int age;
      Adresse monAdresse;
      Vehicule monVehicule;
};

Personne::Personne(int a)       // appel implicite à Vehicule()
: monAdresse("adresse inconnue")  // le constructeur monAdresse() n'existant pas,
                                  // on réalise un appel explicite à
{                                 // Adresse(char*) pour initialiser monAdresse
   this->age = a;
}
```

Les paramètres passés au constructeur de l'attribut-objet peuvent être des littéraux, ou bien des variables issues directement des paramètres d'appels du constructeur.

Ex. : La classe `Personne` possède plusieurs attributs-objets (`monAdresse` et `maProfession`) appartenant à des classes ne possédant pas de constructeur par défaut (`Adresse` et `Profession`).

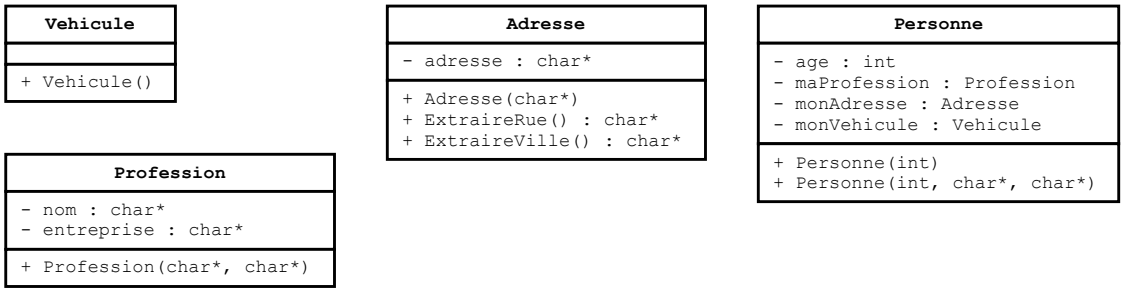

Figure 3.19 : exemple de déclaration d'une classe possédant plusieurs attributs-objets sans constructeur par défaut

```
class Profession
{
   public:
      Profession(char*, char*);    // pas de constructeur par défaut
   private:
      char* nom;
      char* entreprise;
};
...


class Personne
{
   public:
      Personne(int = 0);
      Personne(int, char*, char*);    // constructeur avec des paramètres pour
   private:                           // initialiser monAdresse et maProfession
      int age;
      Adresse monAdresse;
      Profession maProfession;
      Vehicule monVehicule;
};
...

Personne::Personne(int a)
: monAdresse("adresse inconnue")        // appel aux constructeurs Adresse(char*)
, maProfession("sans profession", "")   // et Profession(char*, char*)
{
   this->age = a;
}

Personne::Personne(int a, char* adr, char* pro)   // des paramètres permettant
: monAdresse(adr)                                 // d'initialiser monAdresse et
, maProfession(pro, "n.c")                        // maProfession sont passés
                                                  // au constructeur Personne()
                                                  // pour qu'ils soient utilisés
                                                  // par Adresse(char*) et
{                                                 // Profession(char*, char*)
   this->age = a;
}
```

Si un objet déjà instancié doit être utilisé pour construire un attribut-objet, il suffit de le passer en paramètre du constructeur, lequel fait alors usage du constructeur de recopie de la classe de l'attribut-objet.

Ex. : La classe `Personne` permet d'initialiser certains de ses attributs-objets (`monAdresse` et `maProfession`) en utilisant des objets déjà instanciés.

```
class Personne
{
   public:
      Personne(int);
      Personne(int, char*, char*);
      Personne(int, Adresse, Profession);  // constructeur avec des paramètres
   private:                                // pour initialiser monAdresse et
      int age;                             // maProfession en utilisant des objets
      Adresse monAdresse;
      Profession maProfession;
      Vehicule monVehicule;
};
...

Personne::Personne(int a, Adresse objAdr, Profession objPro)
: monAdresse(objAdr)      // appel aux constructeurs de recopie Adresse(Adresse&)
, maProfession(objPro)    // et Profession(Profession&)
{
   this->age = a;
}
```

En réalité, le mécanisme de liste d'initialisation peut être utilisé pour tout attribut d'une classe : attribut-objet sans constructeur par défaut, attribut-objet avec constructeur par défaut mais que l'on désire initialiser en utilisant une surcharge du constructeur, attribut-non-objet, etc.

La liste d'initialisation doit notamment être impérativement utilisée pour les attributs de type référence ainsi que les attributs constants, même si l'on dispose d'un constructeur par défaut.

L'initialisation d'un attribut constant étant réalisée lors de l'instanciation de l'objet, et non lors de la déclaration de la classe (*.h*) [1], cela ne peut donc s'opérer qu'au moment-même de sa déclaration.

Ex. : La classe `Personne` possède l'attribut constant `monSexe` représentant le sexe de la personne.

```
class Personne
{
   public:
      ...
      Personne(int, char);
   private:
      int age;
      const char monSexe;       // attribut constant (valeur 'm'/'f')
      Adresse monAdresse;
      Profession maProfession;
      Vehicule monVehicule;
};
...

Personne::Personne(int a, char s)
: monAdresse("adresse inconnue")
, maProfession("sans profession", "")
, monSexe(s)      // initialisation de l'attribut constant
{
   this->age = a;
}
```

Une référence ne peut être initialisée qu'au moment-même de sa déclaration, en utilisant une variable / un objet du même type déjà instancié [2].

Ex. : La classe `Personne` possède l'attribut de type référence d'objet `monTelephone` représentant le numéro de téléphone de la personne.

[1] Ce qui reviendrait à donner à l'attribut une valeur identique pour tous les futurs objets de la classe.
[2] cf. 2.2.2.8.

```
class Telephone
{
   public:
      Telephone();
      Telephone(int num);
   private:
      int numero;
};
...


class Personne
{
   public:
      ...
      Personne(int, Telephone);
   private:
      int age;
      const char monSexe;
      Adresse monAdresse;
      Profession maProfession;
      Vehicule monVehicule;
      Telephone& monTelephone;        // attribut de type référence
};
...

Personne::Personne(int a, Telephone t)
: monAdresse("adresse inconnue")
, maProfession("sans profession", "")
, monSexe('?')
, monTelephone(t)    // initialisation de l'attribut de type référence Telephone&
                     // avec l'objet t de type Telephone passé en paramètre
{
   this->age = a;
}
```

3.4.3 Membres statiques

Lors de la création d'une instance – à l'aide du constructeur de recopie si besoin est – le nouvel objet possède son propre exemplaire de chaque attribut défini dans la classe ; deux attributs homonymes appartenant chacun à un objet différent de la même classe possèdent donc leur propre espace mémoire et sont totalement indépendants.

De la même manière, chaque objet a la possibilité d'utiliser n'importe quelle méthode définie dans la classe, de manière totalement indépendante des autres objets de la classe.

À l'inverse, on peut définir certains membres comme étant communs et partagés par tous les objets de la classe ; ce sont des **membres statiques**. En opposition aux membres non-statiques, qui sont des *attributs d'instance* et des *méthodes d'instance*, on les appelle aussi *attributs de classe* et *méthodes de classe*.

De fait, les membres statiques, n'appartenant pas à une instance, mais à la classe, existent et peuvent être appelés même si aucune instance de la classe n'existe ou n'est disponible. Les conséquences sont les suivantes :

- Un attribut statique possède une valeur commune pour tous les objets de la classe, qui, si elle est modifiée par un objet, sera conservée même si cet objet est détruit ; l'initialisation est effectuée en global ;
- Une méthode statique peut être exécutée de manière classique à partir d'une instance, mais aussi localement sans avoir à instancier d'objet ; en revanche, elle ne peut accéder qu'aux membres statiques de la classe [1], ne peut faire usage du pointeur this [2], et ne peut être surchargée.

Pour définir un membre comme étant statique, il suffit de spécifier le modificateur `static` lors de la déclaration du membre dans la déclaration de la classe, suivant la syntaxe `static visibilité nomMembre;`.

Par la suite, la manière d'utiliser le membre statique dépend du contexte :

- dans une méthode non-statique de la classe : le membre est préfixé par le pointeur this (`this->membre`), ou par rien ;
- dans une méthode statique de la classe : le membre est préfixé par le nom de la classe associé à l'opérateur de résolution de portée (`NomClasse::membre`), ou par rien ;

[1] Puisque pour accéder aux membres non-statiques, il faut disposer d'une instance de la classe.

[2] Puisque `this` est une référence à l'objet qui a invoqué la méthode, objet qui n'existe pas dans le cas d'une méthode statique.

- autre (extérieur de la classe) : si le membre est utilisé de manière « classique » en étant associé à une instance de la classe, il est préfixé par le nom de cette instance (`instance.membre` / `instance->membre`) ; si on ne dispose pas d'instance ou si cela n'a pas de sens, le membre est préfixé par le nom de la classe associé à l'opérateur de résolution de portée (`NomClasse::membre`).

Dans la représentation de la classe (représentation UML), un membre statique est souligné.

Ex. : Des membres statiques pour la classe `Personne`.

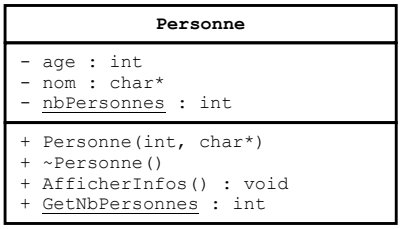

Figure 3.20 : exemple de déclaration d'une classe avec des membres statiques

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
      void AfficherInfos();
      static int GetNbPersonnes();    // méthode statique
   private:
      int age;
      char* nom;
      static int nbPersonnes;    // attribut statique
};
...

int Personne::nbPersonnes = 0;       // initialisation de l'attribut statique

Personne::Personne(int a, char* n)
{
   this->age = a;
   this->nom = new char[strlen(n)+1];
   strcpy(this->nom, n);
   this->nbPersonnes++;       // accès à l'attribut statique commun
}
int Personne::GetNbPersonnes()
{
   return Personne::nbPersonnes;      // contexte statique : préfixe 'this->'
}                                     // impossible


int main(void)
{
   Personne eleve3(20);
   cout << Personne::GetNbPersonnes() << endl;     // affichage: "1"
   Personne eleve4 = Personne(19);
   cout << Personne::GetNbPersonnes() << endl;     // affichage: "2"

   return 0;
}
```

3.4.4 Objets anonymes

La notion d'**objet anonyme** désigne une instance construite mais non déclarée, utilisée une seule fois et localement. On construit un objet anonyme d'une classe de la même manière qu'on instancie un objet à l'aide d'un constructeur [1] en statique ou en dynamique selon les syntaxes respectives `NomClasse(...)` ou `new NomClasse(...)`.

Ex. : Utilisation d'un objet anonyme pour retourner une nouvelle instance.

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
      void AfficherInfos();
      static int GetNbPersonnes();
      static Personne* AjouterPersonne();
   private:
      int age;
      char* nom;
      static int nbPersonnes;
};
...

Personne* Personne::AjouterPersonne()
{
   return new Personne();     // on retourne un objet anonyme
}


int main(void)
{
   Personne eleve3(20);
   cout << Personne::GetNbPersonnes() << endl;     // affichage: "1"

   Personne* eleve14;
   eleve14 = Personne::AjouterPersonne();       // ajout d'une personne
   cout << eleve14->GetNbPersonnes() << endl;      // affichage: "2"

   // appel d'une méthode sur un objet anonyme
   //(l'appel à Personne() ajoute une personne)
   cout << Personne().GetNbPersonnes() << endl;    // affichage: "3"
   // nda : mieux vaudrait ici utiliser une méthode statique, afin d'éviter
   // l'ajout d'une personne

   return 0;
}
```

3.4.5 Objets constants et méthodes constantes

On peut déclarer un **objet constant**, c'est-à-dire que ses attributs-membres ne peuvent être modifiés pendant sa durée de vie. On use pour cela de la même syntaxe que les variables constantes, avec le modificateur `const`.

Ex. : `const Personne eleve15 = Personne(12, "jan"); // instanciation d'un objet constant`

Pour garantir la constance d'un objet déclaré constant, il faut s'assurer que les méthodes qui lui sont appliquées ne le modifient pas. Pour réaliser cela, les méthodes en question doivent être des **méthodes constantes**, c'est-à-dire des méthodes qui mentionnent explicitement qu'elles n'effectuent aucune modification à l'objet sur lequel elles sont appliquées.

[1] En effet, un constructeur (même si aucun type de retour n'est explicitement défini) retourne bien quelque chose ; il s'agit d'un objet ou d'une référence à un objet de la classe à laquelle il appartient. Ce qui explique les syntaxes `NomClasse nomObjet = NomClasse()` (statique) et `NomClasse* nomObjet = new NomClasse()` (dynamique) qui se présentent comme une affectation (même si une construction reste fondamentalement différente d'une affectation).

En effet, dans une méthode non-constante, le pointeur `this` est implicitement déclaré `NomClasse* const this` [1]. Appeler une méthode non-constante sur un objet constant est impossible car cela reviendrait à permettre de contourner la déclaration constante [2]. À l'inverse, dans une méthode constante, le pointeur `this` est implicitement déclaré `const NomClasse* const this` [3], et peut donc être initialisé avec l'adresse de l'objet constant.

Pour définir qu'une méthode n'accède jamais qu'en lecture à l'objet sur lequel elle est appliquée, et qu'elle ne lui apporte donc aucune modification (que ce soit directement ou via l'appel à une autre méthode non-constante), on le précise dans le prototype de la méthode à l'aide du modificateur `const`, positionné à la fin, après la déclaration des paramètres [4].

Ex. : La méthode `AfficherInfos()` n'effectue aucune modification de l'objet – accès en lecture uniquement. Elle peut donc être qualifiée de `const`.

```
class Personne
{
   public:
      Personne(int = 0, char* = "johndoe");
      ~Personne();
      void AfficherInfos() const;  // méthode constante
   private:
      int age;
      char* nom;
};
...

void Personne::AfficherInfos() const
{
   ...
}
```

S'il est impossible d'appeler une méthode non-constante sur un objet constant, l'inverse est en revanche parfaitement possible, et toute méthode constante peut être appliquée à un objet non-constant [5].

L'usage veut donc de qualifier de `const` toute méthode qui n'effectue aucune modification de l'objet [6], il y a tout à y gagner [7].

Le modificateur `const` ajouté à une méthode fait partie de sa signature. Cela induit que :

- `const` doit être précisé dans la déclaration de la méthode (*.h*), et dans son implémentation (*.cpp*) ;
- Une méthode constante peut être surchargée par une méthode non-constante : on peut ainsi préciser le comportement de la méthode selon qu'elle soit appelée pour un objet constant ou non-constant.

3.4.6 Classes, méthodes et fonctions amies

Dans certains cas, il est nécessaire d'augmenter la capacité d'interaction d'une classe avec son environnement, en permettant notamment à des éléments extérieurs à la classe d'accéder aux membres non-publics.

Pour les attributs, le recours aux accesseurs est une solution, mais si on peut contrôler ce qui est fait, on ne peut en revanche contrôler qui le fait, et toute classe ou fonction extérieure peut y accéder ; dans le cas des méthodes, aucun mécanisme vu jusqu'à présent ne le permet [8].

Une classe peut permettre à une autre classe, à une fonction ou à une méthode d'une autre classe de devenir **amie**. L'« amie » a ainsi toute liberté d'accès sur les membres protégés par l'encapsulation, comme si elle appartenait à la classe.

La définition d'une amie se réalise dans la définition de la classe, avant la déclaration des membres en utilisant le modificateur `friend` en déclarant l'entité amie suivant la syntaxe `friend <déclaration amie>`.

[1] L'adresse de `this` ne peut être modifiée (cf. 3.3.5).
[2] Le pointeur `this` (non-constant) pointerait sur un objet constant : il serait possible de modifier l'objet en passant par `this`. Le compilateur n'autorise pas cela.
[3] L'adresse et la valeur de `this` ne peuvent être modifiées.
[4] Car positionné au début du prototype, le modificateur `const` se rapporterait au type de valeur renvoyée par la méthode.
[5] On peut contraindre temporairement un objet non-constant à être constant, tout comme on peut faire pointer un pointeur constant sur un objet non-constant.
[6] Ce qui exclut d'office tous les constructeurs et le destructeur.
[7] Méthode applicable indifféremment pour objets constants et non-constants, plus de contrôle lors de la compilation, meilleure lisibilité, etc.
[8] Généralement, si une méthode doit pouvoir être appelée de l'extérieur de la classe, on lui affecte la visibilité '`public`'.

Ex. : Des amies pour la classe `Personne` (nda : l'attribut `nom` a été provisoirement omis de la déclaration de la classe `Personne` pour cet exemple de classes, méthodes et fonctions amies).

```
class Personne;      // déclaration nécessaire pour que Groupe::personnes et
                     // et Groupe::AddPersonne(Personne) soient reconnus

class Groupe
{
   public:
      Groupe(int);
      ~Groupe();
      void AfficherInfos();
      void AddPersonne(Personne);  /* accès libre à tous les membres de Personne */
   private:
      int maxPersonnes;    // nombre maximal de personnes
      int numPersonnes;    // nombre actuel de personnes
      Personne* personnes;
};


class Personne
{
   friend class Categorie;      // déclaration d'une classe amie
   friend void Groupe::AfficherInfos();   // déclaration d'une méthode amie
   public:
      Personne(int = 0);
      void AfficherInfos();
   private:
      int age;
};


class Categorie
{
   /* accès libre à tous les membres de Personne */
};


Groupe::Groupe(int max)
{
   this->numPersonnes = 0;
   this->maxPersonnes = max;
   this->personnes = new Personne[this->maxPersonnes];
}

Groupe::~Groupe()
{
   delete[] this->personnes;
}

void Groupe::AfficherInfos()
{
   for ( int i=0 ; i < this->numPersonnes ; i++ ) {
      cout << i << ": " << personnes[i].age << endl;
   }
}

void Groupe::AddPersonne(Personne p)
{
   this->personnes[this->numPersonnes++] = p;
}

...
```

```
int main(void)
{
   Groupe g(10);

   g.AddPersonne(Personne(20));
   g.AddPersonne(Personne(19));
   g.AfficherInfos();

   return 0;
}
```

Il est à noter que l'« amitié » n'est ni bijective [1], ni transitive [2] et ne s'hérite pas [3].

Nb : L'utilisation d'« amies » peut traduire un défaut de conceptualisation objet. On y a facilement recours pour résoudre des problèmes d'accès ponctuels. Néanmoins, avant de déclarer des « amies », il convient tout d'abord de remettre en cause la définition de ses différentes classes.

Les « amies » sont en effet un moyen de contourner l'encapsulation d'une classe, et en ce sens ne respectent pas les principes de la programmation objet.

---

[1] Non-bijectivité : si B est « amie » de A (`class A {friend class B; …};`), A n'est pas « amie » de B sauf si explicitement déclaré.

[2] Non-transitivité : si B est « amie » de A, que C est « amie » de B, C n'est pas « amie » de A sauf si explicitement déclaré.

[3] Non-héritage : si B est « amie » de A, que D hérite de A, B n'est pas « amie » de D sauf si explicitement déclaré. Plus exactement, l'amitié persiste tout de même pour les membres hérités : D peut donc avoir accès aux membres de B qui sont hérités de A, mais uniquement ceux-là.

4 SURCHARGE D'OPÉRATEURS

4.1 INTRODUCTION

Pour les variables numériques, l'usage des opérateurs permet de manipuler de manière induite et intuitive ces variables entre elles, et offre la possibilité de réaliser des opérations en utilisant une syntaxe naturelle.

Ex. :
```
double x, y, z;
z = x + y;      // ajouter y à x et affecter le tout dans z
```

En revanche, avec des objets, pour manipuler les instances d'une même classe entre eux on ne dispose que des éventuelles méthodes que fournit la classe.

Ex. : On dispose d'une classe `Chaine` permettant de gérer des chaînes de caractères.

```
Chaine chn1, chn2, chn3;
chn3.Copier(chn1.Concatener(chn2));  // syntaxe chn3=chn1+chn2 ne fonctionne pas
```

Pour augmenter la lisibilité, on dispose d'un mécanisme permettant de donner un sens particulier aux opérateurs lorsqu'ils sont utilisés avec des objets.

4.2 PRINCIPES ET DÉFINITIONS

La **surcharge d'opérateur** consiste à définir, pour un opérateur donné associé à une classe, l'ensemble des actions devant être réalisées lorsque cet opérateur est utilisé avec une instance de cette classe.

Cette surcharge s'opère en définissant une méthode, ou éventuellement une fonction externe à la classe, sachant qu'à chaque opérateur correspond une méthode implicite de nom générique `operator<opérateur>()`, et de prototype générique `type_retour operator<opérateur>(paramètres)`, sachant que les paramètres (entrée et sortie) peuvent être du type de la classe à laquelle est associée l'opérateur, ou bien d'autres types d'objets ou de variables [1].

Les paramètres d'entrée peuvent être passés par valeur ou par référence, sachant qu'il est préférable de les passer par référence dans un souci d'optimisation mémoire.
Le paramètre retourné est généralement passé par valeur lorsque la méthode crée un nouvel objet afin d'assurer une copie totale de ce qui est renvoyé [2]. Dans le cas des surcharges d'opérateurs réalisant une affectation (=, +=, −=, etc., ainsi que ++ en préincrémention), un objet existant est modifié, c'est alors une référence qui est renvoyée [3].

Même surchargé, un opérateur conserve sa priorité, son associativité, et sa pluralité (unaire / binaire / ternaire), mais en revanche, il perd sa commutativité. Ainsi, lorsque l'on écrit `z = a + b + c`, cela correspond à `z.operator=((a.operator+(b)).operator+(c))`.

Tous les opérateurs (affectation, arithmétiques, binaires, évaluation d'expression, etc.) peuvent être surchargés, hormis `:: . ?: .* sizeof`. Les opérateurs =, + et << sont notamment souvent surchargés.

Hormis la surcharge de l'opérateur =, toutes les surcharges d'opérateurs, sont transmises par héritage, comme des méthodes classiques.

[1] La pertinence de l'existence et de la signification de telles surcharges dépend de la classe elle-même, de l'opérateur surchargé et du contexte.
[2] Et pas de l'adresse d'un objet déclaré dans la méthode, qui est détruit à la fin de celle-ci ; l'adresse pointerait alors sur un objet n'existant plus.
[3] Ce qui permet aussi de chaîner les opérateurs en faisant de la valeur retournée une LValue (cf. 2.2.3.3).

4.3 IMPLÉMENTATION

4.3.1 Surcharge d'opérateur par une méthode

Pour une surcharge d'opérateur via une méthode, le premier (ou le seul) opérande est l'objet sur lequel est appliqué la méthode. Le prototype générique est donc :

- pour un opérateur unaire : `X operator<opérateur>()` ;
- pour un opérateur binaire : `X operator<opérateur>(Y)`.

Ex. : La surcharge de l'opérateur + (concaténation de chaînes) pour la classe `Chaine` par la méthode `operator+()`. Il s'agit donc d'un opérateur binaire, appliqué à un objet `Chaine`, prenant en paramètre une chaîne et retournant une chaîne. Le prototype est donc `Chaine operator+(const Chaine&)`.

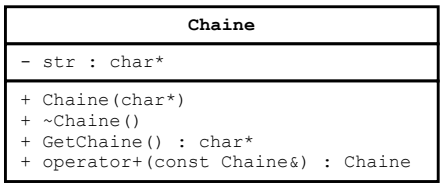

Figure 4.1 : exemple d'une classe avec surcharge de l'opérateur +

```
class Chaine
{
   public:
      Chaine(char*);
      ~Chaine() { delete[] str; }
      char* GetChaine() { return this->str; }
      Chaine operator+(const Chaine&);     // surcharge de l'opérateur +
   private:
      char* str;
};

Chaine::Chaine(char* t)
{
   this->str = new char[strlen(t)+1];      // allocation mémoire (avec prise en
   strcpy(this->str, t);                   // compte du '\0' final)
}

Chaine Chaine::operator+(const Chaine& chn)
{
   char* nvchn;
   nvchn = new char[strlen(this->str) + strlen(chn.str) + 1]; // allocation mémoire
   strcpy(nvchn, this->str);       // copie de la première chaîne
   strcat(nvchn, chn.str);         // concaténation avec la seconde chaîne

   return Chaine(nvchn);      // objet anonyme initialisé avec la nouvelle chaîne
}


int main(void)
{
   Chaine str1("bonjour!");
   Chaine str2("ca va!");

   Chaine str3 = str1 + str2;   // identique à 'Chaine str3 = str1.operator+(str2)'
   cout << str3.GetChaine() << endl;    // affichage: "bonjour!ca va!"

   Chaine str4("");
   str4 = str2 + str1;        // identique à 'str4.operator=(str2.operator+(str1))'
   cout << str4.GetChaine() << endl;    // affichage: "ca va!bonjour!"

   return 0;
}
```

Dans l'exemple ci-dessus, il convient de distinguer les 2 syntaxes différentes d'initialisation d'un objet :

- `Chaine str3 = str1 + str2;` : utilisation du constructeur [1] ;
- `Chaine str4(""); str4 = str2 + str1;` : utilisation de l'opérateur d'affectation = [2].

Pourtant cet opérateur d'affectation = a pu être utilisé avec des objets d'une classe, alors qu'aucune surcharge n'a été définie pour cette classe.

C'est le compilateur qui a automatiquement généré une surcharge « basique » de cet opérateur car aucune n'avait été explicitement définie, réalisant ainsi le même genre d'opérations que le constructeur de recopie créé par défaut par le compilateur. On peut donc lui reprocher les mêmes défauts, à savoir que lors de la recopie d'un attribut alloué dynamiquement, c'est l'adresse qui est recopiée et pas le contenu pointé.

Les raisons justifiant d'écrire le constructeur de recopie justifient donc aussi d'écrire la surcharge de l'opérateur = ; en toute logique, l'écriture de l'un implique donc l'écriture de l'autre.

Ex. : La surcharge de l'opérateur = pour la classe `Chaine` par la méthode `Chaine& operator=(Chaine)`, ainsi que la surcharge du constructeur de recopie.

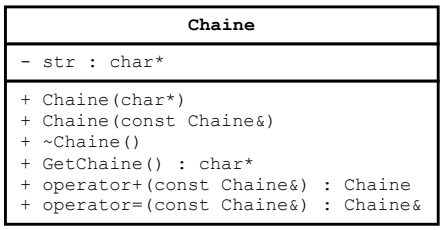

Figure 4.2 : exemple d'une classe avec surcharge des opérateurs + et =

```
class Chaine
{
   public:
      Chaine(char*);
      Chaine(const Chaine&);       // constructeur de recopie
      ~Chaine() { delete[] str; }
      char* GetChaine() { return this->str; }
      Chaine operator+(const Chaine&);
      Chaine& operator=(const Chaine&);    // surcharge de l'opérateur =
   private:
      char* str;
};
...

Chaine::Chaine(const Chaine& chn)
{
   this->str = new char[strlen(chn.str) + 1];   // allocation mémoire
   strcpy(this->str, chn.str);
}

Chaine& Chaine::operator=(const Chaine& chn)
{
   delete this->str;          // destruction de l'ancien objet
   this->str = new char[strlen(chn.str) + 1];   // nouvelle allocation mémoire
   strcpy(this->str, chn.str);

   return *this;
}
```

[1] Plus précisément, dans l'ordre, appel de : `operator+()`, constructeur `Chaine(char*)`.

[2] Plus précisément, dans l'ordre, appel de : constructeur `Chaine(char*)`, `operator+()`, `constructeur Chaine(char*)`, `operator=()`.

```
int main(void)
{
   Chaine str1("bonjour!");
   Chaine str2("ca va!");

   Chaine str3 = str1 + str2;   // appel au constructeur Chaine(char*)
   cout << str3.GetChaine() << endl;    // affichage: "bonjour! ca va!"

   Chaine str4("");        // appel au constructeur Chaine(char*)
   str4 = str2 + str1;     // appel à l'opérateur =
   cout << str4.GetChaine() << endl;    // affichage: "ca va! bonjour!"

   return 0;
}
```

Comme toute méthode, les surcharges d'opérateurs peuvent être surchargées.

Ex. : Une surcharge additionnelle de l'opérateur `+` pour la classe `Chaine` par la méthode `Chaine operator+(char)`.

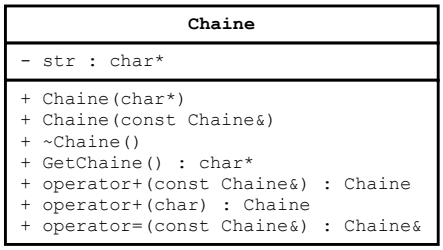

Figure 4.3 : exemple d'une classe avec plusieurs surcharges pour un opérateur

```
class Chaine
{
   public:
      Chaine(char*);
      Chaine(const Chaine&);
      ~Chaine() { delete[] str; }
      char* GetChaine() { return this->str; }
      Chaine operator+(const Chaine&);     // surcharge de l'opérateur + (1)
      Chaine operator+(char);              // surcharge de l'opérateur + (2)
      Chaine& operator=(const Chaine&);
   private:
      char* str;
};
...

Chaine Chaine::operator+(char c)
{
   char* nvchn = new char[strlen(this->str) + 2];
   strcpy(nvchn, this->str);
   nvchn[strlen(this->str)] = c;
   nvchn[strlen(this->str)+1] = '\0';

   return Chaine(nvchn);
}


int main(void)
{
   Chaine str1("bonjour!");

   Chaine str5 = str1 + '?';
   cout << str5.GetChaine() << endl;    // affichage: "bonjour!?"

   return 0;
}
```

4.3.2 Surcharge d'opérateur par une fonction

Certains opérateurs (comme << et >>) ne peuvent pas être surchargés par les méthodes d'une classe car ils font intervenir des types primitifs ou bien des classes auxquelles le programmeur n'a pas accès (Ex. : classes de la STL). Il faut alors utiliser une fonction externe à la classe.

Le prototype générique de la surcharge d'un opérateur par une fonction est :

- pour un opérateur unaire : `X operator<opérateur>(Y)` ;
- pour un opérateur binaire : `X operator<opérateur>(Y, Z)`.

Nb : Une pratique courante veut que les fonctions de surcharges d'opérateurs soient déclarées amies de la classe afin qu'elles puissent accéder aux attributs de la classe. On choisira plutôt de déclarer des accesseurs en lecture pour tous les attributs devant être accessibles et à utiliser ceux-ci dans les fonctions de surcharges d'opérateurs.

Ex. : La surcharge de l'opérateur << pour la classe `Chaine` par la fonction `ostream& operator<<(ostream&, const Chaine&)` [1].

```
class Chaine
{
   public:
      Chaine(char*);
      Chaine(const Chaine&);
      ~Chaine() { delete[] str; }
      char* GetChaine() const { return this->str; }     // méthode constante
      Chaine operator+(const Chaine&);     // surcharge de l'opérateur + (1)
      Chaine operator+(char);              // surcharge de l'opérateur + (2)
      Chaine& operator=(const Chaine&);
   private:
      char* str;
};
...

ostream& operator<<(ostream& os, const Chaine& chn)
{
   os << chn.GetChaine();
   return os;
}

int main(void)
{
   Chaine str1("bonjour!");
   Chaine str2("ca va!");

   cout << str1 << str2 << endl;     // affichage: "bonjour!ca va!"
   return 0;
}
```

Généralement, si l'une des méthodes parmi le *destructeur*, le *constructeur de recopie* ou *la surcharge de l'opérateur d'affectation* est réécrite, c'est que la version « basique » produite par le compilateur ne convient pas.
On prendra alors soin de réécrire impérativement les 2 autres : c'est ce qu'on appelle la *règle des trois*.

Cette règle des trois se retrouve dans la *forme canonique de Coplien*. Il s'agit d'un ensemble minimum de méthodes permettant de manipuler sans problème des instances de toute classe ayant comme caractéristique de posséder au moins 1 attribut alloué dynamiquement, et définie ainsi :

```
Class NomClasse
{
   public:
      NomClasse();                        // constructeur par défaut
      NomClasse(const NomClasse&);        // constructeur de recopie
      ~NomClasse();                       // destructeur (éventuellement virtuel)
      NomClasse& operator=(const NomClasse&);      // surcharge opérateur =
};
```

[1] Cette fonction surcharge la fonction déjà existante `operator<<()` afin que l'opérateur << puisse traiter les objets du type `Chaine`.

5 L'HÉRITAGE DE CLASSES

L'**héritage** de classe, concept fondamental de la POO, définit qu'une nouvelle classe peut être créée à partir d'une classe déjà existante. Cela permet ainsi de préciser, ou mettre à jour une classe.

On distingue :

- l'héritage simple : une classe est créée à partir d'une seule classe ;
- l'héritage multiple : une classe est créée à partir de plusieurs classes.

5.1 L'HÉRITAGE SIMPLE

5.1.1 Principes

L'**héritage simple** désigne le cas où une nouvelle classe est créée à partir d'une et d'une seule classe déjà existante. La nouvelle classe possède alors automatiquement une copie des membres de la classe déjà existante, lesquels peuvent être complétés par de nouveaux.

Dans le cadre d'un héritage où ClasseA existe et permet de créer ClasseB, ClasseA – classe déjà existante – est appelée la *super-classe*, et ClasseB – nouvelle classe – est appelée la *sous-classe*. On parle aussi de **dérivation** de la super-classe en sous-classe ; on dérive la *classe de base*, ClasseA, pour obtenir alors une *classe dérivée*, ClasseB.

Dans la représentation UML, l'héritage est symbolisé par une flèche à tête triangulaire vide qui relie les 2 classes dans le sens sous-classe → super-classe.

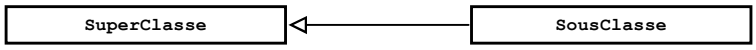

Figure 5.1 : représentation UML d'une relation d'héritage simple

Ex. : Soit une classe qui représente un véhicule, et une autre qui représente une voiture. Une voiture « est une sorte de » véhicule ; dans la relation `Vehicule`/`Voiture`, `Vehicule` est donc la super-classe, et `Voiture` la sous-classe.

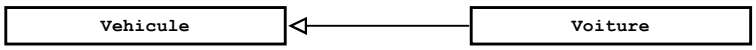

Figure 5.2 : exemple d'un héritage simple entre deux classes

Dans l'héritage simple, une sous-classe ne peut hériter que d'une seule classe. En revanche, une classe peut servir de classe de base pour plusieurs classes dérivées.

Si besoin est, les membres peuvent être redéfinis dans la sous-classe ; ainsi, les méthodes héritées peuvent être redéfinies par surcharge dans la classe dérivée, afin d'être plus adaptées à ce que représente cette classe.

En réalité, l'accès dans la sous-classe aux membres hérités dépend de leur visibilité [1] :

- membre public de la classe de base : devient un membre public de la classe dérivée ;
- membre privé de la classe de base : est inaccessible par la classe dérivée.

L'encapsulation est donc pleinement respectée par l'héritage.

On observe alors un dilemme : comment avoir effectivement une copie des attributs dans la sous-classe sans risque de viol potentiel de l'encapsulation [2] ?

La solution passe par l'utilisation d'un troisième caractère de visibilité, qui est le caractère *protégé* (`protected`) :

- membre protégé de la classe de base : devient un membre protégé de la classe dérivée.

[1] Il s'agit des capacités d'accès et non de la présence des membres : tous les membres hérités sont bien présents, mais certains sont inaccessibles.

[2] Car si un attribut de la classe de base est déclaré `public`, il est alors disponible dans la classe dérivée, mais il l'est de fait aussi pour n'importe quelle autre classe.

Un membre protégé est inaccessible de l'extérieur de la classe (respect de l'encapsulation), mais est disponible pour les classes dérivées (atout de l'héritage).

Dans la représentation UML de la classe, un membre protégé est précisé d'un # (dièse) devant son nom.

Ex. : Soit une classe représentant une voiture et une classe représentant un vélo, dérivées d'une classe `Vehicule`, ayant accès à certains membres hérités de la classe de base tout en respectant l'encapsulation.

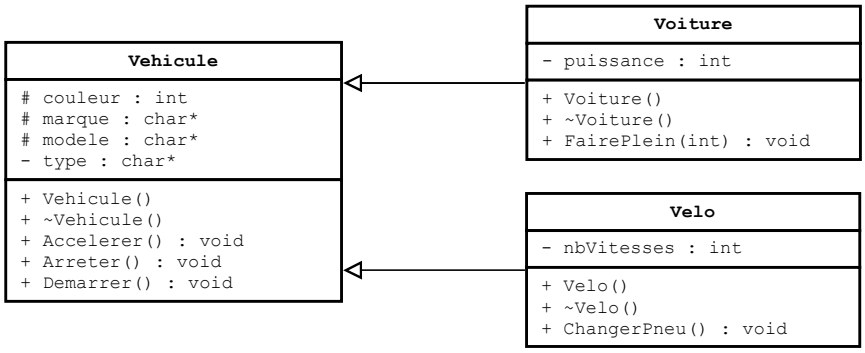

Figure 5.2 : exemple d'utilisation de l'héritage

Les attributs `couleur`, `marque` et `modele` de la classe `Vehicule` seront donc disponibles dans les classes `Voiture` et `Velo`. En revanche, l'attribut `type` (chaîne représentant le type de véhicule : « camion », « voiture », « bateau », etc.) restera indisponible aux classes dérivées.

5.1.2 Implémentation

Pour spécifier qu'une classe dérive d'une autre classe, il faut indiquer, à la déclaration de la classe, le nom de la classe de base, suivant la syntaxe `class SousClasse : SuperClasse { ... };`.

Ex. : La classe `Employe` dérive de la classe `Personne`.

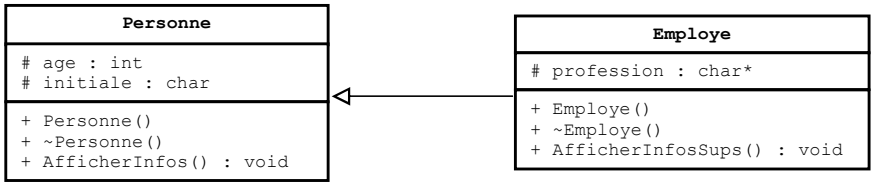

Figure 5.3 : exemple d'implémentation d'un héritage simple

```
class Personne
{
   public:
      Personne();
      ~Personne();
      void AfficherInfos();
   protected:
      int age;
      char initiale;
};
```

```
class Employe : Personne
{
   public:
      Employe();
      ~Employe();
      void AfficherInfosSups();
   protected:
      char* profession;
};
```

5.1.3 Relations entre classe de base et classe dérivée

L'action de dérivation implique plusieurs relations entre la classe de base et la classe dérivée :

- recopie des membres de la classe de base [1], conformément à leur visibilité ;
- appel du constructeur de la classe de base avant l'appel du constructeur de la classe dérivée, afin d'initialiser les attributs hérités ;
- appel du destructeur de la classe de base après l'appel du destructeur de la classe dérivée, afin de réaliser les opérations nécessaires à la destruction propre définies par la classe de base.

[1] Sont exclus de l'héritage : tous les constructeurs (défaut, non par défaut, de recopie) ainsi que la ou les surcharges de l'opérateur d'affectation = (`operator=()`).

Ex. : La classe `Employe` dérive de la classe `Personne`.

```
Personne::Personne()
{
   cout << "Personne()" << endl;
   this->age = 0;
   this->initiale = '?';
}

Personne::~Personne()
{
   cout << "~Personne()" << endl;
}

void Personne::AfficherInfos()
{
   cout << "age " << this->age << endl;
   cout << "initiale " << this->initiale << endl;
}


Employe::Employe()
{
   cout << "Employe()" << endl;
   this->profession = new char[10];
   this->profession = "n.c";
}

Employe::~Employe()
{
   delete this->profession;
   cout << "~Employe()" << endl;
}

void Employe::AfficherInfosSups()
{
   cout << "age " << this->age << endl;
   cout << "initiale " << this->initiale << endl;
   cout << "profession " << this->profession << endl;
}


int main(void)
{
   Employe* emp1;       // déclaration d'un objet Employe (pas de construction)
   emp1 = new Employe;          // affichage: "Personne()" | "Employe()"
   emp1->AfficherInfosSups();   // affichage: "0" | "?" | "n.c"
   delete emp1;                 // affichage: "~Employe()" | "~Personne()"

   return 0;
}
```

C'est donc le constructeur par défaut de la super-classe qui est exécuté.

Dans le cas où la classe de base ne possède pas de constructeur par défaut, ou qu'il est préférable de faire appel à une version surchargée du constructeur (possédant 1 ou plusieurs paramètres), il faut alors spécifier explicitement l'appel au constructeur de la classe de base, en utilisant la syntaxe [1] :

```
SousClasse::SousClasse(...)
: SuperClasse(paramètres)
{ /* code du constructeur */ }
```

Les paramètres peuvent être de type fini ou bien être des variables issues des paramètres d'appel du constructeur de la sous-classe :

```
SousClasse::SousClasse(type_paramètre1 parametre1, type_paramètre2 parametre2, ...)
: SuperClasse(parametre2, ...)
{ /* code du constructeur */ }
```

[1] Cette syntaxe est similaire à celle utilisée lors de la déclation d'une liste d'initialisation (cf. 3.4.2) ; si les 2 concepts demeurent totalement différents, leur usage peut être mélangé sans problème, car servant en réalité la même cause : l'initialisation des attributs de l'objet.

Ex. :
```
class Personne
{
   public:
      Personne(int = 0, char = '?');
      void AfficherInfos();
   protected:
      int age;
      char initiale;
};
...

Personne::Personne(int a, char i)
{
   this->age = a;
   this->initiale = i;
}


class Employe : Personne
{
   public:
      Employe();
      Employe(int, char);
      ~Employe();
      void AfficherInfosSups();
   protected:
      char* profession;
};
...

Employe::Employe()         // appel du constructeur Personne(int, char)
: Personne(1, '-')         // avec des paramètres de valeur finie
{
   this->profession = new char[10];
   this->profession = "n.c";
}

Employe::Employe(int a, char i)    // appel du constructeur Personne(int, char)
: Personne(a, i)                   // avec des paramètres issus des paramètres
{                                  // d'appel du constructeur Employe(int, char)
   this->profession = new char[10];
   this->profession = "n.c";
}


int main(void)
{
   Employe emp2;
   emp2.AfficherInfosSups(); // affichage: "1" | "-" | "n.c"
   Employe emp3(20, 'a');
   emp3.AfficherInfosSups(); // affichage: "20" | "a" | "n.c"
   return 0;
}
```

Là encore, l'usage des valeurs par défaut pour les paramètres des constructeurs permet de faire correspondre plusieurs surcharges à 1 seul constructeur.

Ex. :
```
class Employe : Personne
{
   public:
      Employe(int = 1, char = '-'); // Employe(), Employe(int), Employe(int, char)
      ~Employe();
      void AfficherInfosSups();
   protected:
      char* profession;
};
...
```

```
Employe::Employe(int a, char i)
: Personne(a, i)
{
   this->profession = new char[10];
   this->profession = "n.c";
}
```

Lorsque l'on redéfinit (/re-déclare) un membre [1] dans la classe dérivée, le membre originel n'est pas remplacé mais caché par le membre redéfini [2]. En pratique, on redéfinit rarement un attribut ; en revanche, il est généralement pertinent de redéfinir une méthode, afin de la spécialiser pour la classe dérivée.

Ex. :
```
class Employe : Personne
{
   public:
      Employe(int = 1, char = '-');
      ~Employe();
      void AfficherInfos();   // redéfinition de la méthode
   protected:
      char* profession;
};
...

void Employe::AfficherInfos()
{
   cout << "age " << this->age << endl;
   cout << "initiale " << this->initiale << endl;
   cout << "profession " << this->profession << endl;
}


int main(void)
{
   Employe emp3(20, 'a');
   emp3.AfficherInfos();   // affichage: "20" | "a" | "n.c"

   return 0;
}
```

Néanmoins, les membres hérités, cachés par les membres redéfinis, restent accessibles en utilisant l'opérateur de résolution de portée, suivant la syntaxe `SuperClasse::membre`.

Ex. :
```
void Employe::AfficherInfos()
{
   this->Personne::AfficherInfos();     // appel de AfficherInfos() de Personne
   cout << "profession: " << this->profession << endl;
}
```

5.1.4 Modes de dérivation

Le **mode de dérivation** permet de spécifier les modifications apportées dans la classe dérivée à la visibilité des membres hérités de la classe de base.

On distingue 3 modes de dérivation (/mode d'héritage) :

- public: les membres publics et protégés conservent la même visibilité ;
- protégé : les membres publics deviennent protégés, alors que les membres protégés conservent leur visibilité ;
- privé : les membres publics et protégés deviennent des membres privés.

Dans tous les cas, les membres privés restent inaccessibles.

Le mode de dérivation se spécifie lors de la déclaration de la sous-classe, en utilisant l'un des mots-clefs `public` / `protected` / `private` suivant la syntaxe `class SousClasse : mode_dérivation SuperClasse { ... };`.

Ex. : La classe `Employe` dérive de la classe `Personne` suivant un héritage public.

[1] Dans le cas de la redéfinition d'une méthode, on peut parler de « surcharge ».
[2] Même dans le cas d'une redéfinition d'une méthode avec une méthode de signature différente.

```
class Employe : public Personne
{
   ...
};
```

L'héritage public est la forme la plus courante d'héritage : elle ne dénature pas la visibilité des membres.
Si aucun mode de dérivation n'est précisé, c'est l'héritage privé qui est utilisé [1].

	visibilité dans la classe de base	visibilité dans la classe dérivée
héritage public	public	public
	protected	protected
	private	*inaccessible*
héritage protégé	public	protected
	protected	protected
	private	*inaccessible*
héritage privé	public	private
	protected	private
	private	*inaccessible*

Néanmoins, dans le cas des héritages protégé et privé, il est possible de mentionner des exceptions ; certains membres peuvent ainsi conserver leur visibilité, et ce malgré le mode de dérivation. C'est ce qu'on appelle la **déclaration d'accès**.

Pour ce faire, il faut utiliser l'opérateur de résolution de portée dans la définition de la classe dérivée, en forçant la redéclaration du membre dans la section de visibilité adéquate, suivant la syntaxe `SuperClasse::membre;` (uniquement le nom, pas de modificateurs, paramètres, etc., même pas de parenthèses).

Ex. :
```
class Personne
{
   public:
      Personne(int = 0, char = '?');
      void AfficherInfos();
   protected:
      int age;
      char initiale;
};

class Employe : private Personne     // héritage privé : tous les membres publics
{                                    // et protégés hérités obtiennent par défaut
   public:                           // une visibilité privée
      Employe(int = 1, char = '-');
      ~Employe();
      Personne::AfficherInfos;     // redéclaration de la méthode AfficherInfos()
   protected:                      // pour conserver une visibilité publique
      char* profession;
      Personne::age;             // redéclaration de l'attribut age pour conserver
};                               // une visibilité protégée
// l'attribut initiale, non-redéclaré, reste privé
```

Nb : La déclaration d'accès permet uniquement de conserver la visibilité originelle, mais en aucun cas de la modifier (l'augmenter ou la réduire).

5.2 LE POLYMORPHISME

Le **polymorphisme** [2] désigne la capacité pour une méthode à adopter un comportement différent selon l'objet auquel elle est appliquée ou bien le nombre ou le type de paramètres avec lesquels elle est appelée.

5.2.1 Hiérarchie des classes

Tout l'intérêt de l'héritage réside dans les corrélations qui existent entre différentes classes. Avec un ensemble de classes liées par des relations d'héritage, parfois successifs, une **hiérarchie de classes** peut ainsi être définie.

[1] C'est donc celui qui était utilisé dans ce chapitre jusqu'ici.
[2] Mot d'étymologie grecque signifiant « peut prendre plusieurs formes ».

Lorsqu'une modification est alors apportée à un membre non privé d'une classe, toutes les classes dérivées subissent automatiquement la modification.

Ex. : Une hiérarchie de classes représentant l'organisation des personnes dans un contexte professionnel.

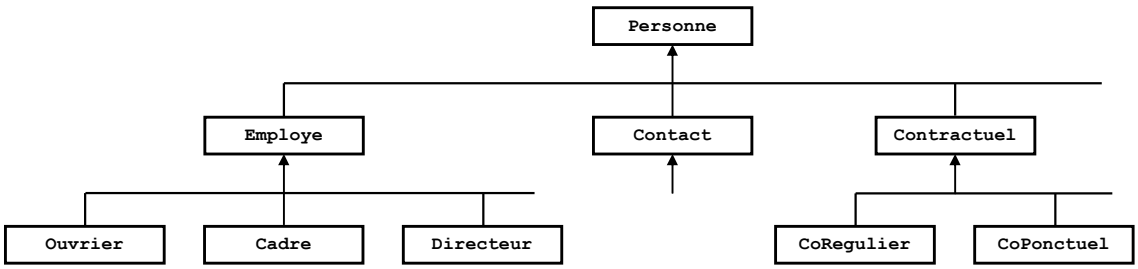

Figure 5.4 : exemple de hiérarchie de classes

Ainsi, si on rajoute l'attribut protégé `email` à la classe `Personne`, et que tous les héritages entre les classes sont publics, cet attribut sera automatiquement disponible dans toutes ses sous-classes.

L'héritage de classe implique une forte cohésion entre les classes d'une même hiérarchie. Ainsi, si on considère la classe `Cadre`, on peut dire que `Cadre` est une sorte d'`Employe`, alors que `Employe` est elle-même une sorte de `Personne` ; par conséquent, `Cadre` est aussi une sorte de `Personne`. En revanche, `Personne` n'est pas nécessairement `Employe`, `Directeur`, ou `Contact`.
Ces relations permettent alors de mettre en évidence des conversions possibles entre classes.

5.2.2 Conversion vers une classe de base

Une hiérarchie de classes implique que l'on peut considérer (/ « réduire ») un objet d'une classe dérivée comme *une sorte d'*objet de la classe de base (niveau supérieur direct ou indirect) ; en revanche, il est impossible de considérer (/ « étendre ») un objet d'une classe de base comme un objet de l'une de ses classes dérivées.

Autrement dit, la **conversion vers une classe de base** d'une instance d'une classe dérivée est possible. Il s'en suit que différentes écritures mêlant classe de base et classe dérivée d'une même hiérarchie de classe sont légitimes.

Ex. : La hiérarchie de classes de personnes dans un contexte professionnel permet d'écrire :

```
Personne unePersonne1;       // instanciation statique d'un objet Personne en
Ouvrier unOuvrier1;          // utilisant un objet Ouvrier déjà créé
unePersonne1 = unOuvrier1;

Personne* unePersonne2;      // instanciation dynamique d'un objet Personne en
Cadre* unCadre2;             // utilisant un objet Cadre déjà créé
unCadre2 = new Cadre();
unePersonne2 = unCadre2;

Personne* unePersonne3 = new Cadre();  // idem en écriture restreinte

Personne* unePersonne4;           // instanciation dynamique d'un objet Personne
Contractuel unContractuel4;       // en utilisant l'adresse d'un objet
unePersonne4 = &unContractuel4;   // Contractuel déjà créé

void uneFonction5(Personne p)     // uneFonction5 attend un objet Personne : usage
{                                 // implicite du constructeur par défaut
   p.AfficherInfos();
}
...
Employe unEmploye5;               // à l'appel de la fonction, l'objet Personne
uneFonction5(unEmploye5);         // est construit en utilisant un objet Employe

void uneFonction6(Personne* p)
{
   p->AfficherInfos();
}
...
Directeur unDirecteur6;
uneFonction6(&unDirecteur6);
```

Nb : Ces différentes « conversions » sont possibles uniquement si les héritages mis en jeu sont publics [1].

Ainsi, là où on attend une instance de la classe de base, on peut donc utiliser une instance de la classe dérivée, car la classe dérivée contient au moins les membres de la classe de base ; l'inverse est en revanche impossible, certaines informations manquent pour construire correctement l'objet de la classe dérivée à partir des données d'un objet de la classe de base.

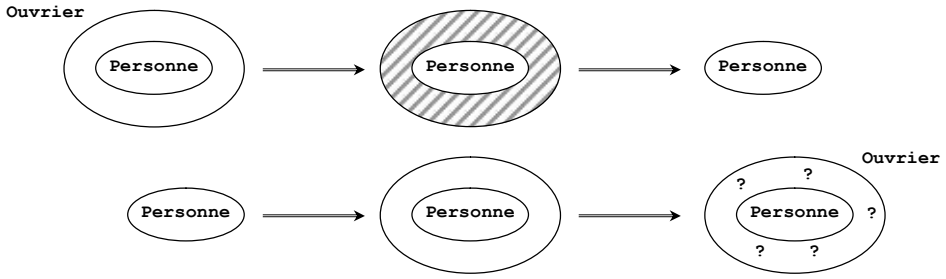

Figure 5.5 : exemple de conversion de type dans une hiérarchie de classes

Plus précisément, dans le cas d'une déclaration statique, la conversion entraîne effectivement une perte d'informations ; dans le cas d'une déclaration dynamique (pointeur ou référence), la conversion ne fait pas perdre d'informations, mais l'instance ne permet plus d'accéder à tous les membres.

5.2.3 Généralisation

La hiérarchie d'un ensemble de classes autorise donc les déclarations suivantes :

```
SousClasse objSousClasse;
SuperClasse objSuperClasse = objSousClasse;
SuperClasse* ptrSuperClasse = &objSousClasse;
SuperClasse& refSuperClasse = objSousClasse;
```

La question peut se poser de savoir de quel type est alors effectivement chacune des instances déclarées. Pour la déclaration statique (`objSuperClasse`), on sait que la conversion implique la perte des informations superflues. En revanche, dans le cas de la déclaration dynamique, le doute est permis.

C'est ici qu'intervient la distinction, pour une instance, de *typage statique* ou de *typage dynamique* :

- **type statique** : type utilisé à la déclaration ;
- **type dynamique** : type utilisé à la construction de l'instance [2].

Ainsi, `ptrSuperClasse` et `refSuperClasse` sont du type statique `SuperClasse`, mais sont du type dynamique `SousClasse`.

Le typage dynamique conduit à la notion de **généralisation** [3] lorsqu'un ensemble d'instances différentes dérivent tous de la même classe de base. Cette propriété permet de déclarer des pointeurs d'une même classe et de construire chacun d'entre eux comme une instance d'une classe dérivée différente.

Cela se révèle pratique pour regrouper des traitements effectués sur divers objets de type statique identique, mais de type dynamique différent.

Ex. : Usage de la généralisation dans une hiérarchie de classes.

[1] La classe de base doit conserver des membres publics accessibles.

[2] Sachant que l'instance d'un objet peut être construite plusieurs fois – donc de manière différente – durant une seule et même exécution du programme.

[3] L'inverse étant la *spécialisation* : on dérive une classe en une sous-classe pour la spécialiser aux besoins.

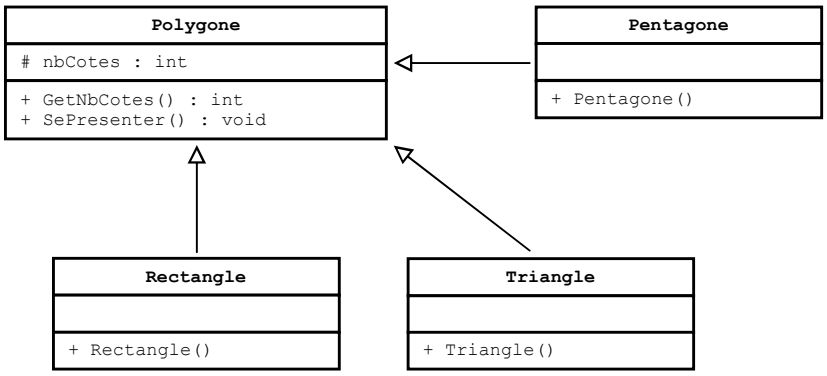

Figure 5.6 : exemple de mise en œuvre de la généralisation

```
class Polygone
{
   public:
      int GetNbCotes() { return this->nbCotes; }
      void SePresenter() { cout << "polygone" << endl; }
   protected:
      int nbCotes;
};

class Triangle : public Polygone
{
   public:
      Triangle() { this->nbCotes = 3; }
};

class Rectangle : public Polygone
{
   public:
      Rectangle() { this->nbCotes = 4; }
};

class Pentagone : public Polygone
{
   public:
      Pentagone() { this->nbCotes = 5; }
};


Polygone* figures[4];      // liste des figures (ensemble d'objets ayant
                           // même type statique Polygone*)
int main(void)
{
   figures[0] = new Polygone();    // instanciation d'objets différents : ils ont
   figures[1] = new Triangle();    // un type dynamique différent du type statique
   figures[2] = new Rectangle();   // (ajout d'un triangle, d'un rectangle et d'un
   figures[3] = new Pentagone();   // pentagone à la liste des figures)

   for ( int i=0 ; i<4 ; i++ ) {
      cout << figures[i]->GetNbCotes() << endl; // affichage: "0" | "3" | "4" | "5"
   }

   return 0;
}
```

Malgré tout, lors de la redéfinition de membres, c'est le type statique qui prime lorsqu'une instance a un type dynamique différent ; le type est effectivement déterminé à la compilation (type statique) et non à l'exécution (type dynamique) [1]. Ainsi, c'est la « version » de la méthode correspondant au type statique qui est appelée, même si l'instance a un type dynamique différent.

Ex. : Les limites de la généralisation dans une hiérarchie de classes.

[1] On parle de « ligature statique » : caractéristique qui provient du souci de compatibilité avec le langage C dont est issu le langage C++.

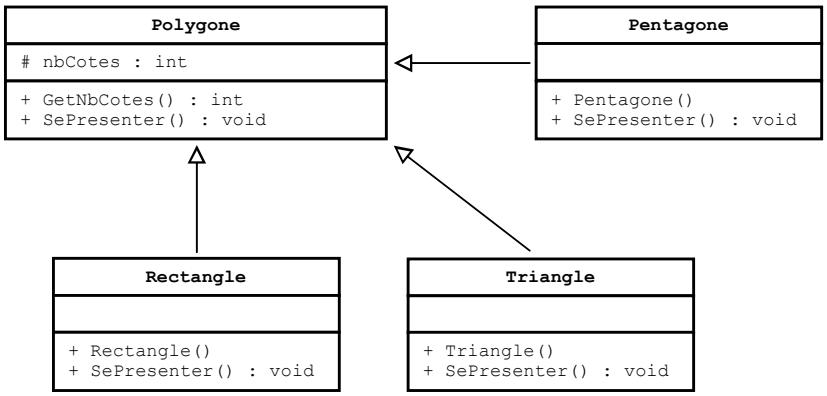

Figure 5.7 : exemple de rédéfinition d'un membre dans une relation de généralisation

```
...


class Triangle : public Polygone
{
   public:
      Triangle() { this->nbCotes = 3; }
      void SePresenter() { cout << "triangle" << endl; }
};

class Rectangle : public Polygone
{
   public:
      Rectangle() { this->nbCotes = 4; }
      void SePresenter() { cout << "rectangle" << endl; }
};

class Pentagone : public Polygone
{
   public:
      Pentagone() { this->nbCotes = 5; }
      void SePresenter() { cout << "pentagone" << endl; }
};


Polygone* figures[4];

int main(void)
{
   figures[0] = new Polygone();
   figures[1] = new Triangle();
   figures[2] = new Rectangle();
   figures[3] = new Pentagone();

   for ( int i=0 ; i<4 ; i++ ) {
      figures[i]->SePresenter();                    // affichage: "polygone 0" |
      cout << figures[i]->GetNbCotes() << endl;     // "polygone 3" | "polygone 4" |
   }                                                // "polygone 5" |

   return 0;
}
```

Cette caractéristique paraît bien peu pratique, et limite fortement l'intérêt de l'héritage et de la généralisation [1]. Mais, si on reste bloqué pour les attributs, il existe cependant une solution concernant les méthodes ; il suffit de mettre en œuvre les méthodes virtuelles.

[1] On imagine très bien un traitement identique à appliquer à un ensemble d'objets de types dynamiques différents.

5.2.4 Méthodes virtuelles

Une méthode déclarée comme **virtuelle** dans la classe de base se fonde sur le type dynamique pour être exécutée. C'est-à-dire que pour une instance ayant comme type dynamique une classe dérivée et comme type statique la classe de base [1], c'est la redéfinition de la méthode qui sera appelée si celle-ci a été surchargée. L'exécution s'adapte ainsi exactement à ce qu'est l'objet au moment de l'exécution du programme.

Pour déclarer une méthode virtuelle, on utilise le modificateur `virtual` dans la définition de la classe de base suivant la syntaxe `virtual type_méthode NomMethode(type_paramètre_1, type_paramètre_2, ...)`.

Ex. : Usage des méthodes virtuelles dans une hiérarchie de classes.

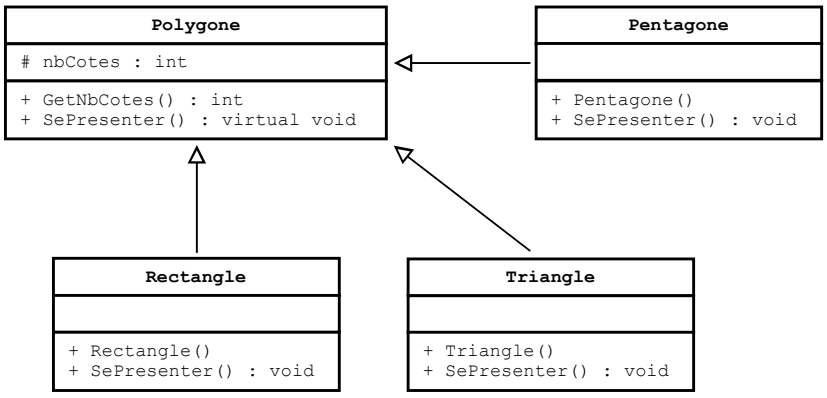

Figure 5.8 : exemple de mise en œuvre d'une méthode virtuelle

```
class Polygone
{
   public:
      int GetNbCotes() { return this->nbCotes; }
      virtual void SePresenter() { cout << "polygone" << endl; }   // virtuelle
   protected:
      int nbCotes;
};

...

int main(void)
{
   figures[0] = new Polygone();
   figures[1] = new Triangle();
   figures[2] = new Rectangle();
   figures[3] = new Pentagone();

   for ( int i=0 ; i<4 ; i++ ) {
      figures[i]->SePresenter();                    // affichage: "polygone 0" |
      cout << figures[i]->GetNbCotes() << endl;     // "triangle 3" |
   }                                                // "rectangle 4" |
                                                    // "pentagone 5"
   return 0;
}
```

Une déclaration virtuelle n'implique pas une redéfinition obligatoire de la méthode dans les classes dérivées, mais est significative d'une souplesse potentielle pour les futures classes dérivées.

Cette caractéristique se propage aussi bien pour les classes dérivées directes que les classes dérivées indirectes. Ainsi, dès qu'une méthode est déclarée virtuelle, elle l'est pour toute la partie « inférieure » de la hiérarchie de classes.

Ex. : Propagation de la qualité « virtuelle » d'une méthode aux sous-classes.

[1] Ceci exclut donc les instances statiques de la classe de base car elles ne peuvent être « construites » que comme objets de la classe de base, et pas comme objets d'une classe dérivée.

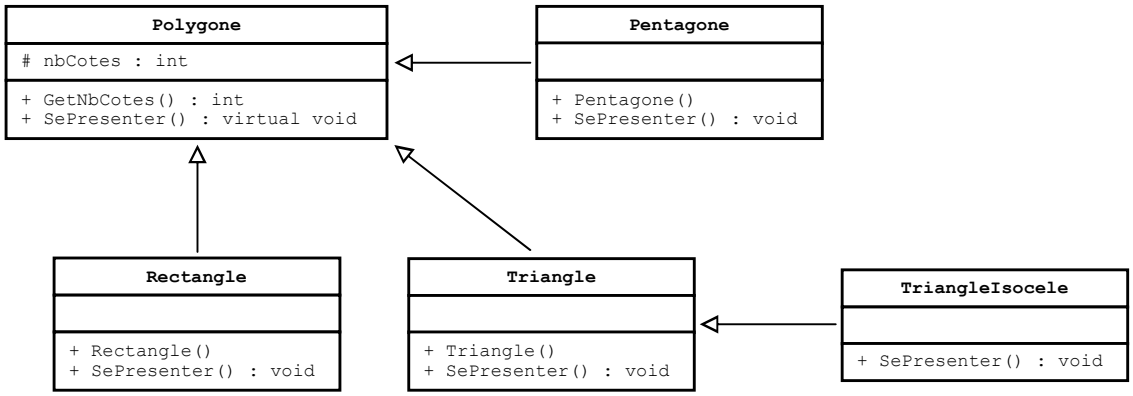

Figure 5.9 : exemple de propagation d'une déclaration virtuelle

```
...

class Triangle : public Polygone // aucun changement à la classe Triangle
{
   public:
      Triangle() { this->nbCotes = 3; }
      void SePresenter() { cout << "triangle" << endl; }   // déclaration virtuelle
};                                                         // inutile : hérité de
                                                           // Polygone
class TriangleIsocele : public Triangle
{
   public:
      void SePresenter() { cout << "isocele" << endl; }
};

...

int main(void)
{
   figures[0] = new Polygone();
   figures[1] = new TriangleIsocele();
   figures[2] = new Rectangle();
   figures[3] = new Pentagone();

   for ( int i=0 ; i<4 ; i++ ) {
      figures[i]->SePresenter();                    // affichage: "polygone 0" |
      cout << figures[i]->GetNbCotes() << endl;     // "isocele 3" | "rectangle 4" |
   }                                                // "pentagone 5" |

   return 0;
}
```

Attention : Lors de l'utilisation d'une méthode virtuelle, le destructeur, s'il est redéfini, doit lui aussi être déclaré virtuel dans la classe de base, afin de s'assurer que ce soit le destructeur de la classe héritée qui soit exécuté à la destruction d'un objet.

5.2.5 Méthodes virtuelles pures et classes abstraites

Lorsqu'une méthode virtuelle est déclarée dans une classe de base, mais que son implémentation ne peut pas être donnée ou n'a pas de sens alors qu'elle en a dans les classes dérivées, on peut alors déclarer la méthode comme étant virtuelle pure.

Une **méthode virtuelle pure** est une méthode définissant un concept dans la classe de base, mais dont l'implémentation n'est pas donnée ; en revanche, l'implémentation de la méthode devra obligatoirement être fournie par toute classe dérivée.

On définit une méthode virtuelle pure en rajoutant `= 0` dans la définition de la classe suivant la syntaxe `virtual type_méthode NomMethode(type_paramètre_1, type_paramètre_2, ...) = 0`.

Dans la représentation UML d'une classe, le nom d'une méthode virtuelle pure est écrite en italique.

Ex. : Une classe abstraite dans une hiérarchie de classes.

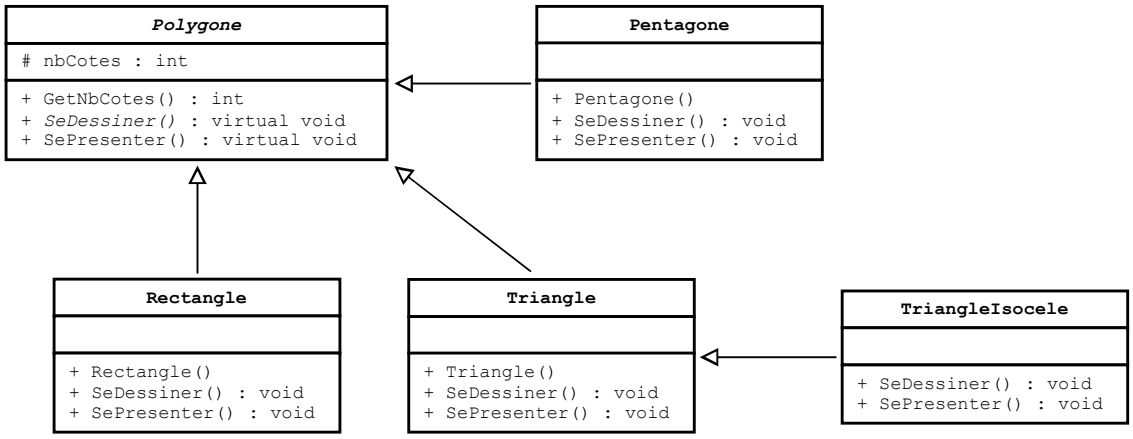

Figure 5.10 : exemple de mise en œuvre d'une méthode virtuelle pure et d'une classe abstraite

```
class Polygone
{
   public:
      int GetNbCotes() { return this->nbCotes; }
      virtual void SeDessiner() = 0;     // pas d'implémentation pour cette méthode
      virtual void SePresenter() { cout << "polygone" << endl; }
   protected:
      int nbCotes;
};

class Triangle : public Polygone
{
   public:
      Triangle() { this->nbCotes = 3; }
      void SeDessiner() { cout << "dessin triangle" << endl; }
      void SePresenter() { cout << "triangle" << endl; }
};

class Rectangle : public Polygone
{
   public:
      Rectangle() { this->nbCotes = 4; }
      void SeDessiner() { cout << "dessin rectangle" << endl; }
      void SePresenter() { cout << "rectangle" << endl; }
};

class Pentagone : public Polygone
{
   public:
      Pentagone() { this->nbCotes = 5; }
      void SeDessiner() { cout << "dessin pentagone" << endl; }
      void SePresenter() { cout << "pentagone" << endl; }
};
```

Définir une méthode virtuelle pure permet de s'assurer que celle-ci sera assurément surchargée dans les classes dérivées, ce qui s'avère nécessaire lorsque la méthode représente un concept générique inutilisable à très haut niveau.

Lorsqu'une méthode est déclarée virtuelle pure, son implémentation n'a donc pas de sens [1] ; en conséquence, toute classe ayant au moins 1 méthode virtuelle pure ne peut être construite [2] et on dit alors que la classe est **abstraite**. La classe est destinée à demeurer une classe « concept », soit donc une classe servant de modèle pour d'autres classes.

Dans la représentation UML, le nom d'une classe abstraite est écrite en italique.

Si une méthode virtuelle pure héritée ne peut être implémentée complètement dans la classe dérivée, cela signifie qu'elle doit être aussi déclarée virtuelle pure, et de fait, la classe qui la contient est abstraite.

Nb : Par extension, on peut parler de *méthode abstraite* dans le cas d'une méthode virtuelle pure.

[1] Elle peut cependant être conservée, mais elle n'est pas prise en compte par le compilateur.
[2] Car il existe au moins une méthode n'ayant pas d'implémentation.

5.3 L'HÉRITAGE MULTIPLE

5.3.1 Principes

L'**héritage multiple** désigne le cas où une nouvelle classe est créée à partir de plusieurs classes déjà existantes. La nouvelle classe hérite donc de tous les membres de chacune des super-classes, au regard du mode d'héritage utilisé pour chacune, et de la visibilité de chacun des membres.

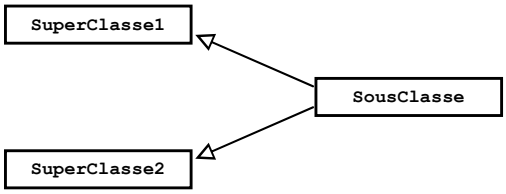

Figure 5.11 : représentation UML d'un héritage multiple

5.3.2 Implémentation

Pour indiquer qu'une classe dérive de plusieurs classes, on procède comme pour l'héritage simple suivant la syntaxe `class SousClasse : mode_dériv1 SuperClasse1, mode_dériv2 SuperClasse2, ... { ... };`.

Ex. : La classe `DelegueSyndical` dérive des classes `Employe` et `Syndicaliste`.

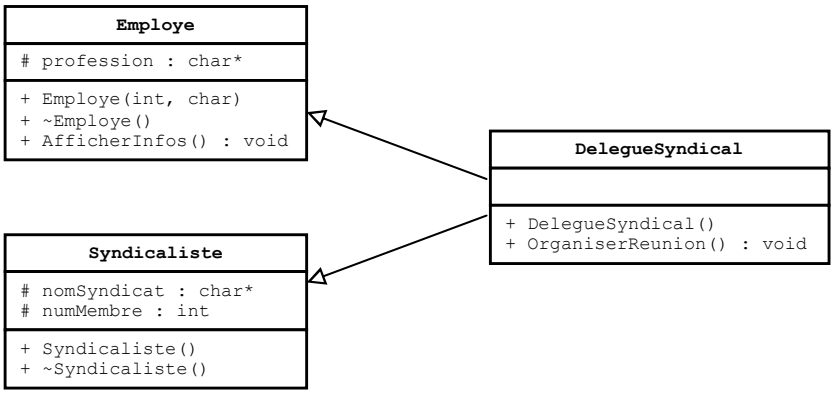

Figure 5.12 : exemple d'héritage multiple

```
class Employe : public Personne
{
   public:
      Employe(int = 1, char = '-');
      ~Employe();
      void AfficherInfos();
   protected:
      char* profession;
};
...

class Syndicaliste
{
   public:
      Syndicaliste();
      ~Syndicaliste();
   protected:
      char* nomSyndicat;
      int numMembre;
};
...
```

```
class DelegueSyndical : public Employe, public Syndicaliste
{
   public:
      DelegueSyndical();
      void OrganiserReunion();
};
...
```

5.3.3 Relations entre classes de base et classe dérivée

L'action de dérivation implique plusieurs relations entre les membres des classes de base et de la classe dérivée :

- recopie des membres de chacune des classes de base, conformément à leur visibilité ;
- appel des constructeurs de chacune des classes de base, dans l'ordre des déclarations d'héritage, avant l'appel du constructeur de la classe dérivée (`SuperClasse1()`, puis `SuperClasse2()`, puis `SousClasse()`), afin d'initialiser les attributs hérités ;
- appel des destructeurs de chacune des classes de base, dans l'ordre inverse des déclarations d'héritage, après l'appel du destructeur de la classe dérivée (`~SousClasse()`, puis `~SuperClasse2()`, puis `~SuperClasse1()`), afin de réaliser les opérations nécessaires à la destruction des éléments hérités et qui sont définies dans les super-classes.

Si rien n'est explicitement spécifié, ce sont les constructeurs par défaut des classes de base qui sont appelés.

Dans le cas où une ou plusieurs classes de base ne possèdent pas de constructeur par défaut, ou qu'il est préférable de faire appel à une version surchargée du constructeur, il faut alors spécifier explicitement l'appel aux constructeurs des classes de base en utilisant la syntaxe :

```
SousClasse::SousClasse(...)
: SuperClasse1(paramètres)
, SuperClasse2(paramètres)
, ...
{ /* code du constructeur */ }
```

5.3.4 Héritage virtuel

La sous-classe hérite des membres de chacune de ses super-classes. Dans le cas où plusieurs super-classes héritent elles-mêmes d'une même autre classe, on se trouve face à un problème : certains membres se retrouvent « en double » dans la sous-classe.

Ex. : La classe `Syndicaliste` dérive en réalité de la classe `Personne` ; la classe `Employe` dérive elle aussi de la classe `Personne`. La classe `DelegueSyndical`, qui dérive des classes `Employe` et `Syndicaliste`, retrouve donc des doublons de membres de `Personne`.

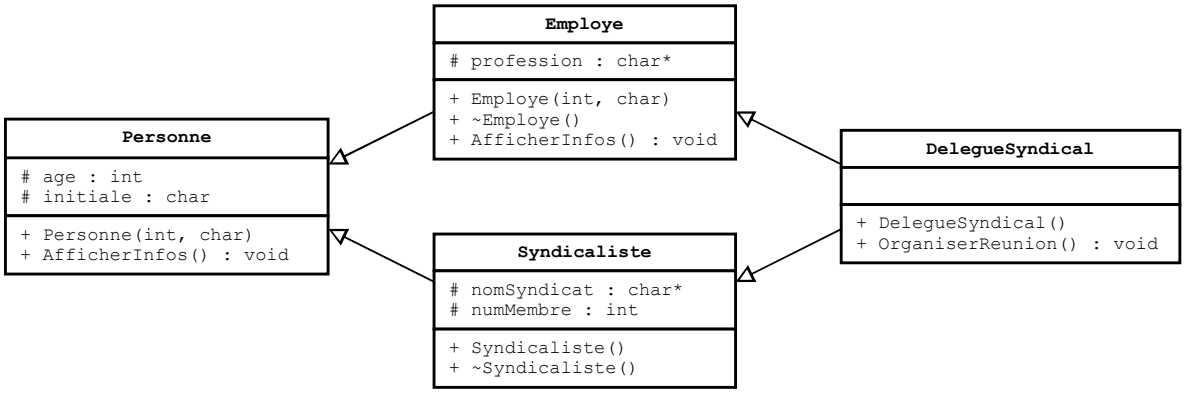

Figure 5.13 : exemple d'un cas de conflit dans un héritage multiple

```
class Personne
{
   public:
      Personne(int = 0, char = '?');
      void AfficherInfos();
   protected:
      int age;
      char initiale;
};
...

class Employe : public Personne
{
   public:
      Employe(int = 1, char = '-');
      ~Employe();
      void AfficherInfos();
   protected:
      char* profession;
};
...

class Syndicaliste : public Personne
{
   public:
      Syndicaliste();
      ~Syndicaliste();
   protected:
      char* nomSyndicat;
      int numMembre;
};
...


class DelegueSyndical : public Employe, public Syndicaliste
{
   public:
      DelegueSyndical();
      void OrganiserReunion();
};
...


int main(void)
{
   DelegueSyndical dsynd1;
   dsynd1.AfficherInfos(); // ambiguïté : s'agit-il de Employe::AfficherInfos() ou
                           // de Syndicaliste::AfficherInfos() (hérité directement
   return 0;               // de Personne::AfficherInfos()) ?
}
```

Les problèmes de conflits de membres peuvent être néanmoins résolus ponctuellement en utilisant l'opérateur de résolution de portée.

```
int main(void)
{
   DelegueSyndical dsynd1;
   dsynd1.Employe::AfficherInfos(); // c'est la version de Employe qui est appelée

   return 0;
}
```

Cependant, cette solution demeure insatisfaisante : l'espace mémoire occupé par un objet `DelegueSyndical` n'est pas optimisé, et celui-ci possède 2 noms et 2 initiales, ce qui manque de cohérence ; de plus, deux appels au constructeur de la classe `Personne` sont réalisés, d'abord `Personne(1, '-')` via le constructeur `Employe()`, puis `Personne()` via le constructeur `Syndicaliste()`.

Il est alors généralement préférable de mettre en place un **héritage virtuel** – aucun lien direct avec les méthodes virtuelles et les classes abstraites – qui permet de s'assurer de n'avoir qu'une seule occurrence des membres redondants hérités.

On définit un héritage virtuel en utilisant le modificateur `virtual` dans la déclaration d'héritage suivant la syntaxe `class SousClasse : virtual mode_dérivation SuperClasse {...};`.

Nb : L'héritage virtuel n'agit pas pour la classe elle-même, mais uniquement pour les classes descendantes de la classe qui porte la marque d'héritage virtuel. En effet, celles-ci ne peuvent pas connaître les origines de leurs classe de base et en conséquence ne peuvent détecter une situation de conflit d'héritage de membres.

Ex. : Pour ne pas avoir de membres redondants dans la classe `DelegueSyndical`, il faut préciser que les classes `Employe` et `Syndicaliste` héritent virtuellement de `Personne`.

```
class Employe : virtual public Personne // héritage virtuel
{
   ...
};
...

class Syndicaliste : virtual public Personne // héritage virtuel
{
   ...
};
...


class DelegueSyndical : public Employe, public Syndicaliste
{
   ...
};
...


int main(void)
{
   DelegueSyndical dsynd1;
   dsynd1.AfficherInfos(); // affichage: "age 0" | "initiale ?" | "profession n.c"

   return 0;
}
```

Un seul appel au constructeur de la classe Personne est réalisé : `Personne()` ; même s'il est bien fait appel à `Employe()` puis `Syndicaliste()`.

L'héritage multiple reste peu utilisé, car il est susceptible de générer des conflits, qui ne sont pas toujours facilement résolvables.

Les langages de programmation orientée objet plus récents, comme Java ou C#, n'autorisent pas l'héritage multiple pour ces raisons, et préfèrent mettre en œuvre le mécanisme des interfaces [1].

[1] Notion désignant dans ce cas un type de classe spécifique pouvant être définie ainsi : une interface est une classe abstraite dont toutes les méthodes sont virtuelles pures.

6 LA GÉNÉRICITÉ

6.1 PRINCIPES ET DÉFINITIONS

La **généricité** désigne le principe de définition d'un modèle de fonction, de méthode ou de classe, hors de toute considération de type ou de taille (cas d'un tableau). Cela permet ainsi de travailler avec des données génériques, id est pouvant correspondre à n'importe quel type (types primitifs, complexes ou classes), en fonction de ses besoins.

Pour définir un paramètre générique, on utilise le mot-clef `class`, quelque soit le type futur qui sera associé à ce paramètre [1], suivant la déclaration `class NomParametreGenerique`.
Le nom d'un paramètre générique suit les mêmes règles que les noms de variables, fonctions, attributs et méthodes.

Pour définir un modèle – qu'il s'agisse de fonction, de méthode ou de classe – il faut donc déclarer un ou plusieurs paramètres génériques juste avant la déclaration du modèle, en utilisant le mot-clef `template` [2] suivant la syntaxe :

```
template <class NomParametreGenerique1, class NomParametreGenerique2, ...>
déclaration_modèle { ... }
```

Une fois le modèle complètement défini, il peut être utilisé. Dans le cadre de chaque utilisation, il faut alors préciser le type exact que doit prendre chaque paramètre générique, suivant la syntaxe :

```
nom_modèle<type_paramètre_générique1, type_paramètre_générique2, ...>
```

On parle en ce cas d'*instanciation du modèle générique* et les différentes utilisations du modèle avec divers types de paramètres génériques constituent donc différentes instanciations du modèle.

La généricité peut être appliquée à une fonction, une méthode ou une classe. C'est le compilateur qui se charge de créer toutes les méthodes, fonctions ou classes nécessaires, à partir du modèle défini et des utilisations de ce modèle dans le programme compilé. Un modèle non instancié (/ non utilisé) n'est donc pas compilé et est absent du programme final.

6.2 LES MÉTHODES ET FONCTIONS GÉNÉRIQUES

Définir une **fonction / méthode générique** permet d'appliquer un même traitement à des variables ou objets de type non fixé au départ, soit donc applicable à tout type de données (pour peu que le traitement appliqué ait du sens).

Ex. :
```
template <class T> // déclaration d'un paramètre générique nommé T
type_fonction nomFonction(T nomParametre)
{
   ...
}
```

Nb : Le paramètre générique doit impérativement faire partie de la signature de la fonction [3] afin que diverses surcharges différenciées puissent par la suite être générées automatiquement.

La fonction / méthode peut alors être instanciée (/utilisée) pour n'importe quel type de variables ou objets.

[1] Il serait d'ailleurs bien difficile de prévoir les utilisations futures du modèle.
[2] Template (eng) ≡ modèle (fr) ; on peut parler aussi de *patron*, ou *gabarit*.
[3] Et pas du prototype, ce qui exclut donc le type de retour.

Ex. : Soit une fonction permettant d'afficher le contenu d'un tableau, ainsi qu'une fonction permettant d'en trier le contenu.

```
template <class T>
void afficher(T* tab, int taille)
{
   for ( int i=0 ; i<taille ; i++ )
      cout << tab[i] << " ";
   cout << endl;
}

template <class T>
void trier(T* tab, int taille)
{
   int indmin;
   T tmp;

   for ( int i=0 ; i<taille ; i++ ) {
      indmin = i;
      for ( int j=i+1 ; j<taille ; j++ )
         if ( tab[indmin] > tab[j] ) indmin = j;

      tmp = tab[i];
      tab[i] = tab[indmin];
      tab[indmin] = tmp;
   }
}


int main(void)
{
   int tab1[5] = {12, -5, 3, 34, 5};
   float tab2[3] = {8.544, 7.2, -87.1};
   char tab3[10] = "bonjour";

   trier<int>(tab1, 5);
   trier<float>(tab2, 3);
   trier<char>(tab3, 7);

   afficher<int>(tab1, 5); // affichage: "-5 3 5 12 34"
   afficher<float>(tab2, 3); // affichage: "-87.1 7.2 8.544"
   afficher<char>(tab3, 7); // affichage: "b j n o o r u"

   return 0;
}
```

Le compilateur crée donc ici 3 versions de la fonction `afficher()`, pour `T` de type `int`, `float` et `char` :

- `void afficher(int* tab, int taille)` ;
- `void afficher(float* tab, int taille)` ;
- `void afficher(char* tab, int taille)`.

ainsi que 3 versions de la fonction `trier()`, pour `T` de type `int`, `float` et `char` :

- `void trier(int* tab, int taille)` ;
- `void trier(float* tab, int taille)` ;
- `void trier(char* tab, int taille)`.

Si le type du paramètre générique peut être déduit sans équivoque par rapport au paramètre fourni dans l'appel de la fonction/méthode, il peut être omis.

Ex. :
```
template <class X>
X min(X val1, X val2)
{
   if ( val1 < val2 ) return val1;
   else return val2;
}
```

```
int main(void)
{
   cout << min(4, -5) << endl;        // X est un int
   cout << min(-54.3, -5) << endl;    // X est un double
}
```

Les fonctions et méthodes génériques peuvent être surchargées par des fonctions et méthodes non-génériques, ce qui permet ainsi de préciser le comportement de la fonction / méthode pour un cas particulier de type de variables ou d'objets.

6.3 LES CLASSES GÉNÉRIQUES

Les **classes génériques**, appelées aussi *classes paramétrables*, permettent de définir un modèle de classes [1] pouvant être appliqué avec divers types de variables ou d'objets.

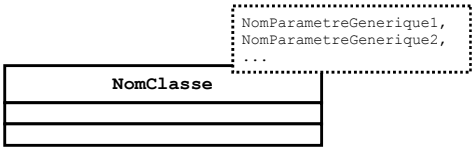

Figure 6.1 : représentation d'une classe générique

```
template <class NomParametreGenerique1, class NomParametreGenerique2, ...>
class NomClasse
{
   ...
};
```

Les différentes méthodes de la classe peuvent alors utiliser le ou les paramètres génériques ainsi définis.

Ex. : Une classe générique `ListeSimple` permettant de gérer une liste de variables ou d'objets quelconques.

```
template <class T>
class ListeSimple
{
   public:
      ListeSimple(int);
      ~ListeSimple();
      void AddElement(const T&);
      void Lister();
   private:
      int nbElements;
      int taille;
      T* elements;
};
```

L'implémentation des méthodes fait partie intégrante du modèle de classes. Chaque définition de méthode, qu'elle utilise le paramètre générique ou pas, doit donc spécifier à nouveau les paramètres génériques suivant la syntaxe :

```
template <class NomParametreGenerique1, class NomParametreGenerique2, ...>
NomClasse<NomParametreGenerique1, NomParametreGenerique2, ...>::NomMethode(...) {...}
```

Ex. :

```
template <class T>
ListeSimple<T>::ListeSimple(int t)
{
   this->nbElements = 0;
   this->taille = t;
   this->elements = new T[this->taille];  // appel du constructeur par défaut de T
}

template <class T>
ListeSimple<T>::~ListeSimple()
{
   delete[] this->elements;
}
```

[1] Une classe étant déjà elle-même un modèle, on parle aussi de *méta-modèle* pour les classes génériques.

```
template <class T>
void ListeSimple<T>::AddElement(const T& nouvelElement)
{
   if ( this->nbElements < this->taille )
      this->elements[this->nbElements++] = nouvelElement;  // appel de l'opérateur
}                                                          // = sur des objets T

template <class T>
void ListeSimple<T>::Lister()
{
   for ( int i=0 ; i < this->nbElements ; i++ )
      cout << this->elements[i] << endl; // appel de l'opérateur << sur un objet T
}
```

La classe générique ainsi définie peut donc être instanciée (/utilisée) avec tout type de variables ou d'objets. Lorsque le type des paramètres génériques est parfaitement défini, on parle alors de *classe paramétrée*.

Ex. : Utilisation de la classe générique `ListeSimple`.

```
int main(void)
{
   ListeSimple<int> data(10);
   data.AddElement(2);
   data.AddElement(4);
   data.AddElement(6);
   data.Lister();

   ListeSimple<Personne> groupe(5);
   groupe.AddElement(Personne(30, "joe"));
   groupe.AddElement(Personne(22, "bob"));
   groupe.AddElement(Personne(28, "ken"));
   groupe.Lister();

   return 0;
}
```

Nb : L'usage de la généricité avec des objets imposent que leur classe puisse se conformer au modèle et se comporter sans ambiguïté. Cela peut pré-supposer par exemple que la classe possède un constructeur par défaut, la surcharge d'opérateurs tels que =, +, <<, ==, !=, etc.

Ex. : L'instanciation de `ListeSimple` avec la classe `Personne` impose pour celle-ci :

- l'existence du constructeur par défaut `Personne()` (utilisé dans le constructeur `ListeSimple()`) ;
- la réécriture de la surcharge de l'opérateur = (utilisé dans `AddElement()`), celle créée automatiquement par le compilateur n'étant pas correcte ;
- l'écriture de la surcharge de l'opérateur << (utilisé dans `Lister()`).

Les classes génériques peuvent être « surchargées » par des instanciations définies, soit donc des classes pour lesquelles le type des paramètres génériques est donné. On utilise pour cela la syntaxe :

```
class NomClasse<type_paramètre_générique1, type_paramètre_générique2, ...>
```

Cela permet ainsi de préciser le comportement de la classe pour un cas particulier de type de variables ou d'objets.

Ex. :

```
class ListeSimple<string>
{
   ...
};

ListeSimple<string>::ListeSimple(int t)
{
   ...
}
...
```

Les entités génériques, lors de leur instanciation, sont parfois analysées par le préprocesseur, ce qui pose des problèmes lors de la compilation.

Dans le cas d'une classe générique dont la déclaration et la définition sont séparées suivant le couple de fichiers *.h*/*.cpp*, il faut positionner l'intégralité du code de l'implémentation des méthodes directement dans le fichier *.h* (en revanche, inutile d'en faire des méthodes inline et de mettre le code dans la déclaration de la classe).

Les fonctions avec lesquelles est appliquée la classe générique nécessitent d'être déclarées dans le *.h*, même si l'implémentation doit en revanche rester dans le fichier *.cpp* (Ex. : la fonction de surcharge de l'opérateur <<).

7 LES EXCEPTIONS

7.1 INTRODUCTION

Dans le cadre de la fiabilisation d'un programme, il est impératif de prévoir et de gérer les erreurs d'exécution et situations anormales. Celles-ci pouvant être de natures très diverses, matérielles (Ex. : erreur d'accès à un fichier) ou logicielles (Ex. : division par zéro), et un programme étant généralement constitué de différentes structures imbriquées (une procédure [1] appelant une autre procédure, qui elle-même appelle une tierce procédure, etc.), cela soulève plusieurs interrogations :

- L'erreur remet-elle en cause la suite de l'exécution du programme (erreur fatale / irrécupérable), ou peut-elle être traitée de manière à poursuivre l'exécution ?
- S'il y a traitement, est-il plus approprié de l'effectuer à bas niveau (procédure qui a détecté l'erreur), ou bien à haut niveau (procédure appelante ou de niveau supérieur) ?
  - bas niveau : l'erreur ou situation anormale est détectable, mais il est difficile de prédir le comportement à adopter, lequel par ailleurs peut ne pas être toujours le même, être fonction de paramètres et conditions gérés par une autre procédure de l'application, ou bien nécessiter l'intervention de l'utilisateur ;
  - haut niveau : le traitement de l'erreur est réalisable, mais il est difficile voire impossible de la détecter.

Ex. : Erreur d'exécution lors de l'ouverture d'un fichier (causes possibles : fichier inexistant, système de fichiers illisible, demande d'ouverture en écriture d'un fichier en lecture seule, …).

```
void ouvrirFichier(char* nomFichier)
{
   // ouvrirFichier() sait détecter une erreur d'ouverture du fichier
   // mais quel traitement réaliser : demander à ouvrir un autre fichier,
   // créer un fichier du nom spécifié, terminer le programme, ... ?
}

int main(void)
{
   char* nomFichier;
   ...
   ouvrirFichier(nomFichier);
   // main() sait comment réagir en cas de problème d'ouverture du fichier
   // mais comment détecter un tel problème ?

   return 0;
}
```

Cette problématique se traduit, pour chaque erreur d'exécution ou situation anormale potentielle, par un traitement selon l'une des 3 stratégies suivantes :

- L'erreur provoque l'arrêt complet du programme ;
- L'erreur est traitée localement, et généralement de manière transparente, par la procédure qui l'a détectée ;
- L'erreur est signalée à la procédure appelante par la procédure qui l'a détectée via un mécanisme permettant de spécifier le type d'erreur (Ex. : retour d'un code d'erreur).
  En ce cas, l'erreur est soit traitée par la procédure appelante, soit est à nouveau signalée à la procédure de niveau supérieur – laquelle a donc en charge le traitement de l'erreur –, soit signalée à la procédure de niveau encore supérieur, etc. ; ce qui au plus haut niveau se caractérise par une interaction avec l'utilisateur.

Le traitement d'une erreur peut être réalisé par une structure de contrôle (`if () else`), mais dans le cas d'une erreur qui n'est pas traitée localement mais doit être signalée à la procédure appelante, cela n'est pas toujours

[1] Fonction ou méthode.

réalisable. Il est effectivement parfois délicat de réaliser à haut niveau l'importance et les conséquences des erreurs d'exécution de bas niveau, et le traitement associé, s'il y en a un, peut ne pas être approprié.

Il est donc nécessaire de disposer d'un mécanisme de communication entre les différents niveaux d'une application.

7.2 DÉFINITIONS

Le terme **exception** désigne le signal envoyé à une procédure par une autre procédure ayant détecté l'occurrence d'une erreur d'exécution ou d'une situation anormale.

À cette exception est associé un mécanisme qui permet de s'assurer que l'erreur détectée puisse être prise en compte par la procédure appelante ou l'une des procédures de niveau supérieur. Ce mécanisme est en réalité une nouvelle structure de contrôle, introduisant de nouveaux mots-clefs :

- Une procédure qui détecte une erreur ou une situation anormale construit une exception et la propage/signale/« lance » (mot-clef `throw`) à l'intention de la procédure appelante ;
- Lorsqu'une procédure désire être capable de prendre en charge l'exception qu'une procédure utilisée directement ou indirectement est susceptible de lancer, elle « essaye » d'exécuter (mot-clef `try`) la procédure en question ;
- L'exception potentielle est interceptée/« attrapée » (mot-clef `catch`) par la procédure appelante via un gestionnaire d'exception, lequel réalise alors le traitement approprié.

```
void procedureBasNiveau()
{
   if ( erreur ) // condition(s) d'erreur
      throw exception; // lancement d'une exception
}

void procedureHautNiveau()
{
   try {
      // code susceptible de lancer une exception
      procedureBasNiveau();

      // code jamais exécuté si la procédure de bas niveau lance une exception
      ...
   }
   catch (...) { // gestion de l'exception (les ... sont littéral)
      // traitement à réaliser en cas de lancement d'une exception
   }

   // poursuite de l'exécution
   ...
}
```

Un bloc de code `try` doit être immédiatement suivi d'un bloc de code `catch ()`.

Lorsqu'une exception est lancée, la suite d'instructions du bloc `try` est interrompue, et l'exécution passe directement au gestionnaire d'exception décrit dans le bloc `catch ()` qui définit le traitement de l'exception, court-circuitant ainsi les instructions restantes du bloc `try` qui ne seront jamais exécutées.

Si des variables ont été déclarées dans le bloc `try`, leur mémoire allouée est automatiquement libérée avant l'exécution du bloc `catch ()`. Si des objets ont été déclarés dans le bloc `try`, leur destructeur est appelé et leur mémoire allouée est automatiquement libérée avant l'exécution du bloc `catch ()`.

Une fois le bloc `catch ()` exécuté, l'exécution reprend son cours normal aux lignes d'instructions qui suivent.

7.3 IMPLÉMENTATION

7.3.1 Interception et gestion des exceptions

L'**interception** et la **gestion** d'une exception sont réalisées grâce à la structure de contrôle `try catch ()`, selon la syntaxe :

```
try {
   // code susceptible de lancer une exception
}
catch (...) { // gestion de tout type d'exception
   // traitement à réaliser en cas de lancement d'une exception
}
```

Ex. : Une procédure de haut niveau appelle une procédure de bas niveau qui nécessite un paramètre entier de valeur strictement positive pour s'exécuter correctement ; si ce n'est pas le cas, elle lance une exception.

```
void procedureBasNiveau(int x)
{
   // x doit être strictement positif sinon une exception est lancée
}

void procedureHautNiveau(int xx)
{
   cout << "avant l'exception" << endl;

   try {
      // code susceptible de lancer une exception
      cout << "avant la procedure de bas niveau" << endl;
      procedureBasNiveau(xx);
      cout << "apres la procedure de bas niveau" << endl;
   }
   catch (...) { // prise en compte de tout type d'exception
      // traitement à réaliser en cas de lancement d'exception
      cout << "!probleme!" << endl;
   }

   // poursuite de l'exécution
   cout << "apres l'exception" << endl;
}


int main(void)
{
   int nb;
   cin >> nb;   // saisie de l'utilisateur devant être strictement positive
   procedureHautNiveau(nb);

   return 0;
}
```

Si le nombre saisi par l'utilisateur est supérieur à 0, alors l'exécution affiche : « avant l'exception » / « avant la procédure de bas niveau » / « après la procédure de bas niveau » / « après l'exception ».
Si le nombre saisi par l'utilisateur est inférieur ou égal à 0, alors l'exécution affiche : « avant l'exception » / « avant la procédure de bas niveau » / « !probleme! » / « après l'exception ».

L'exception construite (via `throw`) est disponible sous la forme de nombre, de chaîne de caractères, d'objet, d'objet dédié pour les exceptions, etc. et est généralement porteuse d'informations sur l'erreur qui a été détectée. La procédure qui attrape l'exception peut récupérer ces informations et les utiliser pour adapter le traitement à l'erreur détectée.
De plus, la capacité d'une exception à être disponible sous différentes formes permet à une procédure de bas niveau de générer plusieurs exceptions distinctes, différenciées par leur type.

Lors de la gestion de l'exception, les différents types d'exceptions pouvant être lancées doivent être filtrés en construisant différents gestionnaires d'exceptions spécialisés. Pour spécialiser un bloc `catch ()`, il suffit de préciser un paramètre, selon la même syntaxe que les paramètres d'entrée d'une procédure.
Les différentes spécialisations d'exceptions sont mentionnées en spécifiant successivement autant de blocs `catch ()` que de types d'exceptions différents à gérer. Le bloc `catch (...)` intercepte tout type d'exception.

```
try {
   // code susceptible de lancer une exception
}
catch (type_exception varExcept) { // gestion des exceptions du type type_exception
   // traitement à réaliser en cas de lancement d'exception de type type_exception
   // le paramètre varExcept peut être utilisé pour récupérer plus d'informations
}
```

Ex. : La procédure de bas niveau peut lancer plusieurs types d'exceptions, afin de renseigner sur l'erreur détectée : exception de type `int` lorsque la valeur du paramètre est strictement négative, exception de type `char*` lorsque la valeur du paramètre est nulle.

```
...

void procedureHautNiveau(int xx)
{
   try {
      procedureBasNiveau(xx);
   }
   catch (int code_erreur) { // exception de type int
      cout << "!error " << code_erreur << "!" << endl;
   }
   catch (char* message) { // exception de type char*
      cout << message << endl;
   }
   catch (...) { // exceptions d'autres types
      cout << "!probleme inconnu!" << endl;
   }
}

...
```

Lors du lancement d'une exception, les différents gestionnaires `catch ()` sont examinés l'un après l'autre, dans l'ordre dans lequel ils sont déclarés, jusqu'à en trouver un capable de gérer le type d'exception qui a été lancée ; c'est-à-dire un gestionnaire dont le paramètre est compatible avec le type de l'exception [1].

Si c'est le cas, alors l'exception est considérée comme ayant été attrapée, et aucun autre gestionnaire `catch ()` ne peut être exécuté, même si le type d'exception correspond.

Une fois le bloc `catch ()` pleinement traité, l'exécution se poursuit à l'instruction qui suit le dernier gestionnaire d'exceptions.

Une procédure peut aussi décider de ne traiter que certains types d'exceptions en particulier ; les autres sont traités par une ou plusieurs tierces procédures à d'autres niveaux (soit plus haut, soit plus bas entre la procédure courante et la procédure qui a lancé l'exception). Pour cela, il suffit de ne mentionner que les gestionnaires d'exceptions désirés.

Ex. : La procédure de haut niveau ne prend en charge qu'une partie des exceptions pouvant être lancées par la procédure de bas niveau – les exceptions de type `int` ; les autres types d'exceptions sont attrapés à plus haut niveau.

```
...

void procedureHautNiveau(int xx)
{
   try {
      procedureBasNiveau(xx);
   }
   catch (int code_erreur) { // exception de type int
      cout << "!error " << code_erreur << "!" << endl;
   }
   // aucun autre type d'exception attrapé ici
}


int main(void)
{
   int nb;
   cin >> nb;
   try {
      procedureHautNiveau(nb);
   }
   catch (char* message) { // exception de type char*
      cout << "-" << message << "-" << endl;
   }

   return 0;
}
```

[1] Cette compatibilité peut être réalisée par des conversions implicites et les propriétés de l'héritage et du polymorphisme.

Dans le cas où une exception lancée n'est attrapée par aucune autre procédure (aucune prise en compte d'exception via `try catch ()` définie, ou aucun gestionnaire `catch ()` compatible), une erreur d'exécution [1] est générée [2] via la fonction prédéfinie `terminate()` (espace de noms `std`).

Il est possible de modifier le comportement par défaut de l'application vis-à-vis de toute exception lancée mais non attrapée, en utilisant la fonction `set_terminate()` (espace de noms `std`), selon la syntaxe `set_terminate(fonction)`.

Ex. : Modification du traitement par défaut pour les exceptions lancées mais non attrapées.

```
void termProc()
{
   cout << "!exception non attrapee!" << endl;
   exit (-1);   // on n'oublie pas de quitter le programme en signalant une erreur
}

void procedureBasNiveau(int x)
{
   ...
}

void procedureHautNiveau(int xx)
{
   set_terminate(termProc); // redéfinition du comportement en cas d'exception
                            // non attrapée
   try {
      procedureBasNiveau(xx);
   }
   catch (int code_erreur) { // seul ce type d'exception est attrapé
      cout << "!error " << code_erreur << "!" << endl;
   }
   // en cas de valeur nulle pour xx, une exception de type char* est lancée
   // comme celle-ci n'est pas attrapée, une erreur d'exécution est générée
   // et termProc() est alors exécuté

   procedureBasNiveau(-1); // aucun type d'exception attrapé
   // comme aucune exception n'est attrapée, il y a erreur d'exécution
   // dans tous les cas où xx n'est pas strictement positif
}


int main(void)
{
   int nb;
   cin >> nb;
   procedureHautNiveau(nb);

   return 0;
}
```

7.3.2 Lancement des exceptions

Le **lancement** d'une exception est réalisée grâce à l'instruction `throw`, selon la syntaxe `throw exception`. Généralement le lancement d'une exception est subordonné au résultat d'un test dont le but est de détecter l'erreur ou la situation anormale.

```
if ( erreur ) // condition(s) d'erreur
   throw exception; // lancement de l'exception
```

Ex. : Une procédure de haut niveau appelle une procédure de bas niveau qui nécessite un paramètre de valeur positive pour s'exécuter correctement ; si ce n'est pas le cas, elle lance une exception : exception de type `int` pour une valeur du paramètre strictement négative, exception de type `char*` pour une valeur du paramètre nulle.

[1] Erreur d'exécution (fr) ≡ Runtime Error (eng).
[2] L'utilisation de la procédure susceptible de signaler une exception ne génère une erreur d'exécution que si une exception est lancée, sinon l'exécution suit un cours parfaitement normal.

```
void procedureBasNiveau(int x)
{
   if ( x < 0 )
      throw -1; // lancement d'une exception de type int
   if ( x == 0 )
      throw (char*)"!valeur nulle!"; // lancement d'une exception de type char*
}

void procedureHautNiveau(int xx)
{
   ...
}

...
```

Comme il est délicat voire impossible de savoir quels types d'exceptions est susceptible de générer une procédure [1], on le précise généralement dans son en-tête, à la suite du prototype en utilisant le mot-clef `throw` suivant la syntaxe `throw (type_exception1, type_exception2, …)` [2].

Ex. : La procédure de bas niveau peut lancer uniquement des exceptions de type `int` et `char*`, afin de renseigner sur l'erreur détectée.

```
void procedureBasNiveau(int x) throw (int, char*)
{
   ...
}

...
```

Lorsque l'en-tête d'une procédure ne mentionne aucun type d'exception, c'est qu'elle peut en fait lancer tout type d'exception.

Pour spécifier qu'aucune exception ne peut être générée, on précise simplement `throw ()` (sans paramètre) dans l'en-tête.

Lorsqu'une procédure lance une exception non déclarée dans l'en-tête, cela revient en fait au cas d'une erreur d'exécution non prévue et donc non gérée. Une fonction spécifique est alors exécutée : il s'agit de la fonction `unexpected()` (espace de noms `std`), dont le comportement par défaut est d'exécuter un traitement qui en définitif appelle la fonction `terminate()`.

Il est possible de modifier ce comportement par défaut vis-à-vis de toute exception lancée mais non déclarée en utilisant la fonction `set_unexpected()` (espace de noms `std`), selon la syntaxe `set_unexpected(fonction)`.

Ex. : Modification du traitement par défaut pour les exceptions lancées mais non déclarées.

```
void termProc()
{
   cout << "!exception non attrapee!" << endl;
   exit (-1); // sortie avec retour d'un code d'erreur
}

void unexProc()
{
   cout << "!exception non declaree!" << endl;
   exit (-1); // sortie avec retour d'un code d'erreur
}

void procedureBasNiveau(int x) throw (int) // seul un type d'exception est déclaré
{
   ...
}
```

[1] L'utilisateur d'une procédure n'est pas obligatoirement – voire même peu souvent – le développeur de la procédure.

[2] Cependant, la déclaration des exceptions qu'une procédure laisse échapper ne fait partie ni de son prototype, ni de sa signature.

```
void procedureHautNiveau(int xx)
{
   set_terminate(termProc);  // redéfinition du comportement en cas d'exception
                             // non attrapée
   set_unexpected(unexProc); // modification du comportement en cas d'exception
                             // non déclarée
   try {
      procedureBasNiveau(xx);
   }
   catch (int code) {
      cout << "!" << code << "!" << endl;
   }
   catch (char* message) {
      cout << message << endl;
   }
   catch (...) { // exceptions d'autres types
      cout << "!probleme inconnu!" << endl;
   }
   // tous les types d'exceptions sont attrapés
   // en cas de valeur nulle pour xx, une exception de type char* est lancée
   // mais comme celle-ci n'est pas déclarée, une erreur d'exécution est générée
   // et unexProc() est alors exécuté

   procedureBasNiveau(-1); // aucun type d'exception attrapée
   // comme aucune exception n'est attrapée, une erreur d'exécution est générée
   // dans tous les cas où xx n'est pas strictement positif
   // et termProc() est alors exécuté
}

int main(void)
{
   int xx;
   cin >> xx;
   procedureHautNiveau(xx);

   return 0;
}
```

A MOTS-CLEFS DU LANGAGE C++

A.1 ORDRE ALPHABÉTIQUE

A	`auto`					
B	`bool`	`break`				
C	`case`	`catch`	`char`	`class`	`const`	`continue`
D	`default`	`delete`	`do`	`double`		
E	`else`	`enum`	`explicit`	`extern`		
F	`false`	`float`	`for`	`friend`		
G	`goto`					
I	`if`	`inline`	`int`			
L	`long`					
N	`new`	`NULL`				
O	`operator`					
P	`private`	`protected`	`public`			
R	`register`	`return`				
S	`short`	`signed`	`sizeof`	`static`	`struct`	`switch`
T	`template`	`this`	`throw`	`true`	`try`	`typedef`
U	`union`	`unsigned`				
V	`virtual`	`void`	`volatile`			
W	`while`					

Les mots-clefs sont réservés, donc inutilisables comme nom de variable, fonction, classe, attribut ou méthode [1].

A.2 CATÉGORIES

A.2.1 Classes et membres

`class`	Déclaration d'une classe.
`delete`	Destruction d'un objet (libération de l'espace mémoire).
`friend`	Pour une classe, définition d'une classe, méthode ou fonction amie (contournement de l'encapsulation pour accéder aux membres de la classe).
`new`	Instanciation d'un objet (réservation de l'espace mémoire).
`operator`	Préfixe générique aux surcharges d'opérateurs.
`template`	Définition de paramètres génériques.
`this`	À l'intérieur d'une méthode d'une classe, référence à l'objet de la classe auquel est appliqué la méthode.

A.2.2 Modificateurs de visibilité de membres

`private`	Visibilité limitée à la propre classe du membre.
`protected`	Visibilité limitée à la propre classe du membre et à ses sous-classes.
`public`	Aucune restriction de visibilité.

[1] Même si le langage C++ distingue la casse, il est déconseillé d'utiliser un nom, même avec une casse distincte, proche d'un mot-clef existant.

A.2.3 Types primitifs de variables et attributs

`bool`	Booléen (valeur `true`/`false`).
`char`	Caractère sur 1 octet (standard ASCII étendu).
`double`	Réel sur 8 octets.
`float`	Réel sur 4 octets.
`int`	Entier sur 4 octets.
`short`	Entier sur 2 octets.
`void`	Type « vide » (utilisé uniquement pour désigner un retour de fonction/méthode d'un type inutilisé/inconnu).

A.2.4 Modificateurs d'attributs

`static`	Attribut de classe (et non attribut d'instance) ; disponible même sans disposer d'instance de la classe, commun pour toutes les instances, valeur partagée par toutes les instances.

A.2.5 Modificateurs de méthodes

`const`	Méthode constante, ne modifiant pas l'objet auquel est appliqué la méthode.
`explicit`	Constructeur étant obligatoirement appelé explicitement (`Classe objet = Classe();`), empêchant ainsi notamment le compilateur de réaliser des conversions implicites potentiellement dangereuses (`Classe objet(valeur);` / `Classe objet = valeur;`).
`static`	Méthode de classe (et non méthode d'instance) ; disponible même sans disposer d'instance de la classe, commune et partagée par toutes les instances.
`virtual`	Méthode virtuelle ; pour les sous-classes, priorité au type dynamique (instanciation) plutôt qu'au type statique (déclaration).
`= 0`	Méthode virtuelle pure ; aucune implémentation donnée dans la classe elle-même, qui devient alors une classe abstraite ; implémentation donnée dans les sous-classes.

A.2.6 Modificateurs de variables et d'attributs

V : variable de type primitif, O : objet, A : attribut de type primitif, Ao : attribut-objet.

	V	O	A	Ao	
`auto`	X	X			Variable locale, qui n'existe que durant l'exécution du bloc de code de déclaration – *par défaut*.
`const`	X	X	X	X	Variable de valeur constante initialisée une seule fois et qui ne peut pas être modifiée lors de l'exécution.
`extern`	X	X			Déclaration partagée par plusieurs fichiers source ; sa durée de vie est celle d'une déclaration globale (cf. `static`).
`long`	X		X		Variable dont la taille est doublée.
`register`	X	X			Variable stockée dans l'un des registres du processeur, et dont les accès sont ainsi optimisés (Nb : amélioration des performances non garantie).
`signed`	X		X		Variable en format numérique signé, utilisant le bit de poids fort en guise de signe (0 : positif, 1 : négatif) ; échelle de valeurs centrée autour de 0 (types numériques uniquement, cf. `unsigned`) – *par défaut*.
`static`	X	X	X	X	Variable locale dont la valeur est conservée entre deux appels de la fonction (cf. `auto`).
`unsigned`	X		X		Variable en format numérique non signé, utilisant tous les bits pour la valeur ; début de l'échelle de valeurs à partir de 0 (types numériques uniquement, cf. `signed`).
`volatile`	X	X	X	X	Variable sauvegardée uniquement en mémoire vive, et qui n'est jamais bufférisée ; sa valeur instantanée est ainsi assurée lors de l'accès par diverses fonctions, lorsque celles-ci sont susceptibles d'en modifier la valeur et que le programme ne peut le prévoir à la compilation.

A.2.7 Modificateurs de fonctions et de méthodes

`inline`	Définition d'une fonction-macro – appel de la fonction remplacé par son code source avant la compilation par le préprocesseur.

A.2.8 Types de variables et attributs complexes

`enum`	Définition d'un type énuméré – liste de mnémoniques alphanumériques de valeur entière et successive (0, 1, 2, …).
`struct`	Définition d'un type structuré – regroupement de données de types différents (appelés *champs*) dans un seul ensemble.
`typedef`	Création d'un nouveau type sur la base d'une variable utilisateur (variable simple, tableau, pointeur, structure, etc.).
`union`	Définition d'un type structuré commun – mise en commun d'un espace mémoire accessible par des données de types différents (appelés *champs*).

A.2.9 Structures de contrôle

`break`	Sortie de boucle ou de bloc de code.
`case`	Test multiple – sélection (cf. `default`, `switch`).
`continue`	Poursuite de l'exécution de la boucle (branchement à l'itération suivante (test de continuité)).
`default`	Test multiple – sélection par défaut (cf. `case`, `switch`).
`do`	Boucle avec une itération minimale (cf. `while`).
`else`	Test simple – test conditionnel inverse (cf. `if`).
`for`	Boucle avec itération.
`if`	Test simple – test conditionnel (cf. `else`).
`return`	Sortie du bloc de code avec ou sans valeur de retour.
`switch`	Test multiple – test de sélection (cf. `case`, `default`).
`while`	Boucle avec une itération minimale (cf. `do`) ou aucune.

A.2.10 Gestion des exceptions

`catch`	Capture et traitement d'une exception (cf. `try`).
`throw`	Lancement explicite d'une exception, ou déclaration des types d'exceptions qu'une fonction/méthode est susceptible de lancer.
`try`	Déclaration d'un bloc de code susceptible de lancer une exception (cf. `catch`).

A.2.11 Autres

`false`	Valeur booléenne « fausse ».
`goto`	Branchement à une étiquette (désuet et déconseillé).
`NULL`	Pointeur nul (pas d'espace mémoire réservé).
`sizeof`	Renvoi de la taille en octets occupée par un type donné, ou par une variable (donc par son type).
`true`	Valeur booléenne « vraie ».

B DÉLIMITEURS ET OPÉRATEURS

B.1 DÉLIMITEURS

`;`	Marque la fin d'une ligne d'instruction ou d'une déclaration.
`,`	Sépare deux éléments d'une liste.
`" "`	Délimite une chaîne de caractères.
`' '`	Encadre un caractère.
`( )`	Encadre une liste de paramètres.
`{ }`	Délimite un bloc d'instructions, ou une liste de valeurs d'initialisation.
`[ ]`	Encadre la taille ou l'indice d'un tableau.
`< >`	Encadre la définition de paramètres génériques avant la définition d'un modèle, ou la désignation des types à utiliser pour les paramètres génériques lors d'une utilisation du modèle.

B.2 OPÉRATEURS

On distingue trois types d'opérateurs :

- unaire : 1 argument ($_1$), syntaxe `opérateur opérande` ;
- binaire : 2 arguments ($_2$), syntaxe `opérande1 opérateur opérande2` ;
- ternaire : 3 arguments ($_3$), syntaxe `opérande1 opérateurA opérande2 opérateurB opérande3`.

B.2.1 Opérateur d'affectation

`=`	Affectation. ($_2$)
`(op)=`	Opérateurs combinés – opération puis affectation (cf. B.2.4). ($_2$)

B.2.2 Opérateurs arithmétiques

`+`	Addition. ($_2$)
`-`	Soustraction. ($_2$)
`*`	Multiplication. ($_2$)
`/`	Division. ($_2$)
`%`	Modulo – reste de la division entière (division euclidienne) entre deux entiers. ($_2$)
`-`	Opposé (positif/négatif). ($_1$)

B.2.3 Opérateurs binaires

`&`	ET logique (AND) – vrai si les deux sont vrais / faux si l'un des deux est faux. ($_2$)
`|`	OU logique (OR) – vrai si au moins l'un des deux est vrai / faux si les deux sont faux. ($_2$)
`!`	NON logique (NOT) – vrai si faux / faux si vrai. ($_1$)
`^`	OU logique exclusif (XOR) – vrai si l'un ou l'autre est vrai mais pas les deux / faux si les deux sont vrais ou si les deux sont faux. ($_2$)
`~`	Complément à un (inversion de tous les bits). ($_1$)
`<<`	Décalage à gauche du nombre de bits spécifiés – 1 bit : multiplication par 2, 2 bits : multiplication par 4, etc. ($_2$)
`>>`	Décalage à droite du nombre de bits spécifiés – 1 bit : division par 2, 2 bits : division par 4, etc.. ($_2$)

B.2.4 Opérateurs combinés

`++`	Incrémentation (pré- et post-). (1)
`--`	Décrémentation (pré- et post-). (1)
`+=`	Addition et affectation. (2)
`-=`	Soustraction et affectation. (2)
`*=`	Multiplication et affectation. (2)
`/=`	Division et affectation. (2)
`%=`	Modulo et affectation. (2)
`<<=`	Décalage à gauche et affectation. (2)
`>>=`	Décalage à droite et affectation. (2)
`&=`	ET logique bit-à-bit et affectation. (2)
`|=`	OU logique bit-à-bit et affectation. (2)
`^=`	OU logique exclusif bit-à-bit et affectation. (2)

B.2.5 Opérateurs d'évaluation d'expression

`==`	Égalité. (2)
`!=`	Différence. (2)
`<`	Infériorité stricte. (2)
`<=`	Infériorité ou égalité. (2)
`>`	Supériorité stricte. (2)
`>=`	Supériorité ou égalité. (2)
`&&`	ET logique (AND). (2)
`||`	OU logique (OR). (2)
`!`	NON logique (NOT). (1)

B.2.6 Opérateurs divers

`type()`	Conversion explicite de type (transtypage, cast). (1)
`(type)`	Conversion de type (transtypage, cast). (1)
`*`	Opérateur d'indirection (/déférencement) – accès au contenu. (1)
`&`	Opérateur d'adresse – accès au contenant. (1)
`delete`	Destruction de variable/instance créée dynamiquement. (1)
`delete[]`	Destruction de tableau de variables/instances créé dynamiquement. (1)
`new`	Création dynamique de variable/instance. (1)
`new[ ]`	Création dynamique de tableau de variables/instances. (1)
`::`	Opérateur de résolution de portée. (2)
`.`	Accès à un champ d'une classe, d'une structure ou d'une union. (2)
`->`	Accès à un champ d'une classe, d'une structure ou d'une union pointée. (2)
`? :`	Opérateur ternaire (exécution conditionnelle simplifiée). (3)

B.3 PRIORITÉ DES OPÉRATEURS ET DÉLIMITEURS

Le niveau de priorité des opérateurs et délimiteurs est donné du plus prioritaire au moins prioritaire. Pour forcer les priorités, il faut donc utiliser des parenthèses.

← + prioritaire

`::` `[ ]` `( )` `type( )` `.` `->` `var++` `var--` `new` `new[ ]`

`delete` `delete[]` `++var` `--var` `*` (1) `&` (1) `+` (1) `-` (1) `!` `~` `sizeof` `typeid`

`(type)` `const_cast` `dynamic_cast` `reinterpret_cast` `static_cast` `.*` `->*` `*` (2)

`/` `%` `+` (2) `-` (2) `<<` `>>` `<` `>` `<=` `>=` `==` `!=` `&` (2) `^` `|` `&&` `||`

`? :` (3) `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `|=` `^=` `,` – prioritaire →

(1) : opérateur unaire, (2) : opérateur binaire, (3) : opérateur ternaire.

C LA STL : LIBRAIRIE STANDARD

C.1 INTRODUCTION

Le langage C++, à l'instar du langage C, reste limité en « opérations de base ». La **STL** (Standard Template Library [1]) – ou *librairie standard* –, intégrée à la norme ISO (98), regroupe un ensemble de bibliothèques et enrichit ainsi considérablement le langage.

Au-delà du gain certain en efficacité et en simplicité pour tout programmeur C++, l'utilisation de la STL favorise la portabilité ainsi que la maintenabilité.

La STL offre ainsi :

- une partie des fonctions système des bibliothèques standard du langage C (standard ANSI) ;
- des classes `string` (ASCII ou Unicode) ;
- des classes pour la gestion des structures de données (conteneurs, itérateurs et algorithmes) ;
- un ensemble de classes pour la gestion des flux d'entrées/sorties ;
- …

Toutes les inclusions de fichiers d'en-tête de la STL sont réalisées par convention sans préciser l'extension *.h*. Cela permet de ne pas avoir à se préoccuper de la manière dont est implémentée la STL employée.

C.2 PRÉSENTATION

C.2.1 Les espaces de noms

Un **espace de noms** est un regroupement en un seul ensemble d'entités diverses (classes, fonctions, types, variables et constantes) ayant trait généralement à une même thématique [2].

La déclaration d'un espace de noms se réalise à l'aide du mot-clef `namespace` en implémentant les entités de l'espace dans un bloc de code associé suivant la syntaxe :

```
namespace NomEspaceNoms {
   /* déclaration, définition, implémentation, ... des entités de l'espace de noms */
}
```

L'utilisation d'un membre d'un espace de noms implique que le nom de celui-ci soit mentionné :

- par préfixation du membre avec le nom de son espace de noms suivi de l'opérateur de résolution de portée (nom pleinement qualifié) ;
  Ex. : `NomEspaceNoms::membre`
- par déclaration dans l'en-tête de l'intégralité de l'espace de noms [3] ;
  Ex. : `using namespace NomEspaceNoms;`
- par déclaration dans le code du membre de l'espace de noms.
  Ex. : `using NomEspaceNoms::membre;`

L'utilisation des unités d'organisation que sont les espaces de noms vise à classifier des éléments de code source, et permet ainsi de réduire les sources de conflits. Cependant, des conflits peuvent subsister lors d'une utilisation en déclaration dans l'en-tête.

[1] Standard Template Library (eng) ≡ Librairie de modèles standard (fr).
[2] Un espace de noms se rapproche ainsi de la notion de paquetage, présente dans les langages ADA et Java par exemple.
[3] Un peu à la manière d'une inclusion de bibliothèque.

Il est à noter qu'un espace de noms :

- peut être imbriqué dans un autre (`NomEspaceNoms1::NomEspaceNoms2::membre`) ;
- peut être défini dans plusieurs fichiers sources distincts ;
- peut être défini par un alias (`namespace MonEspaceNom NomEspaceNoms1::NomEspaceNoms2`) ;
- ne propose – malheureusement – aucun caractère de visibilité spécifique à ses membres.

Un espace de noms par défaut existe ; il s'agit de l'espace de noms global, et n'a pas de nom. Pour accéder à ses membres, on écrit donc `::membre`.
Tout espace de noms défini est donc « imbriqué » implicitement dans l'espace de noms par défaut.

L'espace de noms standard `std` contient l'intégralité des entités de la STL ; la déclaration `using namespace std` ou la syntaxe `std::membre` est donc très courante.

C.2.2 La gestion des chaînes de caractères

La STL propose deux classes facilitant grandement la manipulation des chaînes de caractères (bibliothèque *string*) :

- `string` : pour les caractères codés sur 8 bits (ASCII étendu) ;
- `wstring` : pour les caractères codés sur 16 bits (Unicode).

Les classes type *string* ne sont ni plus ni moins que des classes stockant une chaîne de caractères et proposant diverses méthodes afin de rendre la gestion de la chaîne la plus pratique et ergonomique qui soit, tout en masquant les problèmes inhérents à ce type (gestion de la taille automatique, optimisation de l'espace mémoire, etc.).

Voici une liste non-exhaustive des méthodes de la classe `string` :

`string()`	Constructeur par défaut (généralement, crée une chaîne de 20 caractères).
`string(const string&)`	Constructeur de recopie.
`string(char*)`	Construction d'un objet `string` à partir d'une chaîne de caractères.
`string(char*, int)` `string(const string&, int)`	Construction d'un objet `string` à partir des `n` premiers caractères d'une chaîne ou d'un objet `string` déjà créé ; si le nombre de caractères spécifié est supérieur à la taille de la chaîne source, la taille de l'objet `string` est calquée sur celle-ci.
`string& operator=(...)` `string& assign(...)`	Modification d'un objet `string`.
`char* c_str()` `char* str()`	Renvoi de la chaîne stockée ; la chaîne se termine par le caractère fin de chaîne.
`char* data()`	Renvoi du tableau de caractères stocké ; le tableau ne se termine pas par le caractère fin de chaîne.
`int size()` `int length()`	Renvoi de la taille de la chaîne stockée dans l'objet `string`.
`char operator[](int)`	Renvoi du énième caractère de la chaîne stockée.
`char at(int)`	Renvoi du énième caractère de la chaîne stockée avec contrôle de la validité de l'indice.
`void clear()`	Effacement de la chaîne stockée.
`bool empty()`	Renvoi de `TRUE` si la chaîne est vide, `FALSE` sinon.
`string& operator+=(...)` `string& append(...)`	Concaténation de la chaîne stockée.
`string& insert(...)`	Insertion dans la chaîne stockée.
`string& erase(...)`	Effacement de caractères dans la chaîne stockée.
`string& replace(...)`	Remplacement de caractères dans la chaîne stockée.
`int find(...)` `int rfind(...)`	Recherche de caractères dans la chaîne stockée.
`int compare(...)`	Comparaison avec la chaîne stockée.
`string substr(int, int)`	Extraction d'une sous-chaîne.

C.2.3 Les structures de données

La gestion des structures de données est assurée par le trio indissociable des concepts fondamentaux suivants :

- conteneur : stockage des données dans la structure ;
- itérateur : accès et parcours de la structure ;
- algorithme : fonction applicable à la structure.

C.2.3.1 Les conteneurs

Un **conteneur** est un objet qui représente une structure de données dynamique. En d'autres termes, un conteneur [1] est un objet qui permet de stocker un ensemble d'objets du même type.

Pour pouvoir être utilisé avec n'importe quel type d'objets, les conteneurs sont implémentés en tant que classes génériques [2], le paramètre correspondant au type d'objets que le conteneur doit stocker.

Il existe plusieurs types de conteneurs :

- conteneurs élémentaires : structures de données simples avec accès selon un index ;
  - vecteur (`vector`) : tableau dynamique, stockage ordonné et accès indexé à tout élément de la structure ;
  - liste (`list`) : liste chaînée, stockage non-ordonné et accès au premier ou au dernier élément de la structure ;
  - DQ (`deque`) : liste à accès préférentiel premier dernier, stockage non-ordonné et accès au premier, au dernier élément ou au milieu de la liste.
- conteneurs élaborés : structures de données basées sur un conteneur élémentaire ;
  - pile (`stack`) : structure de données type LIFO (Last In First Out), le premier traité est le dernier arrivé ;
  - file (`queue`) : structure de données type FIFO (First In First Out), le premier traité est le premier arrivé ;
  - tas (`priority_queue`) : file à priorité, le premier traité est le plus prioritaire.
- conteneurs associatifs : structures de données type (clef, élément) avec accès selon la clef, celle-ci pouvant être un objet.
  - ensemble (`set`) : collection ordonnée pour laquelle la clef est l'élément associé lui-même ;
  - table (`map`) : collection ordonnée pour laquelle la clef et l'élément associé sont distincts.

  Nb : On notera qu'il existe des versions « multiples » (`multiset`, `multimap`) des conteneurs associatifs pour lesquels une même clef permet d'accéder à plusieurs éléments.

C.2.3.2 Les itérateurs

Un **itérateur** est un objet qui permet d'accéder en lecture ou écriture à l'un des éléments de la structure de données, ou bien de se déplacer au sein de cette structure ; un itérateur peut ainsi être comparé à un pointeur qui permet de parcourir un tableau et d'accéder à son contenu.

Un itérateur peut ainsi proposer les opérations suivantes :

- déréférencement : accès en lecture ou écriture de la valeur de l'élément courant ;
- déplacement : modification de l'élément courant (pas de sa valeur) en parcourant la structure en avant, en arrière, ou de manière aléatoire.

C.2.3.3 Les algorithmes

Un **algorithme** est une fonction applicable à un conteneur afin de modifier son contenu ou d'y accéder, obtenir des informations, gérer le conteneur, etc. L'algorithme fait usage des itérateurs pour réaliser ses opérations.

Le type d'algorithmes utilisables avec chaque type de conteneurs dépend directement du conteneur considéré.

---

[1] On parle aussi parfois de *collection*.

[2] cf. 6.3.

D CORRESPONDANCE UML ET C++

D.1 STRUCTURE STATIQUE

D.1.1 Classes

D.1.1.1 Classe

`NomClasse` : déclaration de la classe.

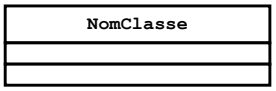

Figure D.1 : représentation UML d'une classe

Fichier *NomClasse.h* :

```
class NomClasse
{
   ...
};
```

D.1.1.2 Classe abstraite

`NomClasseItalique` (classe abstraite) : aucun mot-clef ou modificateur existant en C++.

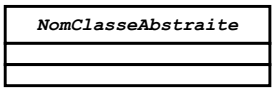

Figure D.2 : représentation UML d'une classe abstraite

Fichier *NomClasseAbstraite.h* :

```
class NomClasseAbstraite
{
   ...
};
```

D.1.1.3 Classe paramétrable

`NomParametre` (classe générique / patron) : utilisation du mécanisme de généricité (template).

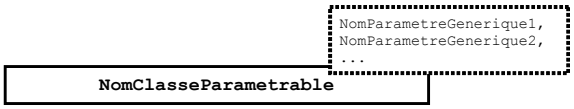

Figure D.3 : représentation UML d'une classe paramétrable

Fichier *NomClasseParametrable.h* :

```
template <class NomParametreGenerique1, class NomParametreGenerique2, ...>
class NomClasseParametrable
{
   ...
};
```

D.1.2 Paquetage

`NomPaquetage` : regroupement des entités appartenant au paquetage dans un espace de noms.

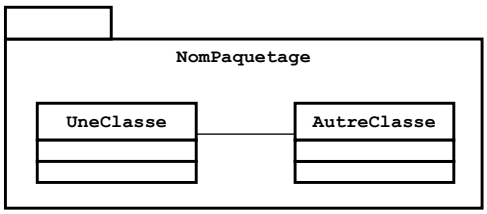

Figure D.4 : représentation UML d'un paquetage

```
namespace NomPaquetage
{
   class UneClasse
   {
      ...
   };

   class AutreClasse
   {
      ...
   };
};
```

D.1.3 Membres

D.1.3.1 Membre

`nomAttribut` : déclaration d'une variable associée à la classe ;
`NomMethode(...)` : déclaration d'une fonction associée à la classe.

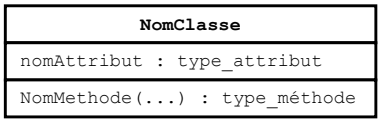

Figure D.5 : représentation UML d'une classe et de ses membres

Fichier *NomClasse.h* :

```
class NomClasse
{
   type_attribut nomAttribut;
   type_méthode NomMethode(...);
};
```

Attribut de type *instance de classe* / *objet* : cf. associations (D.2).

D.1.3.2 Implémentation des méthodes

Fichier *NomClasse.cpp* :

```
#include "NomClasse.h"

type_méthode NomClasse::NomMethode(...)
{
   ...
}
```

D.1.3.3 Visibilité

\- (privé) : section `private` ;
\# (protégé) : section `protected` ;
\+ (public) : section `public`.

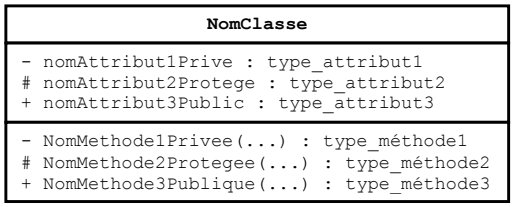

Figure D.6 : représentation UML d'une classe, de ses membres et de leur visibilité

Fichier *NomClasse.h* – regroupement des types de visibilités par section ; ordre par convention : `public` / `protected` / `private` ; déclaration hors section (au tout début) : `private` (par défaut) :

```
class NomClasse
{
   public:
      type_attribut3 nomAttribut3Public;
      type_méthode3 NomMethode3Publique(...);
   protected:
      type_attribut2 nomAttribut2Protege;
      type_méthode2 NomMethode2Protegee(...);
   private:
      type_attribut1 nomAttribut1Prive;
      type_méthode1 NomMethode1Privee(...);
};
```

D.1.3.4 Membre statique

nomMembreSouligne (statique / membre de classe) : modificateur `static`.

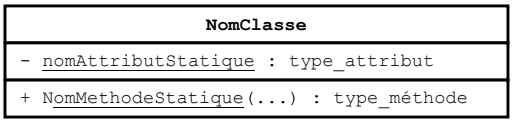

Figure D.7 : représentation UML d'une classe avec des membres statiques

Fichier *NomClasse.h* :

```
class NomClasse
{
   private:
      static type_attribut nomAttributStatique;
   public:
      static type_méthode NomMethodeStatique(...);
};
```

D.1.3.5 Méthodes virtuelle et virtuelle pure

`NomMethode` (virtuelle) : modificateur `virtual` ;
`NomMethodeItalique` (virtuelle pure / abstraite) : modificateur `virtual` et initialisation `= 0`.

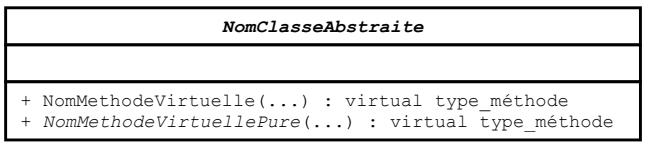

Figure D.8 : représentation UML d'une classe avec une méthode virtuelle et une méthode virtuelle pure

Fichier *NomClasseAbstraite.h* :

```
class NomClasseAbstraite
{
   public:
      virtual type_méthode NomMethodeVirtuelle(...);
      virtual type_méthode NomMethodeVirtuellePure(...) = 0;
};
```

D.2 ASSOCIATIONS

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet.
Plusieurs cas possibles : association simple, agrégation et composition.

Nb : Pas de règles à respecter impérativement, juste des conventions de codage visant à respecter les concepts POO.

D.2.1 Association « simple »

Utilisation d'une instance d'une classe par une autre classe (*a besoin de* / *nécessite*) en tant qu'attribut-objet.
Aucune responsabilité vis-à-vis de la création de l'instance ou de la destruction de l'instance.

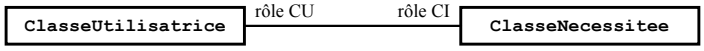

Figure D.9 : représentation UML d'une association

D.2.1.1 Association unidirectionnelle 1-1

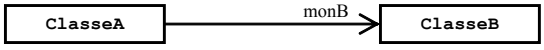

Figure D.10 : représentation UML d'une association unidirectionnelle 1-1

Implémentation :
Pour la classe utilisatrice (`ClasseA`), déclaration en dynamique d'un objet de la classe nécessitée (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).
Initialisation de l'objet par récupération d'une référence externe à la classe utilisatrice via l'une de ses méthodes, généralement le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

class ClasseA
{
   public:
      ClasseA(ClasseB*);
   private:
      ClasseB* monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   ...
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"

ClasseA::ClasseA(ClasseB* b)
{
   monB = b;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

Autre implémentation :

Pour la classe utilisatrice (`ClasseA`), déclaration du type `class` de l'objet + déclaration en dynamique d'un objet de la classe nécessitée (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).

Initialisation de l'objet par récupération d'une référence externe à la classe utilisatrice via l'une de ses méthodes, généralement le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseB;

class ClasseA
{
   /* idem */
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   ...
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"

/* idem */
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

D.2.1.2 Association unidirectionnelle 1-plusieurs

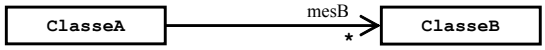

Figure D.11 : représentation UML d'une association unidirectionnelle 1-plusieurs

Implémentation :

Pour la classe utilisatrice (`ClasseA`), déclaration en dynamique d'un tableau d'objets de la classe nécessitée (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).

Initialisation de chaque objet du tableau par récupération d'une référence externe à la classe utilisatrice via l'une de ses méthodes, généralement le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

#define nombre 10

class ClasseA
{
   public:
      ClasseA(ClasseB*[]);
   private:
      ClasseB* mesB[nombre];
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   ...
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"

ClasseA::ClasseA(ClasseB* bs[])
{
   for ( int i=0 ; i<nombre ; i++ )
      mesB[i] = bs[i];
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

D.2.1.3 Association bidirectionnelle

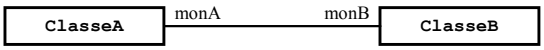

Figure D.12 : représentation UML d'une association bidirectionnelle

Implémentation :
Pour chaque classe, déclaration du type `class` de l'objet + déclaration en dynamique d'un objet de l'autre classe en tant qu'attribut dans la déclaration de classe (*.h*).
Pour chaque classe, initialisation de l'objet par récupération d'une référence externe à la classe via l'une de ses méthodes, sauf le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseB;

class ClasseA
{
   public:
      ClasseA();
      void UneMethodeA(ClasseB*);
   private:
      ClasseB* monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseA;

class ClasseB
{
   public:
      ClasseB();
      void UneMethodeB(ClasseA*);
   private:
      ClasseA* monA;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"
#include <stddef.h>

ClasseA::ClasseA()
{
   monB = NULL;
}

void ClasseA::UneMethodeA(Classeb* b)
{
   monB = b;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
#include "ClasseA.h"
#include <stddef.h>

ClasseB::ClasseB()
{
   monA = NULL;
}

void ClasseB::UneMethodeB(ClasseA* a)
{
   monA = a;
}
```

Nb : Un pointeur `NULL` est un pointeur non-initialisé ; le mot-clef `NULL` est défini dans la bibliothèque *stddef.h*.

D.2.1.4 Association réflexive

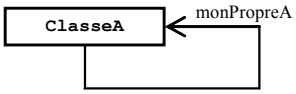

Figure D.13 : représentation UML d'une association réflexive

Implémentation :
Déclaration en dynamique d'un objet de la même classe en tant qu'attribut dans la déclaration de classe (*.h*).
Initialisation de l'objet par récupération d'une référence externe à la classe via l'une de ses méthodes, sauf le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseA
{
   public:
      ClasseA();
      void UneMethodeA(ClasseA*);
   private:
      ClasseA* monPropreA;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include <stddef.h>

ClasseA::ClasseA()
{
   monPropreA = NULL;
}

void ClasseA::UneMethodeA(ClasseA* a)
{
   monPropreA = a;
}
```

D.2.2 Agrégation

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet avec un couplage fort et des durées de vies distinctes (*est constitué de* / *est fait de* / *fait partie de*) ; appelée aussi *agrégation par référence*.

Responsabilité vis-à-vis de la création de l'instance, même si celle-ci reste partageable avec d'autres instances (même classe ou autre classe) ; responsabilité vis-à-vis de la destruction de l'instance pas obligatoire mais fortement conseillée.

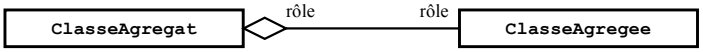

Figure D.14 : représentation UML d'une agrégation

D.2.2.1 Agrégation 1-1

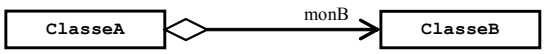

Figure D.15 : représentation UML d'une agrégation 1-1

Implémentation :

Pour la classe agrégat (`ClasseA`), déclaration en dynamique (nda : « par référence ») d'un objet de la classe agrégée (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).

Instanciation explicite de l'objet + destruction explicite (s'il y a) dans les méthodes de la classe agrégat, généralement le constructeur et le destructeur respectivement (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

class ClasseA
{
   public:
      ClasseA();
      ~ClasseA();
   private:
      ClasseB* monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   ...
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"

ClasseA::ClasseA()
{
   monB = new ClasseB();
}

ClasseA::~ClasseA()
{
   delete monB;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

D.2.2.2 Agrégation 1-plusieurs

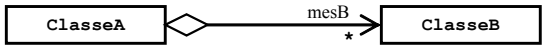

Figure D.16 : représentation UML d'une agrégation 1-plusieurs

Implémentation :

Pour la classe agrégat (`ClasseA`), déclaration en dynamique (nda : « par référence ») d'un tableau d'objets de la classe agrégée (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).

Instanciation explicite de chaque objet du tableau + destruction explicite (s'il y a) dans les méthodes de la classe agrégat, généralement le constructeur et le destructeur respectivement (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

#define nombre 10

class ClasseA
{
   public:
      ClasseA();
      ~ClasseA();
   private:
      ClasseB* mesB[nombre];
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   public:
      ClasseB();
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"

ClasseA::ClasseA()
{
   for ( int i=0 ; i<nombre ; i++ )
      mesB[i] = new ClasseB();
}

ClasseA::~ClasseA()
{
   for ( int i=0 ; i<nombre ; i++ )
      delete mesB[i];
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"

ClasseB::ClasseB()
{
   ...
}
```

D.2.2.3 Agrégation navigable

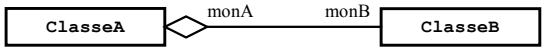

Figure D.17 : représentation UML d'une agrégation navigable

Implémentation :

Pour chaque classe, déclaration du type `class` de l'objet + déclaration en dynamique (nda : « par référence ») d'un objet de l'autre classe en tant qu'attribut dans la déclaration de classe (*.h*).

Pour la classe agrégat (`ClasseA`), instanciation explicite de l'objet + destruction explicite (s'il y a) dans les méthodes de la classe agrégat, généralement le constructeur et le destructeur respectivement (*.cpp*).

Pour la classe agrégée (`ClasseB`), initialisation de l'objet par récupération d'une référence externe à la classe via l'une de ses méthodes, généralement le constructeur (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseB;

class ClasseA
{
   public:
      ClasseA();
      ~ClasseA();
   private:
      ClasseB* monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseA;

class ClasseB
{
   public:
      ClasseB(ClasseA*);
   private:
      ClasseA* monA;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"

ClasseA::ClasseA()
{
   monB = new ClasseB(this);
}

ClasseA::~ClasseA()
{
   delete monB;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
#include "ClasseA.h"

ClasseB::ClasseB(ClasseA* a)
{
   monA = a;
}
```

D.2.2.4 Agrégation réflexive

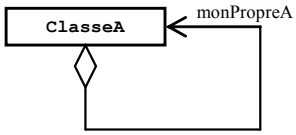

Figure D.18 : représentation UML d'une agrégation réflexive

Implémentation :
Déclaration en dynamique (nda : « par référence ») d'un objet de la même classe en tant qu'attribut dans la déclaration de classe (*.h*).
Instanciation explicite dans l'une de ses méthodes, sauf le constructeur, + destruction explicite (s'il y a) dans une méthode, généralement le destructeur (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseA
{
   public:
      ClasseA();
      ~ClasseA();
      void UneMethodeA();
   private:
      ClasseA* monPropreA;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include <stddef.h>

ClasseA::ClasseA()
{
   monPropreA = NULL;
}

ClasseA::~ClasseA()
{
   if ( monPropreA != NULL )
      delete monPropreA;
}

void ClasseA::UneMethodeA()
{
   monPropreA = new ClasseA();
}
```

D.2.3 Composition

Utilisation d'une instance d'une classe par une autre classe en tant qu'attribut-objet avec un couplage fort et des durées de vies identiques (*est composé de*) ; appelée aussi *agrégation par valeur*.

Responsabilité complète vis-à-vis de la création de l'instance ainsi que de la destruction de l'instance ; instance partageable avec aucune autre instance (même classe ou autre classe).

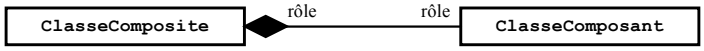

Figure D.19 : représentation UML d'une composition

D.2.3.1 Composition 1-1

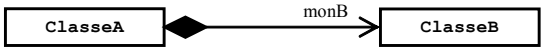

Figure D.20 : représentation UML d'une composition 1-1

Implémentation :

Pour la classe composite (`ClasseA`), déclaration en statique (nda : « par valeur ») d'un objet de la classe composant (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).

Instanciation implicite de l'objet dans le constructeur, via le constructeur par défaut de la classe composant, + destruction implicite dans le destructeur, via le destructeur de la classe composant (*.cpp*) ; éventuellement, instanciation explicite de l'objet dans la liste d'initialisation du constructeur, via une surcharge du constructeur de la classe composant (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

class ClasseA
{
   private:
      ClasseB monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   public:
      ClasseB();
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
...
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"

ClasseB::ClasseB()
{
   ...
}
```

Nb : Possibilité de déclarer en dynamique (plutôt qu'en statique) l'objet de la classe composant en tant qu'attribut dans la déclaration de classe (*.h*), instanciation explicite dans le constructeur + destruction explicite dans le destructeur (*.cpp*).

Représentation alternative : classe imbriquée → instance partageable avec autre instance (même classe ou autre classe) + classe composant utilisable avec aucune autre classe.

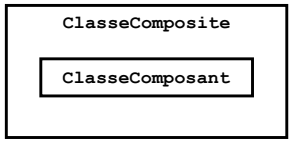

Figure D.21 : représentation UML d'une composition sous forme de classe imbriquée

Implémentation :

Pour la classe composite (`ClasseA`), déclaration + définition de la classe composant (`ClasseB`), puis déclaration en statique (nda : « par valeur ») d'un objet de la classe composant en tant qu'attribut dans la déclaration de classe (*.h*).

Instanciation implicite de l'objet dans le constructeur, via le constructeur par défaut de la classe composant, + destruction implicite dans le destructeur, via le destructeur de la classe composant (*.cpp*) ; éventuellement,

instanciation explicite de l'objet dans la liste d'initialisation du constructeur, via une surcharge du constructeur de la classe composant (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseA
{
   private:
      class ClasseB
      {
         ClasseB() {...}
      };
      ClasseB monB;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
...
```

D.2.3.2 Composition 1-plusieurs

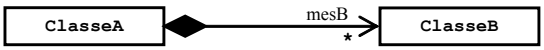

Figure D.22 : représentation UML d'une composition 1-plusieurs

Implémentation :
Pour la classe composite (`ClasseA`), déclaration en statique (nda : « par valeur ») d'un tableau d'objets de la classe composant (`ClasseB`) en tant qu'attribut dans la déclaration de classe (*.h*).
Instanciation implicite de chaque objet du tableau dans le constructeur, via le constructeur par défaut de la classe composant, + destruction implicite dans le destructeur, via le destructeur de la classe composant (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

#define nombre 10

class ClasseA
{
   private:
      ClasseB mesB[nombre];
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   public:
      ClasseB();
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
...
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"

ClasseB::ClasseB()
{
   ...
}
```

D.3 SPÉCIALISATIONS / GÉNÉRALISATIONS

Spécialisation : d'une super-classe vers une sous-classe ;
Généralisation : d'une sous-classe vers une super-classe.

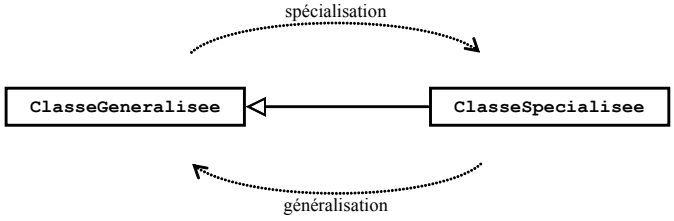

Figure D.23 : représentation UML d'une spécialisation / généralisation

D.3.1 Spécialisation à partir d'une classe

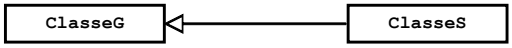

Figure D.24 : représentation UML d'une spécialisation à partir d'une classe

Implémentation :
Utilisation du mécanisme de l'héritage avec le mode de dérivation adéquat (`private` / `protected` / `public`).

Fichier *ClasseG.h* :

```
class ClasseG
{
   ...
};
```

Fichier *ClasseS.h* :

```
#include "ClasseG.h"

class ClasseS : mode_dérivation ClasseG
{
   ...
};
```

D.3.2 Spécialisation à partir de plusieurs classes

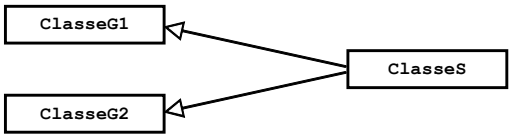

Figure D.25 : représentation UML d'une spécialisation à partir de plusieurs classes

Implémentation :
Utilisation du mécanisme de l'héritage multiple avec le mode de dérivation adéquat pour chaque super-classe (`private` / `protected` / `public`).

Fichier *ClasseG1.h* :

```
class ClasseG1
{
   ...
};
```

Fichier *ClasseG2.h* :

```
class ClasseG2
{
   ...
};
```

Fichier *ClasseS.h* :

```
#include "ClasseG1.h"
#include "ClasseG2.h"

class ClasseS : mode_dérivationG1 ClasseG1, mode_dérivationG2 ClasseG2 {
   ...
};
```

D.4 CLASSE D'ASSOCIATION

Évolution d'une association vers le concept de classe, possédant à la fois les caractéristiques d'une association et celles d'une classe.

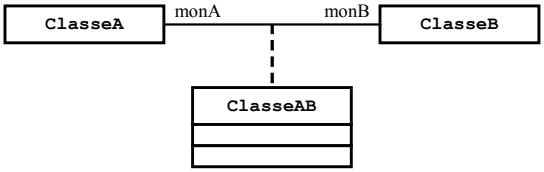

Figure D.26 : représentation UML d'une classe d'association

Implémentation :

Pour la classe d'association (`ClasseAB`), déclaration comme une classe classique + déclaration en dynamique d'un objet de chaque classe associée (`ClasseA` et `ClasseB`) en tant qu'attributs dans la déclaration de classe (*.h*)

Pour chacune des classes associées, au regard de la navigabilité, déclaration en tant qu'attribut (*.h*) + initialisation (*.cpp*) d'un objet de l'autre classe comme dans le cas d'une association simple.

Fichier *ClasseA.h* :

```
class ClasseB;

class ClasseA
{
   public:
      ClasseA();
      void UneMethodeA(ClasseB*);
   private:
      ClasseB* monB;
};
```

Fichier *ClasseB.h* :

```
class ClasseA;

class ClasseB
{
   public:
      ClasseB();
      void UneMethodeB(ClasseA*);
   private:
      ClasseA* monA;
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"
#include <stddef.h>

ClasseA::ClasseA()
{
   monB = NULL;
}

void ClasseA::UneMethodeA(ClasseB* b)
{
   monB = b;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
#include "ClasseA.h"
#include <stddef.h>

ClasseB::ClasseB()
{
   monA = NULL;
}

void ClasseB::UneMethodeB(ClasseA* a)
{
   monA = a;
}
```

Fichier *ClasseAB.h* :

```
#include "ClasseA.h";
#include "ClasseB.h";

class ClasseAB
{
   public:
      ClasseAB(ClasseA*, ClasseB*);
   private:
      ClasseA* unA;
      ClasseB* unB;
};
```

Fichier *ClasseAB.cpp* :

```
#include "ClasseAB.h"

ClasseAB::ClasseAB(ClasseA* a, ClasseB* b)
{
   unA = a;
   unB = b;
}
```

D.5 DÉPENDANCES

Instanciation, utilisation ou tout autre besoin sémantique entre deux classes ne conditionnant ni n'apportant de signification quant à la structure interne de ces classes (*utilise* / *est abstraction de* / *est lié à* / *a permission sur*).

Concept très général pouvant signifier différents types de relations entre une classe A et une classe B, généralement caractérisé par l'usage d'un stéréotype ; exemples :

- l'instanciation d'un objet de B dans une méthode de A autre que le constructeur de A (stéréotype `<<instanciate>>`) ;
- l'utilisation d'une référence sur une instance de B comme paramètre d'entrée d'une méthode de A (stéréotype `<<create>>`) ;
- l'utilisation par A d'une méthode statique de B (stéréotype `<<call>>`) ;
- l'« instanciation » d'une classe générique B avec définition complète des types des paramètres pour créer une classe B (stéréotype `<<bind>>`) ;
- la déclaration de A comme amie de B (stéréotype `<<permit>>`) ;
- …

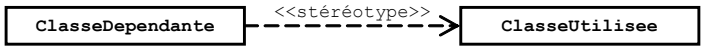

Figure D.27 : représentation UML d'une dépendance

Implémentation relative au type de dépendance (`<<stéréotype>>`), au contexte, etc.

D.5.1 Dépendance du type <<instanciate>>

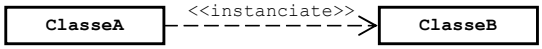

Figure D.28 : représentation UML d'une dépendance du type <<instanciate>>

Implémentation :
Dans l'une des méthodes de la classe dépendante (`ClasseA`), déclaration, le plus souvent en dynamique, + instanciation (`<<instanciate>>`) d'un objet de la classe instanciée (`ClasseB`), là où il est nécessité (*.cpp*), et pas en tant qu'attribut.

Fichier *ClasseA.h* :

```
class ClasseA
{
   public:
      void UneMethodeA();
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   ...
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"

void ClasseA::UneMethodeA()
{
   ClasseB* unB;
   unB = new ClasseB();
   ...
   delete unB;
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

D.5.2 Dépendance du type <<create>>

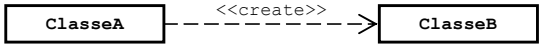

Figure D.29 : représentation UML d'une dépendance du type <<create>>

Implémentation :
Pour l'une des méthodes de la classe dépendante (`ClasseA`), déclaration en dynamique d'un paramètre (entrée/sortie) du type de la classe utilisée (`ClasseB`) et utilisation (`<<create>>`) de la référence sur ce paramètre (*.cpp*).

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

class ClasseA
{
   public:
      void UneMethodeA(ClasseB*);
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   public:
      void UneMethodeB();
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"

void ClasseA::UneMethodeA(ClasseB* b)
{
   b->UneMethodeB();
   ...
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"

void ClasseB::UneMethodeB()
{
   ...
}
```

D.5.3 Dépendance du type <<call>>

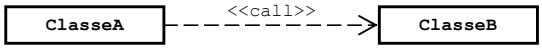

Figure D.30 : représentation UML d'une dépendance du type <<call>>

Implémentation :
Pour la classe dépendante (`ClasseA`), appel (`<<call>>`) d'une méthode statique de la classe utilisée (`ClasseB`) (*.cpp*).

Fichier *ClasseA.h* :

```
class ClasseA
{
   public:
      void UneMethodeA();
};
```

Fichier *ClasseB.h* :

```
class ClasseB
{
   public:
      static void UneMethodeB();
};
```

Fichier *ClasseA.cpp* :

```
#include "ClasseA.h"
#include "ClasseB.h"

void ClasseA::UneMethodeA()
{
   ...
   ClasseB::UneMethodeB();
   ...
}
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"

void ClasseB::UneMethodeB()
{
   ...
}
```

D.5.4 Dépendance du type <<bind>>

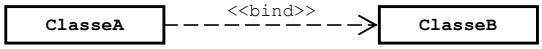

Figure D.31 : représentation UML d'une dépendance du type <<bind>>

Implémentation :
La classe paramétrée (`ClasseA`), est liée (`<<bind>>`) à la classe générique (`ClasseB`) par instanciation du modèle avec des paramètres définis.

Fichier *ClasseA.h* :

```
#include "ClasseB.h"

typedef ClasseB<int> ClasseA;
```

Fichier *ClasseB.h* :

```
template <class T>
class ClasseB
{
   ...
};
```

Fichier *ClasseB.cpp* :

```
#include "ClasseB.h"
...
```

E RÉFÉRENCE

E.1 PROGRAMMATION ORIENTÉE OBJET

E.1.1 Généralités

Le terme **objet** désigne une entité informatique spécifique, qui est une représentation abstraite simplifiée d'un objet concret et abstrait du monde réel ou virtuel. Un objet appartient à une famille d'objets et possède des variables associées et des fonctions associées ; en terminologie objet, on parle respectivement de **classe** (famille d'objets), **attributs** [1] (variables) et **méthodes** (fonctions).

Lorsque l'on fait référence à une classe, le terme de **membres** désigne les attributs et les méthodes de cette classe.

La programmation orientée objet (POO) peut être définie par les trois grands principes suivants :

- l'encapsulation ;
- l'héritage de classe ;
- le polymorphisme.

E.1.2 Principes

E.1.2.1 L'encapsulation

L'**encapsulation** consiste à préserver la valeur des attributs de l'objet ; ceux-ci ne sont accessibles de l'extérieur de la classe que par l'intermédiaire des méthodes de la classe. Les attributs sont dits alors *privés* (accès direct restreint à la classe seulement) ou *protégés* (accès direct restreint à la classe et aux éventuelles sous-classes), alors que les méthodes sont généralement *publiques* [2] (accès possible à l'intérieur et à l'extérieur de la classe).

E.1.2.2 L'héritage de classe

L'**héritage** de classe permet de créer une nouvelle classe à partir d'une classe déjà existante. Cela permet ainsi de préciser, ou mettre à jour une classe.

Du fait de l'héritage, la nouvelle classe possède automatiquement des membres copiés sur ceux de la classe déjà existante [3], lesquels peuvent être complétés par de nouveaux attributs et nouvelles méthodes. Dans le cadre de cet héritage, la classe déjà existante est appelée la *super-classe* ou *classe de base*, et la nouvelle classe est appelée la *sous-classe* ou *classe dérivée* / *classe héritée*.

E.1.2.3 Le polymorphisme

Le **polymorphisme** [4] est caractérisé par la définition de plusieurs méthodes homonymes, mais dont le comportement et/ou la signature [5] diffèrent, ou bien qui s'appliquent à des types d'objets distincts.

Il existe plusieurs types de polymorphismes :

- polymorphisme ad hoc : 2 classes différentes possèdent chacune une méthode homonyme (nom et signature identiques) mais dont le comportement est différent (adapté à chaque classe) ; généralement appelé *surcharge* [6] ;

[1] Parfois appelées aussi *propriétés* dans certains langages.
[2] Une méthode peut cependant être privée si elle est destinée à être appelée uniquement par d'autres méthodes de la classe.
[3] Uniquement si ceux-ci ont été déclarés publics ou protégés.
[4] Mot d'étymologie grecque signifiant « peut prendre plusieurs formes ».
[5] La signature d'une méthode correspond au nombre et au type d'arguments (paramètres d'entrée) de la méthode.
[6] Surcharge (fr) ≡ overloading (eng).

- polymorphisme d'héritage : 1 sous-classe hérite d'une méthode [1] de la super-classe (nom et signature de la méthode identiques), mais dont le comportement est différent (adapté à la sous-classe par redéfinition de la méthode) ; aussi appelé *spécialisation* [2]. Ce type de polymorphisme est mis en œuvre lors de l'utilisation des propriétés de la généralisation dans une hiérarchie de classes.
- polymorphisme paramétrique : 1 classe possède plusieurs définitions d'une méthode homonyme mais dont la signature est différente ; aussi appelé *généricité* [3].

Par simplification, on parle souvent de surcharge pour n'importe quel type de polymorphisme.

E.2 EXÉCUTION ET RESSOURCES

L'un des rôles du système d'exploitation est la gestion des ressources, et notamment de la mémoire vive disponible physiquement sur une machine. En effet, celle-ci est commune – on parle de *mémoire centrale* –, et est partagée entre le système d'exploitation [4] et l'ensemble des programmes en cours d'exécution.

Pour développer efficacement il est donc extrêmement important de se préoccuper de l'utilisation mémoire des programmes créés, soit donc de la manière dont les différentes allocations nécessaires sont réalisées.
Il s'agit donc de développer en optimisant l'utilisation de mémoire vive selon deux axes :

- allouer à un programme la mémoire nécessaire pour s'exécuter de manière correcte ;
- minimiser la consommation de mémoire d'un programme en libérant celle-ci une fois utilisée.

E.2.1 Allocation de mémoire

Dans un souci d'optimisation, trois stratégies d'allocation différentes existent, chacune correspondant à une partie – appelée segment – de la mémoire allouée par le système d'exploitation à un programme :

- allocation statique, segment *text* ;
- allocation sur la pile, segment *stack* [5] ;
- allocation dans le tas, segment *heap* [6].

Pour chaque cas d'utilisation de mémoire (stockage d'une information, traitement d'une procédure, création d'un objet, etc.), l'une des stratégies d'allocation est donc utilisée. Si certaines utilisations suivent une stratégie prédéfinie, d'autres sont laissées au choix du programmeur.

E.2.1.1 Allocation statique

L'**allocation statique** consiste à prévoir l'espace mémoire nécessaire dès la phase de développement ; il est alors réservé avant l'exécution du programme, dans le segment text, lorsque le celui-ci se charge en mémoire, juste avant d'être exécuté.
Lors de l'exécution, aucune modification n'est apportée à cette allocation ; l'espace mémoire est libéré à la terminaison du programme.

L'allocation statique offre ainsi un gain certain au niveau des performances, puisqu'à l'exécution la mémoire est immédiatement disponible.
Cependant, elle monopolise une partie de la mémoire centrale pendant toute la durée de l'exécution, et ce même si un espace mémoire est très peu voire pas utilisé par le programme.

Les variables et instances *globales*, les déclarations *externes*, ainsi que les membres *statiques* suivent une allocation statique.

E.2.1.2 Allocation sur la pile

L'**allocation sur la pile** consiste à allouer/désallouer automatiquement de la mémoire à un programme au fur et à mesure des besoins en utilisant le segment stack, dont le stockage des informations est organisé selon un principe de pile de type LIFO [7].

[1] Parmi un ensemble d'autres membres.
[2] Dans le cas d'une méthode virtuelle spécialisation (fr) ≡ overriding (eng), et dans le cas d'une méthode non-virtuelle redéfinition (fr) ≡ redefinition (eng).
[3] Généricité (fr) ≡ overloading (eng).
[4] Le système d'exploitation n'est rien d'autre qu'une suite de logiciels coopératifs dont le rôle est d'exploiter la machine physique.
[5] Stack (eng) ≡ pile (fr).
[6] Heap (eng) ≡ tas (fr).
[7] Last In First Out (eng) ≡ Dernier Arrivé Premier Sorti (fr).

Chaque besoin en mémoire devant utiliser la pile réserve un espace mémoire dans ce segment. Cet espace est positionné dans la zone libre du segment, située sur le dessus de la pile, juste après les précédents espaces mémoire antérieurement réservés, à partir du premier espace mémoire libre ; on dit qu'il est *empilé*.

Lorsque l'espace mémoire n'est plus nécessité, il peut être libéré. Seul l'espace mémoire situé sur le dessus de la pile peut être libéré ; on dit qu'il est *dépilé*. Un espace mémoire situé sous un autre ne peut en aucune manière être libéré ; il faut attendre que toutes les réservations postérieures soient libérées.

Les besoins en mémoire sont donc généralement alloués et libérés en suivant le fil d'exécution décrit dans le code source, les uns à la suite des autres, suivant les adresses décroissantes.

L'espace mémoire étant alloué/libéré de manière automatique au cours de l'exécution, on peut parler d'allocation dynamique pour les allocations utilisant la pile.

Ex. : Des allocations dynamiques sur la pile.

```
void procedureA()
{
   char a;
}

int main(void)
{
   char x;
   procedureA();
   char y;

   return 0;
}
```

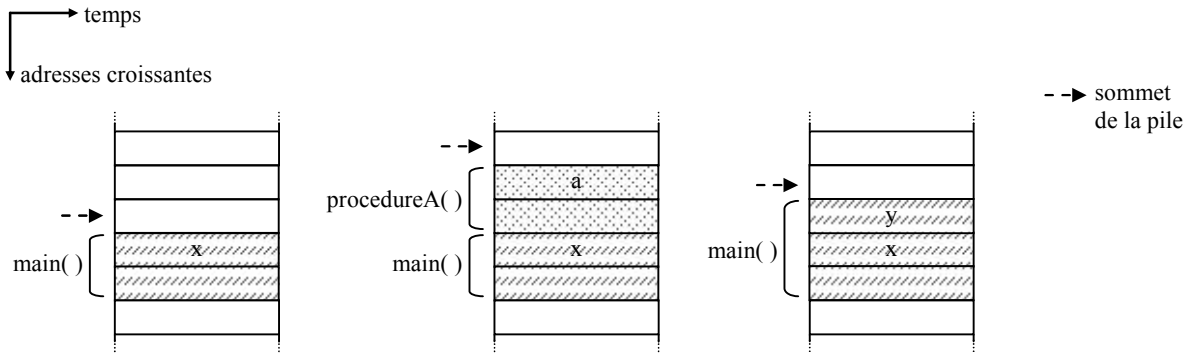

Figure E.1 : exemple d'allocations dynamiques sur la pile

En réalité, lors de la phase de compilation, il est facile de prévoir toutes les allocations sur la pile qui seront réalisées lors de l'exécution ainsi que leur position relative. Dans un souci d'optimisation, une allocation peut se voir réserver un espace mémoire situé plus haut sur la pile qu'au niveau du sommet, laissant ainsi des espaces libres, car le compilateur sait qu'ils seront nécessaires au cours de l'exécution.

L'allocation sur la pile peut donc être aussi appelée **allocation statique sur la pile**, en comparaison avec l'allocation dynamique dans le tas, et par rapprochement avec l'allocation statique dans le segment text.

Ex. : Des allocations statiques sur la pile.

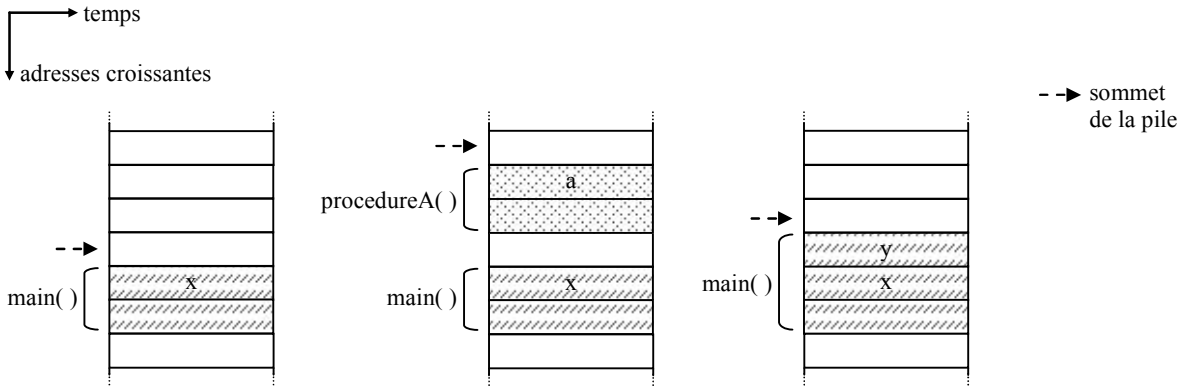

Figure E.2 : exemple d'allocations statiques sur la pile

L'allocation sur la pile offre d'excellentes performances, puisqu'il est très facile de retrouver le sommet de la pile, et que les besoins en mémoire sont évalués et gérés en cours d'exécution ; de plus, elle assure l'absence de fuite de mémoire [1] puisque les libérations sont réalisées automatiquement.

Par contre, il est nécessaire de connaître à l'avance la taille exacte de la réservation nécessitée, information qui peut évoluer ou qui n'est parfois connue que durant l'exécution [2] ; en ce cas, l'allocation sur la pile est donc inadéquate.

Les *appels de fonctions et de méthodes* sont toujours gérés suivant une allocation sur la pile [3]. Les variables et instances *locales*, définies dans la portée d'une procédure, sont aussi allouées sur la pile ; par extension, il en est de même pour toutes les définitions locales à un bloc de code. On peut aussi les appeler variables et instances *lexicales*, car elles sont spécifiques à la procédure, ou variables et instances *automatiques* [4] puisque l'espace mémoire nécessaire est alloué à la déclaration et libéré automatiquement à la fin de la procédure.

E.2.1.3 Allocation dans le tas

L'**allocation dans le tas** consiste à allouer/désallouer explicitement de la mémoire à un programme lorsque cela est nécessaire en utilisant le segment heap, qui ne suit aucune organisation ou logique particulière.

Chaque besoin en mémoire destiné à utiliser le tas nécessite la réservation d'un espace mémoire dans ce segment, réservation qui doit figurer explicitement dans le code source [5], et qui est donc du ressort du développeur. Cet espace est positionné dans toute zone libre du segment suffisamment grande pour accueillir la réservation nécessitée, généralement la première à partir du début du tas.

Lorsque l'espace mémoire n'est plus nécessité, il peut être libéré. Cette libération doit aussi figurer explicitement dans le code source [6] [7].

Ex. : Des allocations dynamiques dans le tas.

```
int main(void)
{
   char* x = new char;
   char* y = new char;
   char* z = new char;
   delete y;
   char* t = new char[2];
   char* u = new char;
   delete x, z, u;
   delete[] t;

   return 0;
}
```

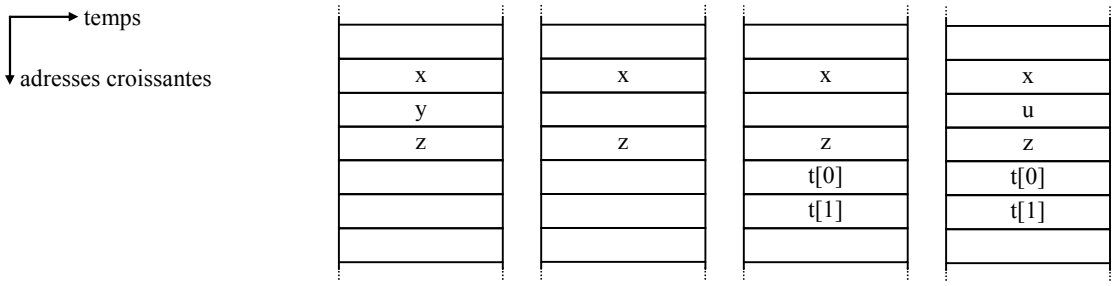

Figure E.3 : exemple d'allocations dynamiques dans le tas

L'allocation dans le tas est dynamique, et offre donc de bonnes performances, à condition de libérer chaque espace mémoire lorsqu'il n'est plus utilisé. C'est de plus une solution extrêmement souple qui peut s'adapter aux desiderata du programmeur.

Néanmoins, cette souplesse a pour contre-partie de demander une grande rigueur, car l'allocation et la libération sont sous la responsabilité du programmeur. Tout espace mémoire alloué, même inutilisé, reste réservé jusqu'à

[1] Fuite de mémoire (fr) ≡ memory leak (eng).

[2] C'est ce qui explique qu'il faut toujours spécifier la taille d'un tableau lors d'une déclaration statique d'un tableau (`type tableau[taille];`).

[3] cf. E.2.2.

[4] Avant la normalisation ANSI, le modificateur '`auto`' était nécessaire pour déclarer une variable ou instance locale (cf. A.2.6).

[5] Opérateur `new`.

[6] Opérateur `delete`.

[7] Certains langages, comme Java et C#, sont dotés d'un processus dit de « ramasse-miettes » (garbage collector (eng)) dont le rôle est de libérer automatiquement les espaces mémoires alloués dans le tas qui ne sont plus nécessités. Le développeur n'a donc plus à s'en soucier.

libération explicite ou à la terminaison du programme, et pas à la fin de la procédure ; les risques de fuite de mémoire sont donc réels, surtout si le programme s'exécute sur une longue durée. De plus, une allocation dans le tas est très coûteuse en ressources, puisqu'il faut parcourir le tas à la recherche d'une zone de mémoire libre d'une capacité suffisante. Enfin, lors d'un grand nombre d'allocation/désallocation, le tas subit un phénomène de fragmentation préjudiciable aux performances.

Toutes les *déclarations dynamiques*, réalisées à l'aide des pointeurs et de `new`/`delete`, sont allouées dans le tas.

E.2.2 Appel de procédure

Lors de l'exécution d'un programme, chaque procédure [1] d'un programme occupe un segment de la mémoire allouée pour le programme qui est indépendant des autres.

Chaque procédure se voit allouer de la mémoire dynamiquement sur la pile lors de son appel en suivant le fil d'exécution du code source. La mémoire nécessitée par une procédure est empilée sur le dessus de la pile, et est libérée lorsque la procédure se termine. Lorsqu'une procédure A appelle une autre procédure B, la procédure A est donc empilée, puis la procédure B est empilée sur la A puisque celle-ci n'est pas terminée ; lorsque la procédure B est terminée, elle est dépilée, laissant alors champ libre à la procédure A pour se terminer à son tour et enfin être dépilée.

Ex. : Une suite d'appels de procédures alloués sur la pile.

```
void procedureB()
{
   ...
}

void procedureC()
{
   ...
}

void procedureA()
{
   procedureB();
   procedureC();
}

int main(void)
{
   procedureA();

   return 0;
}
```

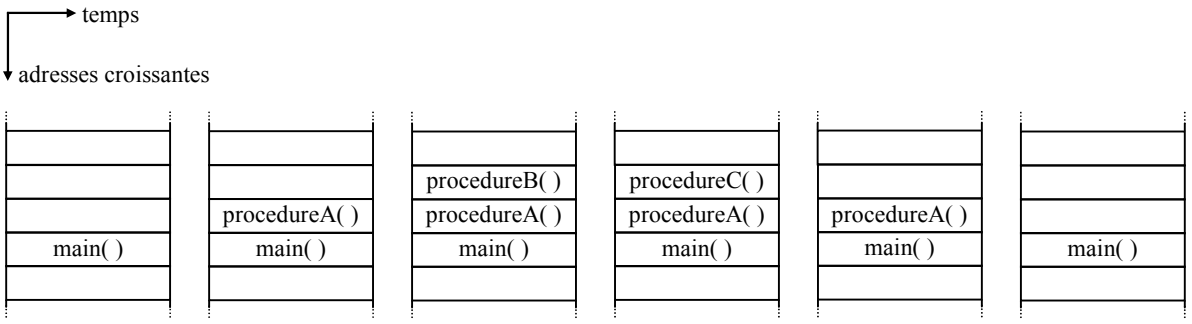

Figure E.4 : appels de procédures alloués sur la pile

Lorsque l'on passe des paramètres d'entrée à une procédure, les valeurs sont évaluées puis recopiées sur la pile, dans le sens inverse de la liste d'arguments. L'espace mémoire nécessaire à la récupération de la valeur renvoyée dans la procédure appelante est réservé sur la pile à la suite des valeurs des paramètres d'entrée, avant même que la procédure soit exécutée.

À la terminaison de la procédure, le paramètre de l'instruction `return` est évalué puis recopié dans cet espace mémoire ; ainsi toutes les allocations sur la pile réalisées « à l'intérieur » de la procédure peuvent être dépilées. Ensuite, après avoir été utilisé, il est dépilé ; enfin, c'est au tour des paramètres d'entrée d'être dépilés.

[1] Fonction ou méthode.

Ex. : Passage de paramètres d'entrée et récupération de la valeur renvoyée lors de l'appel d'une procédure.

```
int procedureA(int p1, int p2)
{
   int a=3;
   return a;
}

int main(void)
{
   int x=1, y=2, z;
   z = procedureA(x, y);

   return 0;
}
```

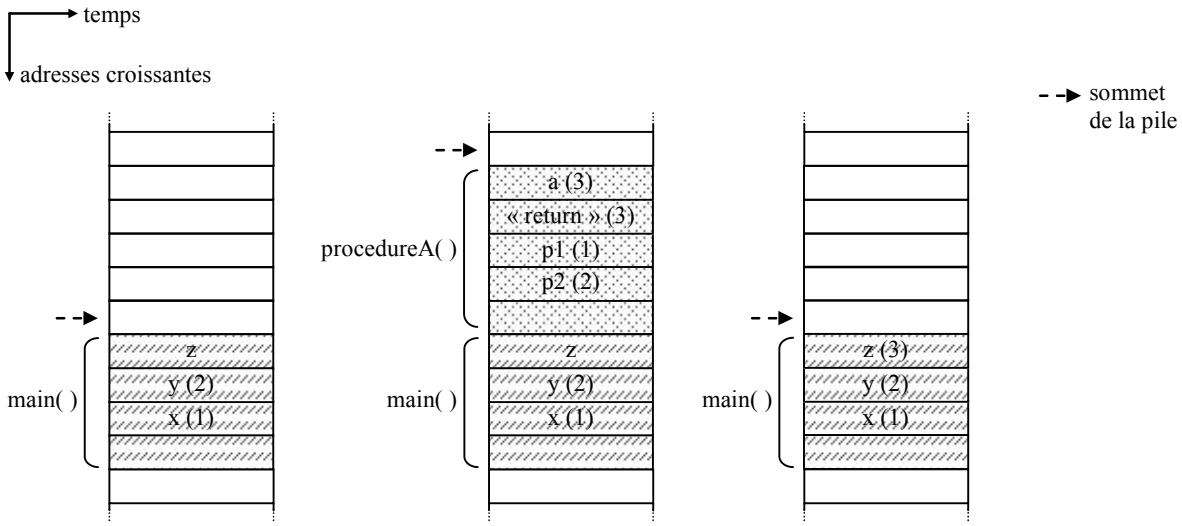

Figure E.5 : passage de paramètres d'entrée et récupération de la valeur renvoyée lors de l'appel d'une procédure

F BIBLIOGRAPHIE

Garreta Henri, *Le langage C++*, Université de la Méditerranée – Faculté des sciences de Luminy, 2005 ;

Infini-software.com, *Le langage C++ ANSI*,
http://www.infini-software.com/Encyclopedie/Developpement/C_Cpp/CppAnsi/French/Index.wp, 2005 ;

Bodeveix Jean-Paul, **Filali Mamoun**, **Sayah Amal**, *Programmation en C++*, InterEditions, 1994 ;

Dancel Alain, *Langage C++*, http://membres.lycos.fr/dancel/, ENAC, 1999 ;

Groupe Steria, *C++*, Institut Aquitain d'Informatique, 1994 ;

Garcia Bruno « Ours blanc des Carpathes™ »,
Le C++ pour les pros et *La librairie standard du C++*, ISIMA, 1999 ;

CPlusPlus.com, *C++ library reference*, http://www.cplusplus.com/reference, 2008 ;

Guéganno Claude, *Programmation (II) – Langage C++*, http://claude.gueganno.free.fr/, 2004 ;

Casteyde Christian, *Cours de C/C++ 1.40.6*, http://casteyde.christian.free.fr/, 2002 ;

Eckel Bruce, *Penser en C++ vol. 1* (*Thinking in C++*), http://bruce-eckel.developpez.com/, 2008 ;

Hardware.fr, *Forum C/C++*, http://forum.hardware.fr/hfr/Programmation/C++/liste_sujet-1.htm, 2006 ;

Hérauville Stéphane, *Cours Langage C++*, CFAI Marcel Sembat – Sotteville-lès-Rouen, 2002 ;

Muller Pierre-Alain, **Gaertner Nathalie**, *Modélisation objet avec UML*, Eyrolles, 2000 ;

Décoret Xavier, *Tutoriel STL*, http://artis.imag.fr/~Xavier.Decoret/, ARTIS – INRIA Rhône-Alpes, 2000 ;

CommentÇaMarche.net, *C++*, http://www.commentcamarche.net/cpp/, 2006 ;

Développez.com, *Langage C++*, http://c.developpez.com/, 2006 ;

Roques Pascal, *UML par la pratique*, Eyrolles, 2003 ;

Bichon Jean, *Cours de Système d'Information – Approche objet*,
Licence informatique – UFR Mathématiques et Informatique – Université Bordeaux 1, 2004 ;

Lachaud Jacques-Olivier, *Programmation C++ – TD et TP*, http://www.lama.univ-savoie.fr/wiki/,
Laboratoire de Mathématiques – UFR SFA – Université Savoie, 2000 ;

Wikipedia – l'encyclopédie libre, *Allocation de mémoire* et *Call stack*,
http://fr.wikipedia.org/ et http://en.wikipedia.org/, 2009 ;

Chang Chain-I, *Assembly Language Programming – C funtion call conventions and the stack*,
http://www.cs.umbc.edu/~chang/cs313.s02/assembly.shtml,
CSEE – Computer Science Electrical Engineering – University of Maryland Baltimore County, 2001 ;

Seleborg Carl, *Cours C++*, http://carl.seleborg.free.fr/cpp/, 2001 ;

MediaDICO, http://www.mediadico.com, 2006 ;

Wikiversité – communauté pédagogique libre, *Langage C++*, http://fr.wikiversity.org/, 2009 ;

Peignot Christophe, *Langage C* et *Langage UML*, http://info.arqendra.net/, 2009.