Université de Montpellier / Faculté Des Sciences

Objets Avancés - Partie C++
HLIN 603

Marianne Huchard

15 janvier 2015

Table des matières

1 Introduction **4**
- 1.1 L'approche par objets . . . 4
  - 1.1.1 les langages à classes . . . 5
  - 1.1.2 Une brève histoire de C++ . . . 6

2 Classes et messages en C++ : le socle de la programmation par objets **7**
- 2.1 Vers et strophes . . . 7
- 2.2 Classes et objets en C++ . . . 8
  - 2.2.1 Le principe de séparation de l'interface et de l'implémentation des classes . . . 8
  - 2.2.2 Encapsulation et protection . . . 10
  - 2.2.3 La création des objets . . . 12
  - 2.2.4 Un (tout petit) peu d'introspection . . . 13
- 2.3 Méthodes et envoi de messages . . . 14
  - 2.3.1 Passage de paramètres et valeur retournée . . . 14
  - 2.3.2 Moi (l'objet courant) . . . 15
  - 2.3.3 Constructeurs et destructeur . . . 16
  - 2.3.4 Accesseurs . . . 18

3 Spécialisation/généralisation et héritage simple **19**
- 3.1 La notion de spécialisation/généralisation et d'héritage . . . 19
- 3.2 Réalisation en C++ . . . 20
  - 3.2.1 Déclaration . . . 20
  - 3.2.2 Construction/Destruction . . . 21
  - 3.2.3 Méthodes . . . 22
  - 3.2.4 Classes et méthodes abstraites . . . 24
  - 3.2.5 Coercition/Affectation . . . 25

4 Initialisation, affectation et conversion **27**
- 4.1 Initialisation versus affectation . . . 28
- 4.2 Les mystères de la conversion élucidés . . . 29
  - 4.2.1 Conversion standard d'un type (classe) vers un super-type (super-classe) . . . 29
  - 4.2.2 Conversion basée sur un constructeur . . . 29
  - 4.2.3 Opérateurs de conversion . . . 30
  - 4.2.4 Ambiguïtés . . . 32
  - 4.2.5 Régle et moralité . . . 32
- 4.3 Surcharge versus redéfinition . . . 33

5 Attributs et méthodes de classes **37**

6 Généricité paramétrique **40**
6.1 Aspects conceptuels . . . 40
6.2 Modèles de fonctions . . . 41
6.3 Modèles de classes . . . 42

7 STL : la librairie standard C++ **49**
7.1 Introduction . . . 49
7.2 Conteneurs . . . 49
7.2.1 Séquences . . . 50
7.2.2 Conteneurs associatifs . . . 52
7.3 Itérateurs . . . 53
7.4 Algorithmes . . . 55
7.5 Classes-Fonctions . . . 57
7.5.1 Classes existantes . . . 57
7.5.2 Classes définies par l'utilisateur . . . 58
7.6 Adaptateurs . . . 58
7.6.1 Adaptateurs de conteneurs . . . 58
7.6.2 Adaptateurs d'itérateurs . . . 58
7.6.3 Adaptateurs de classes-fonctions . . . 58
7.7 Allocateurs . . . 58
7.8 Annexe : les fichiers à inclure . . . 59

8 Spécialisation/généralisation et héritage multiple **61**
8.1 Définition . . . 61
8.2 Discussion des avantages et inconvénients . . . 61
8.2.1 Une meilleure classification . . . 61
8.2.2 Un meilleur partage . . . 63
8.2.3 Conflits de valeurs et conflits de noms . . . 63
8.2.4 Héritage répété . . . 65
8.2.5 Complexité de la classification . . . 65
8.3 Réalisation en C++ . . . 65
8.3.1 Résolution des conflits par désignation explicite . . . 65
8.3.2 Héritage répété versus héritage `virtual` . . . 66
8.3.3 Transmission des paramètres aux constructeurs . . . 68
8.3.4 Ordre d'appel des constructeurs et destructeurs . . . 68

9 Gestion des exceptions **70**
9.1 Introduction . . . 70
9.2 Modélisation . . . 70
9.3 Définition d'exceptions en C++ . . . 71
9.4 Déclaration et signalement . . . 74
9.5 Récupération . . . 74
9.6 Exceptions prédéfinies . . . 76
9.7 Exceptions du mécanisme de gestion d'exceptions . . . 76

10 Contrôle d'accès statique **78**
10.1 Définition et usage 78
10.2 Contrôle d'accès sur les propriétés 79
10.3 La déclaration `friend` 80
10.4 Contrôle d'accès sur les liens d'héritage 81
10.5 La déclaration `using` 82
10.6 Contrôles d'accès lors de la redéfinition 83
10.7 Regard sur la modélisation 84

Chapitre 1

Introduction

1.1 L'approche par objets

L'approche par objets propose de fonder la construction d'un système informatique sur la modélisation des entités, vues sous leurs aspects statiques et dynamiques, qui interagissent pour produire les services attendus, En cela elle se démarque d'une approche basée seulement sur une modélisation des fonctionnalités attendues, généralement décomposées en sous-fonctionnalités jusqu'à l'obtention de fonctions ou de procédures de taille réduite. Cette approche de construction par décomposition fonctionnelle a trouvé rapidement ses limites : création de fonctionnalités redondantes par manque de vision globale, mais surtout faible durée de vie des systèmes par manque de stabilité des constituants. Un logiciel évolue en effet plus dans ses fonctionnalités que dans les entités manipulées, pourvu qu'elles soient d'un niveau d'abstraction suffisant. Pour donner l'exemple courant d'un système bancaire, les entités telles que les comptes ou les clients sont peu amenées à disparaître tandis que des procédures spécifiques de calcul d'intérêt le sont.

Retenons cinq concepts fondateurs :

- objet : une représentation informatique d'une entité du domaine sur lequel porte le système (ex. un compte bancaire particulier) ou utile pour la solution informatique (une pile particulière). Un objet se caractérise par :
  - un état, ensemble d'informations descriptives, données, la partie *statique* de l'objet. Par exemple, un compte bancaire se décrit par un numéro, une référence vers le client qui le possède, un solde, etc.
  - un comportement, constitué d'un ensemble d'opérations qui décrivent les actions de l'objet, la partie *dynamique* de l'objet. Par exemple, un compte bancaire peut être crédité d'une certaine somme, l'opération consistera à ajouter la somme au solde du compte.
  - une identité, propre à chaque objet et permettant de le distinguer d'un autre sans ambiguïté.
- message : une unité de communication entre les objets. Dans sa version la plus simple, un message envoyé à un objet correspond à une opération possible sur cet objet et qui sera invoquée. Un message consistera par exemple à invoquer l'opération de crédit sur un certain compte bancaire
- classe : une classe est l'abstraction (regroupe) d'un ensemble d'objets ayant une structure et un comportement commun. Par exemple la classe `Compte bancaire` représentera les comptes bancaire du système. On dira souvent qu'une classe est un concept du domaine sur lequel porte le logiciel ou du domaine du logiciel (par exemple un type de données abstrait tel que la pile). Une classe peut se voir sous

deux aspects :

- un aspect *extensionnel* : l'ensemble des objets (ou instances) représentés par la classe
- un aspect *intensionnel* : la description commune à tous les objets de la classe, incluant les données (partie statique ou attributs) et les opérations (partie dynamique)

A la notion de classe et d'objet s'attache la notion d'encapsulation, puisque les attributs et les opérations sont regroupés dans l'objet ou dans l'objet et certaines de ces caractéristiques peuvent être mises en accès restreint afin de limiter le couplage dans un logiciel. Nous rappelons que la limitation du couplage augmente la facilité à maintenir le logiciel.

- spécialisation/généralisation et héritage : la spécialisation/généralisation est une organisation des classes considérant le point de vue extensionnel qui se base sur l'inclusion de leurs ensembles d'objets (par exemple la classe `Compte bancaire` vue comme un ensemble d'objets contient et généralise la classe `Compte bancaire rémunéré`). L'héritage adopte un point de vue intensionnel et permet le partage et la réutilisation des attributs et des opérations (par exemple la classe `Compte bancaire rémunéré` hérite de la classe `Compte bancaire` les caractéristiques qui y sont déclarées comme si elles avaient été décrites localement).
- polymorphisme : un nom d'objet (une variable) peut désigner des objets de classes différentes, un nom d'opération peut désigner différentes opérations . C'est un mécanisme qui permet notamment d'écrire des expressions valables pour des objets de différentes classes, y compris des classes ... qui ne sont pas encore écrites au moment ou l'expression est définie ! C'est ce qui rend les programmes à objets si facilement extensibles.

Dans le présent cours, ces concepts seront vus en détails avec l'interprétation qu'en donne le langage C++. Ils seront confrontés à deux autres aspects du langage avec lesquels ils s'intègrent, le mécanisme de gestion d'exceptions et la généricité paramétrique.

1.1.1 les langages à classes

L'origine des langages à objets remonte aux années 1970 et dans les deux décennies qui ont suivi de nombreux langages de programmation et de modélisation ont été conçus. Certains n'ont pas dépassé le stade de l'utilisation par des laboratoires de recherche, mais les langages actuels font une bonne synthèse, quoiqu'imparfaite sur certains aspects, du bouillonnement d'idées de cette époque. Nous ne parlons dans ce cours que des langages à classes cependant il faut savoir qu'une lignée entière de langages de programmation par objets se réclame de la *programmation par objets sans classes*, et nous renvoyons le lecteur intéressé vers la famille des langages à frames ou à prototypes tels que SELF pour de plus amples informations sur le sujet.

Il est de coutume de différencier deux grandes écoles dans les langages de programmation à classes.

- L'école scandinave

  mots d'ordre : typage essentiellement statique, destinés à la production d'applications lourdes
  - Langages « tout objet », $Trellis/Owl$, $Eiffel$, $Simula$
  - Langages mixtes : Ada9X, Java (possède aussi un interpréteur), surcouches de C, Pascal/DELPHI, Virtual Basic, Cobol objet

- L'inspiration Lisp

mots d'ordre : typage dynamique, destinés au prototypage rapide
- Langages « tout objet », *Smalltalk*, *Objective − C* (préprocesseur C)
- Langages mixtes, par exemple sur Lisp : *Clos*, *ObjvLisp*

Retenez qu'il s'agit de grandes catégories et que tout doit être modéré : Smalltalk est utilisé avec succès dans de grandes applications industrielles, en particulier dans le domaine de la banque.

1.1.2 Une brève histoire de C++

L'histoire raconte que Stroustrup, développant avec le langage Simula67 (un langage à objets fortement typé), trouva qu'il était très agréable à utiliser, mais fort peu efficace (en temps d'exécution). Il en conclut qu'il serait très intéressant de développer un langage possédant les deux qualités. Il choisit C comme langage de base, notamment parce que C est un langage efficace car de bas niveau, disponible à peu près partout, et facile à porter. Le monde de référence de C++ est donc le monde UNIX/linux. *C with classes* (1979) fut le résultat de la première tentative, qui évolua par la suite vers C++ (1983), nommé ainsi car ++ est l'opérateur d'incrémentation en C. C++ est un sur-ensemble de C, qui corrige certains défauts de C et lui ajoute la partie objet. Il connaît un grand succès, pas seulement dans le monde UNIX et reste la référence pour le développement de logiciels nécessitant une grande efficacité d'exécution. Il a été une première fois normalisé par un comité ANSI/ISO : ce processus s'est terminé en 1998 et la norme a subi une première remise à jour en 2003. Il vient de subir de nouvelles évolutions (C++ 2011), dont nous parlerons dans un document annexe à ce cours, avec l'introduction de nouvelles caractéristiques, incluant une nouvelle itération `foreach`, des fonctions anonymes et fermetures, des tuples, la gestion du temps, des tableaux de taille fixe dans la STL, l'initialisation des attributs avec une nouvelle syntaxe, une amélioration de la sûreté de typage, de la programmation générique et le `multi-threading`.

Chapitre 2

Classes et messages en C++ : le socle de la programmation par objets

Ce chapitre rappelle les principaux aspects de la programmation par objets en C++, à l'exclusion de l'héritage, et se concentre sur les notions de classes et d'envoi de messages. Il est destiné à des lecteurs ayant déjà une bonne connaissance des notions de base en C++ : types de base (incluant les tableaux et pointeurs), expressions, fonctions et programmes.

2.1 Vers et strophes

Pour illustrer le cours nous utiliserons une modélisation très simplifiée de quelques éléments empruntés au domaine de la littérature et plus précisément de la poésie. Les modèles seront écrits en UML qui est le langage de modélisation par objets le plus utilisé actuellement. Nous préciserons les éléments de notation UML utilisés au fil des exemples.

Nous nous limiterons ici à la représentation des vers et des strophes dans une perspective métrique (étude des régularités de rime et de rythme dans la poésie).

Un *vers* est essentiellement une suite de mots à laquelle on peut attacher une rime, la sonorité terminale (écrite sous une forme normalisée). Un vers peut être saisi et affiché (voir la Figure 2.1). Aux deux attributs `suiteMots` et `rime` sont associées deux opérations particulières, habituellement appelées des accesseurs car elles se spécialisent dans l'accès en lecture (ex. `getSuiteMots`) ou en écriture (ex. `setSuiteMots`).

Une *strophe* est une suite de vers. Dans le cas général (qui admet la poésie en vers libres) nous admettrons que l'on peut saisir une strophe par saisie du nombre de vers puis saisie successive des différents vers (dans l'ordre d'apparition dans la strophe). L'affichage d'une strophe est l'affichage de ses vers successifs.

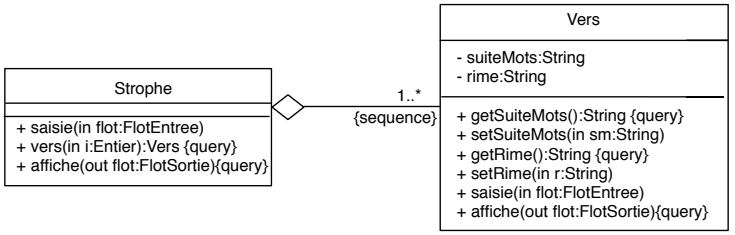


FIGURE 2.1 – Eléments de la représentation des strophes

La contrainte `query` qui complète la description des deux accesseurs en lecture de la

classe `Vers` précise que ces opérations, lorsqu'elles sont appliquées à un objet de la classe ne modifie pas celui-ci, ce sont des sortes de *requêtes*. Entre une strophe et un vers, une association de type agrégation (losange évidé) est utilisée pour traduire la sémantique *se compose de*. L'agrégation autorise le partage, c'est-à-dire qu'un vers peut apparaître dans plusieurs strophes. La multiplicité contre la classe `Vers` indique qu'une strophe se compose d'(au moins) un ou plusieurs vers. La contrainte `{sequence}` précise que les vers sont ordonnés et autorise un même vers à figurer plusieurs fois dans la même strophe. Le diagramme est du niveau conception (proche de la traduction dans un langage de programmation), donc on s'y attarde également sur le niveau de protection (visibilité) des attributs, qui sont privés (symbole `-`) et des opérations qui sont toutes publiques ici (symbole `+`). De manière générale nous respecterons cette règle (attributs privés, méthodes publiques) dont les bénéfices seront discutés dans le chapitre sur la protection.

2.2 Classes et objets en C++

2.2.1 Le principe de séparation de l'interface et de l'implémentation des classes

Dans un code C++ bien construit, une classe se présente en deux parties, ordinairement placées chacune dans un fichier auquel nous donnerons un nom en respectant une convention bien utile pour s'y retrouver lorsque l'on manipule de nombreuses classes à la fois.

L'interface Cette partie de la description de la classe est placée dans un fichier d'entête (fichier *header*, muni d'une extension `.h`) et comporte les attributs et les déclarations (signatures ou entêtes) des méthodes. Les fichiers relatifs aux interfaces des classes `Vers` et `Strophe` sont présentés respectivement dans les figures 2.2 et 2.3. Nous étudierons plus en détails plus loin les différents éléments syntaxiques. Les macros (exécutées par un préprocesseur qui manipule le texte du programme avant la compilation) `#ifndef`, `#define` et `#endif` évitent des erreurs de déclarations multiples lorsqu'il y a des inclusions croisées. Nous les utiliserons systématiquement pour nous affranchir de ce problème. Le texte du programme inclus entre `#ifndef vers_h` et `#endif` est ignoré lorsque la variable `vers_h` est définie, c'est-à-dire dès que le préprocesseur a analysé une première fois cette portion de texte qui contient justement la définition `#define vers_h`.

Examinons les déclarations des attributs pour quelques rappels. Une déclaration simple commence par le nom du type suivi du nom de l'attribut. Dans la déclaration de l'attribut `suiteVers` nous trouvons le caractère $*$ qui est la marque de déclaration des pointeurs. `Vers* pv` est une déclaration où `pv` est l'adresse d'un emplacement mémoire contenant une instance de vers, on peut dire aussi `*pv` est une instance de vers, en utilisant $*$ comme opérateur de déréférencement. Les types tableau et pointeur sont (plutôt malheureusement) assimilés en C++, et `Vers* pv` peut être compris également comme un tableau de vers d'une taille qui n'est pas encore connue et qui sera alloué dynamiquement [1]. Compliquons maintenant un peu les choses ... `Vers** pv` est ici un tableau (ou pointeur) de pointeurs vers des instances de vers. Ce tableau implémente l'agrégation spécifiée en UML qui nous indique qu'une strophe se compose de vers.

Attention, les attributs ne sont pas initialisés automatiquement en C++ ! Il faut penser à le faire, par exemple dans les constructeurs.

1. La notation utilisant des crochets nécessite de connaître la taille du tableau, et l'allocation est alors automatique : `Vers[5] tv;` est un tableau de 5 vers.

```
#ifndef vers_h
#define vers_h
class Vers
{
private:
  string suiteMots;     // suiteMots, attribut de type string
  string rime;               // rime, attribut de type string
public:
  Vers();
  Vers(string s);
  Vers(string s, string r);
  virtual ~Vers();
  virtual string getSuiteMots()const;
  virtual void setSuiteMots(string sm);
  virtual string getRime()const;
  virtual void setRime(string r);
  virtual void saisie(istream& is);
  virtual void affiche(ostream& os)const;
};
#endif
```

FIGURE 2.2 – `Vers.h`

```
#ifndef Strophe_h
#define Strophe_h
class Strophe
{
private:
  Vers ** suiteVers;  // suiteVers, attribut de type tableau de pointeurs vers des Vers
                           // implémente l'agrégation "une strophe se compose de vers"
  int nbVers;
public:
  Strophe();
  virtual ~Strophe();
  virtual void saisie(istream& is);
  virtual Vers* vers(int i)const;
  virtual void affiche(ostream& os)const;
};
#endif
```

FIGURE 2.3 – `Strophe.h`

L'implémentation Cette deuxième partie de la description de la classe se compose des corps des méthodes et de la définition de certains attributs (attributs de classes, `static`) dont nous reparlerons plus loin. Elle apparaît dans un fichier d'extension `.cc` ou `.cpp` suivant les environnements.

Les fichiers relatifs aux implémentations des classes `Vers` et `Strophe` sont présentés respectivement dans les figures 2.4 et 2.5. Ces fichiers d'implémentation commencent par un entête constitué des fichiers de déclaration qui sont inclus. Il s'agit soit d'éléments de la librairie standard (comme `iostream` pour les entrées/sorties ou `string` pour les chaînes de caractères), soit de fichiers créés par l'utilisateur, comme les fichiers header de nos deux classes. La syntaxe est différente dans les deux cas.

Notez également que chaque méthode déclarée dans l'interface réapparaît ici, avec un nom préfixé par le nom de la classe et l'opérateur de portée. Par exemple la méthode `getSuiteMot` apparaît sous le nom étendu `Vers::getSuiteMots` dans le fichier d'implémentation. Retenez que la classe est un bloc nommé, et que les méthodes, en dehors du cas de l'envoi d'un message à un objet, ont besoin d'être référencées comme un élément de ce bloc. Les autres aspects sont expliqués dans la suite.

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"

Vers::Vers(){}
Vers::Vers(string sm){suiteMots=sm;}
Vers::Vers(string sm, string r){suiteMots=sm;rime=r;}
Vers::~Vers(){}

string Vers::getSuiteMots()const
{return suiteMots;}
void Vers::setSuiteMots(string sm)
{suiteMots=sm;}
string Vers::getRime()const
{return rime;}
void Vers::setRime(string r)
{rime=r;}

void Vers::saisie(istream& is)
{cout <<"vers puis rime" <<endl;is>>suiteMots>>rime;}

void Vers::affiche(ostream& os)const
{os<<"<<"<<suiteMots<<">>";}
```

FIGURE 2.4 – `Vers.cc`

2.2.2 Encapsulation et protection

Le propre de la classe est d'encapsuler les attributs et les opérations et cette encapsulation s'accompagne d'une protection (également appelée visibilité ou contrôle d'accès). En C++, la directive de protection est un mot-clef qui s'applique à une suite de décla-

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"
#include"Strophe.h"

Strophe::Strophe(){suiteVers=NULL; nbVers=0;}
Strophe::~Strophe(){if (suiteVers) delete[] suiteVers;}
                // on ne détruit que le tableau, pas les vers.

Vers* Strophe::vers(int i)const
{if (i>=0 && i<nbVers) return suiteVers[i]; else return NULL;}

void Strophe::saisie(istream& is)
{
  if (suiteVers) delete[] suiteVers;
  cout << "Entrer le nombre de vers : " << endl;
  is>>nbVers; suiteVers = new Vers*[nbVers];
  for (int i=0; i<nbVers; i++)
    {
      Vers *v=new Vers(); v->saisie(is); suiteVers[i]=v;
    }
}
void Strophe::affiche(ostream& os)const
{
  for (int i=0; i<nbVers; i++)
    {
      suiteVers[i]->affiche(os); os << endl;
    }
}
```

FIGURE 2.5 – `Strophe.cc`

rations d'attributs ou de méthodes. Ainsi dans notre exemple, tous les attributs ont été déclarés `private` et toutes les méthodes `public`, ce qui est une stratégie habituelle de masquage des données internes à un objet. En effet les propriétés `private` d'une classe ne sont, en première approximation, accessibles que dans les méthodes de cette classe. Cette politique couramment admise permet de n'utiliser que les méthodes pour accéder à l'état d'un objet et minimise la connaissance que les autres objets en ont. En conséquence, le programme est plus facilement maintenable : une modification des données utilisées pour stocker l'état d'un objet n'a de conséquence que sur le code de certaines des méthodes de la classe, et pas au-delà. Le cas typique est celui des types de données abstraits. Le choix d'implémentation d'un type `Pile` n'est pas connu des parties de code qui utilisent la pile et n'en connaissent que les opérations et les spécifications (préconditions, postconditions et axiomes). Que la pile soit implémentée à un stade du développement par un tableau puis à un stade ultérieur par une liste chaînée n'a pas de conséquence sur les fonctionnalités, les services prévus restent rendus. Seules peuvent changer des propriétés que l'on qualifie de « non fonctionnelles » comme la complexité en temps de calcul.

Par défaut, c'est-à-dire si aucun mot-clef n'est précisé, les propriétés d'une classe sont privées. La politique est inverse pour les `struct` qui sont des classes pour lesquelles les propriétés sont publiques par défaut.

Le troisième mode de contrôle d'accès en C++ est `protected`, nous l'aborderons plus loin dans un chapitre consacré à ce sujet.

2.2.3 La création des objets

Créer des objets consiste, dans les langages à classes, à instancier les classes, et ceci se fait de trois manières en C++. Nous privilégierons la troisième forme qui permet la pleine utilisation du polymorphisme. C'est une particularité de C++ que de demander au programmeur d'être actif en ce qui concerne la gestion de la mémoire. La plupart des autres langages de programmation à objets (Java et Smalltalk) disposent d'un mécanisme de ramasse-miettes et déchargent le programmeur de cette préoccupation. Chaque point de vue présente ses intérêts : pour des logiciels standards, il est préférable que le programmeur ne se préoccupe pas de l'allocation mémoire, qui est un problème de bas niveau par rapport aux abstractions de la programmation par objets. Inversement, pour des logiciels sensibles (logiciels embarqués, temps réel, installés sur des systèmes procurant peu de ressources comme les téléphones mobiles ou les agendas électroniques, logiciels manipulant de grandes quantités de données comme le traitement d'images), pouvoir maîtriser la gestion de la mémoire peut être un atout.

L'allocation statique L'objet est créé lorsque le flot de contrôle atteint l'instruction de création et pour toute la durée du programme. Nous rencontrerons ce cas d'allocation avec les attributs `static`.

L'allocation automatique L'objet, qui est créé dans un bloc, a une durée de vie qui correspond à celle des objets locaux à ce bloc. Il existe dès que le flot de contrôle passe par l'instruction de création et détruit lorsque le flot de contrôle quitte le bloc. Le bloc peut être par exemple une déclaration de classe ou une méthode (variable locale ou paramètre).

L'allocation dynamique L'objet, qui sera référencé par un pointeur, est créé grâce à l'opérateur `new` et détruit par l'usage de l'opérateur `delete`.

```
{
. . .
Vers v; // v est une variable de type Vers
Strophe s; // s est une variable de type Strophe
. . .
}
```

FIGURE 2.6 – Exemple d'allocation automatique (dans la pile)

```
Vers *pv=new Vers(); // pv est une variable pointeur sur Vers
Strophe *ps = new Strophe(); //ps est une variable de type pointeur sur Strophe
 . . .
delete pv; // destruction de pv
delete ps; // destruction de ps
```

FIGURE 2.7 – Allocation dynamique (dans le tas)

Récapitulons Dans la classe `Vers`, les deux attributs sont des instances de `string` allouées automatiquement (leur emplacement sera réservé dans l'emplacement alloué au vers, elles apparaissent et disparaissent avec le vers). Pour autant cela n'empêche pas que des données propres à la chaîne soient réservées ailleurs ... mais nous n'en savons rien a priori. Dans la classe `Strophe`, un emplacement pour un pointeur et pour un entier sont alloués automatiquement (réservés dans l'emplacement alloué à une strophe). Par contre les vers et le tableau qui les référence doivent être alloués ailleurs. La Figure 2.8 donne un aperçu des structures en mémoire juste après les déclarations (et création par `new` le cas échéant) de `v, s, pv, ps`. Les valeurs placées dans les instances de strophe sont mises conformément aux instructions exécutées par les constructeurs. Nous ne faisons pas d'hypothèse sur la représentation des chaînes (vides) construites comme valeurs pour les attributs des instances de vers.

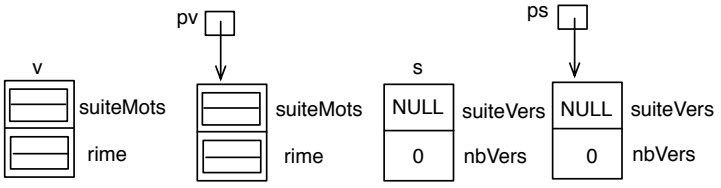


FIGURE 2.8 – Structure en mémoire après création des vers et strophes

2.2.4 Un (tout petit) peu d'introspection

L'introspection est la capacité d'un langage à inspecter, en cours d'exécution, les éléments d'un programme, par exemple pour connaître la classe exacte de l'objet stocké dans une variable ou encore la liste des attributs d'une classe. Le stade suivant est la réflexivité, grâce à laquelle les éléments du programme peuvent être véritablement manipulés, qui permet par exemple de créer une nouvelle classe pendant l'exécution d'un programme ou encore d'instancier une classe en donnant seulement son nom sous forme d'une chaîne de caractères. C++ est assez dépourvu de toutes ces caractéristiques avancées de la programmation par objets, mais il est tout de même utile de connaître le peu qui existe.

Cela se présente sous le nom de RTTI (pour *Run Time Type Information*). Un opérateur particulier, `typeid`, lorsqu'il est appliqué à un objet o, retourne une instance d'une classe `type_info` qui décrit le type (la classe) de o. La classe `type_info` ne contient malheureusement pas beaucoup de méthodes, principalement les opérateurs $!=$ et $==$ pour comparer les types de deux objets et la méthode `name` qui retourne une chaîne contenant le nom de la classe de l'objet. Dans l'exemple suivant, le nom de la classe `Vers` sera affiché.

```
Vers *pv1=new Vers();
cout << typeid(*pv1).name() << endl;
```

2.3 Méthodes et envoi de messages

En C++, méthodes et fonctions cohabitent, il faut donc tirer le meilleur parti de deux modes de programmation, objets et fonctionnelle. Le programme principal, en particulier est une fonction et non une méthode, pas même une méthode statique comme en Java.

Dans le cadre de la programmation par objets on parle souvent d'envoi de message plutôt que d'appel de méthode (ou d'application de méthode à un objet) pour bien mettre en évidence le fait que le mécanisme d'appel est dynamique (la méthode est choisie pendant l'exécution) plutôt que statique (cas de l'appel de fonction où la fonction qui sera exécutée est connue et liée avant l'exécution). C'est pour cela que vous verrez toujours dans ce cours les méthodes précédées du mot-clef `virtual` qui assure que l'on utilise le mécanisme de liaison dynamique dont nous reparlerons longuement dans les chapitres sur l'héritage.

2.3.1 Passage de paramètres et valeur retournée

Il y a une seule manière de passer les paramètres d'une fonction ou d'une méthode en C++ qui respecte simplement la sémantique de l'initialisation : lorsque la fonction ou la méthode est appelée, un emplacement est réservé pour chacun des paramètres formels (ceux de la signature) et chaque paramètre formel est initialisé avec le paramètre réel (ceux de l'expression d'appel) correspondant.

Les effets sur l'environnement peuvent cependant être notablement différents suivant le type des paramètres formels. Dans le programme qui suit, nous avons trois exemples typiques de passage de paramètres :

- passage *par valeur* lors de l'appel `f(i, pi, j)`, la valeur de `i` est copiée dans `fi`, puis `fi` est incrémenté, ce qui est sans effet sur `i`
- passage *par adresse* lors de ce même appel, la valeur de `pi` (qui est l'adresse d'une zone de type entier) est copiée dans `fpi`, puis la valeur pointée par `fpi` est incrémentée, et cette valeur est toujours pointée par `pi` donc apparaît bien modifiée lorsqu'on termine `f`
- passage *par référence* après l'initialisation d'une référence telle que `fri` par une variable telle que `j`, il faut comprendre que `fri` est un alias pour `j`, quand on incrémente `fri` on incrémente donc `j` puisque c'est la même entité avec deux noms différents.

```
void f(int fi, int* fpi, int &fri)
{
fi++; // incremente fi, copie locale du premier parametre reel
(*fpi)++; // incremente le contenu de la zone pointee par le deuxième paramètre réel
fri++; // incremente le troisième parametre reel
}
```

```
main()
{
int i = 4;
int *pi = new int;
*pi = 4;
int j=4;
cout << "i=" << i << "  *pi=" << *pi << "  j=" << j << endl; // i=4  *pi=4  j=4
f(i, pi, j);
cout << "i=" << i << "  *pi=" << *pi << "  j=" << j << endl; // i=4  *pi=5  j=5

}
```

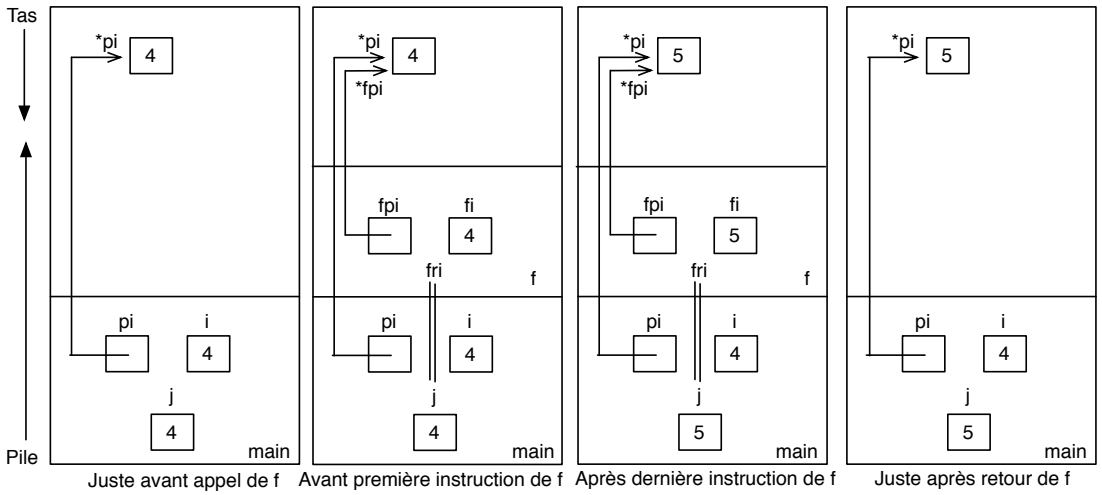


FIGURE 2.9 – Evolution de la mémoire lors des passages de paramètres et retour de f

Le retour de valeur suit la même sémantique : une valeur non nommée du type retourné est initialisée avec la valeur retournée.

Les signatures que nous rencontrerons lors de ce cours porteront souvent sur des passages par référence mais précédés du mot-clef `const` pour signifier que l'objet est passé sans copie, mais cependant on ne désire pas qu'il soit modifié pendant l'exécution de l'opération. Qu'en est-il de l'objet auquel on a envoyé un message ? Il peut aussi être déclaré constant pour la durée de l'exécution de la méthode pourvu que la signature se termine par le mot-clef `const` placé derrière la parenthèse fermante (voir par exemple la méthode `getSuiteMots()`). C'est ainsi que se traduira la contrainte `query` d'UML.

2.3.2 Moi (l'objet courant)

Pour désigner l'objet auquel on a envoyé un message pendant l'exécution de la méthode correspondante, on utilise la pseudo-variable `this`. En C++ `this` est un pointeur constant sur l'objet. Il est sous-entendu pour l'accès aux propriétés (attributs et méthodes) de l'objet. Par exemple, la méthode `getSuiteMots()` peut se réécrire comme suit :

```
string Vers::getSuiteMots()const
{return this->suiteMots;}    // equivaut a return (*this).suiteMots;
```

Pendant l'exécution de `getSuiteMots()`, `this` est de type statique `Vers* const` (pointeur constant vers un vers).

2.3.3 Constructeurs et destructeur

Les constructeurs et destructeurs sont des méthodes spéciales appelées automatiquement lors de la création et de la destruction des objets, il suffit de suivre les règles énoncées dans la partie 2.2.3 pour comprendre à quel moment ils sont appelés.

Constructeurs Les constructeurs portent le même nom que la classe et ne sont jamais déclarés `virtual`. Leur rôle consiste à initialiser les attributs (on ne peut le faire au point de leur déclaration) et à acquérir des ressources dont l'objet a besoin (allocation mémoire, ouverture de fichiers, etc.). On écrit autant de constructeurs qu'il peut exister de cas de création d'objets à condition d'avoir des listes de types de paramètres différentes. Lorsqu'aucun constructeur n'est déclaré, le compilateur en crée un par défaut, vide et sans paramètres. Lorsqu'un objet est copié (par exemple lors d'un passage de paramètres par valeur), un constructeur particulier, le constructeur par copie, est appelé. S'il n'a pas été écrit par le programmeur il est synthétisé automatiquement par le compilateur et effectue une copie champ par champ de l'objet (avec la sémantique de la copie liée au type du champ). Lorsque la sémantique du programme veut que lors de la copie les deux objets ne partagent pas la même ressource mémoire dynamique, on écrit un constructeur qui effectue une copie dite profonde des objets. Pour la classe `Strophe` on se trouve dans une telle situation, on peut décider par exemple que les deux strophes ne partagent pas le même tableau de vers, mais les vers restent partagés, ce qui correspond à la sémantique de l'agrégation en UML.

Dans le fichier header on ajoute la déclaration `Strophe(const Strophe& autreStrophe);`. Dans le fichier d'implémentation, on écrit la méthode.

```
Strophe::Strophe(const Strophe& autreStrophe)
{
nbVers=autreStrophe.nbVers;
suiteVers=new Vers*[nbVers];
for (int i=0; i<nbVers; i++)
suiteVers[i]=autreStrophe.suiteVers[i];
}
```

Par défaut le constructeur synthétisé aurait eu la forme suivante.

```
Strophe::Strophe(const Strophe& autreStrophe)
{
suiteVers=autreStrophe.nbVers;
      // *this et autreStrophe partagent alors le meme tableau
nbVers=autreStrophe.nbVers;
}
```

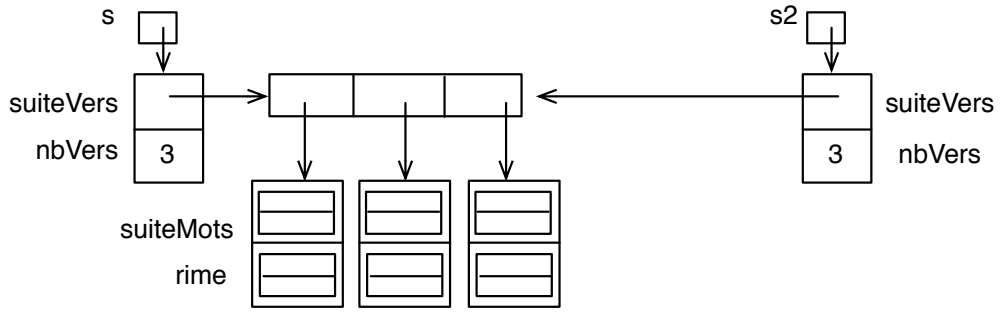
Un petit programme simple permet de voir l'utilisation de ce constructeur (Figure 2.10).

```
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"
```

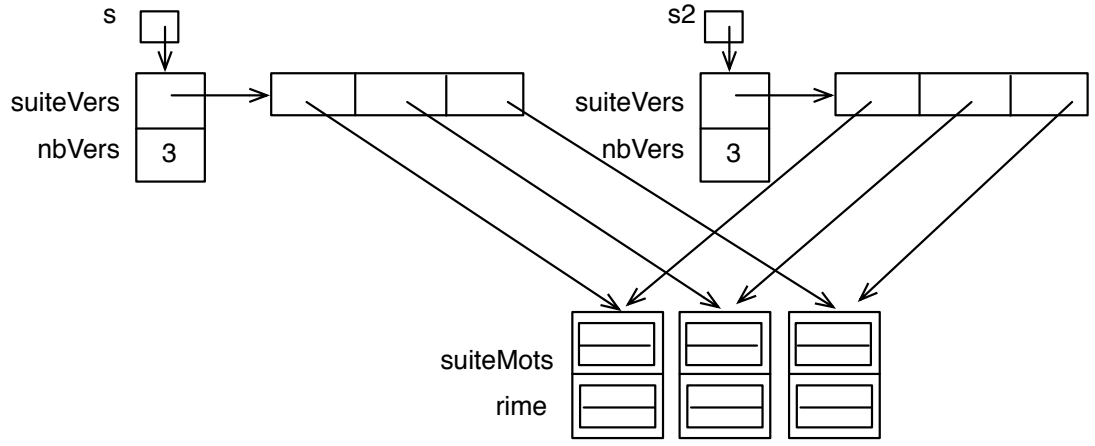
```
#include"Strophe.h"

Strophe *s = new Strophe();
s->saisie(cin);
s->affiche(cout);

Strophe *s2 = new Strophe(*s);
s2->saisie(cin);
s2->affiche(cout);
s->affiche(cout);
```



Avec une copie champ par champ (par défaut)



Avec une copie plus profonde (constructeur par copie défini dans le cours)

FIGURE 2.10 – Effet d'une copie avec la copie par défaut ou avec le constructeur effectuant une copie plus profonde

Destructeur Le destructeur (unique) porte également le même nom que la classe derrière le préfixe composé du caractère ~. Son rôle est de restituer les ressources acquises par l'objet lors de sa construction ou plus tard, notamment de restaurer l'espace mémoire occupé. Dans le cas des strophes, on ne détruit que le tableau, pas les vers conformément à la sémantique de l'agrégation en UML, et de manière cohérente avec le fait que deux strophes peuvent

partager les mêmes vers.

2.3.4 Accesseurs

Vous aurez compris que nous attachons beaucoup d'importance dans ce cours à la protection des attributs. Ils seront en règle générale privés et munis lorsque c'est approprié (il est utile de les manipuler depuis l'extérieur de la classe) de méthodes d'accès auxquelles nous avons donné un nom particulier, issu de la convention proposée par Java.

On n'écrit pas de méthode d'accès pour tous les attributs ! Par exemple les attributs codant la structure interne d'une pile ne doivent en aucun cas apparaître donc on ne fera pas d'accesseurs publics (nous discuterons plus tard d'autres types de contrôles d'accès). Le nombre de vers d'une strophe n'est pas accessible directement en écriture : on doit utiliser les autres méthodes pour ne pas créer d'incohérence.

Lorsqu'on écrit une méthode d'accès, on en profite pour se poser diverses questions sur les contraintes que doivent vérifier les attributs. L'accesseur en écriture est tout indiqué pour servir de point de contrôle de cette valeur. Il peut ainsi n'affecter de valeur à l'attribut que si la valeur transmise en paramètre est cohérente, sinon, rien n'est modifié.

Chapitre 3

Spécialisation/généralisation et héritage simple

3.1 La notion de spécialisation/généralisation et d'héritage

L'approche objet a pour but de rapprocher les objets du monde réel de leur représentation informatique. Elle se fonde pour cela sur :

— la notion de « concept » dans la pensée scientifique actuelle [1], c'est-à-dire une association entre une *extension* (l'ensemble des objets couverts par le concept) et une *intension* (l'ensemble des prédicats vérifiés par les objets couverts) ; les concepts sont représentés dans les langages à objets par les classes et les interfaces (en Java) ;

Exemple 1 *Le concept de « rectangle » a pour extension l'ensemble des rectangles et pour intension un certain nombre de propriétés, dont le fait de posséder quatre côtés parallèles deux à deux, deux côtés consécutifs formant un angle droit, etc. Dans un langage de programmation, on ne représentera pas forcément toutes ces contraintes, mais on en déduira par exemple la possibilité de décrire un rectangle par une hauteur et une largeur, un calcul spécifique du périmètre ou de l'aire, etc.*

— la notion de *classification par spécialisation* qui consiste à organiser les classes d'après l'inclusion de leurs extensions, dont on déduit souvent l'inclusion inverse ou le raffinement des intensions. Cette classification est représentée dans les langages de programmation par l'héritage, terme qui met l'emphase sur le fait qu'une classe transmet ses caractéristiques à ses sous-classes.

Exemple 2 *Le concept de « carré » spécialise le concept de « rectangle » : ceci se fonde sur le fait que l'ensemble des carrés est en effet inclus dans l'ensemble des rectangles (inclusion des extensions). En sens inverse les propriétés du concept « rectangle » s'appliquent au concept « carré » et se spécialisent (raffinement des intensions) : la largeur et la hauteur sont égales par exemple, le calcul du périmètre et de l'aire peuvent être spécialisés grâce à cette propriété, etc.*

Ces classifications sont parfois des arborescences (héritage simple) comme en Smalltalk, en Java entre les classes, mais peuvent être plus généralement des graphes sans circuit (héritage multiple), comme en C++, Eiffel, en Java entre les interfaces. Dans ce cours, nous traitons l'héritage en C++, sans aborder les problèmes spécifiques dûs à l'héritage multiple.

1. D'après J. Ladrière, « Concept » Encyclopœdia Universalis, 1993

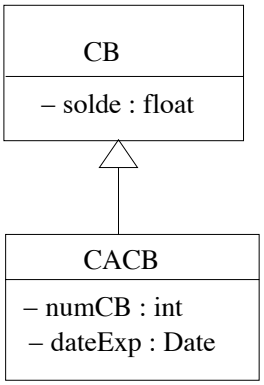


FIGURE 3.1 – Comptes bancaires simplifiés

L'héritage est également le support d'une caractéristique propre aux langages à objets connue sous le nom de « polymorphisme d'héritage », qui consiste à redéfinir (spécialiser) une méthode d'une classe dans ses sous-classes (la méthode prend « plusieurs formes »). Dans les langages tels que C++, on parlera parfois de *variables polymorphes* pour désigner des variables qui peuvent repérer des instances de classes différentes dans une même hiérarchie. L'envoi de message au travers d'une telle variable aura pour résultat d'invoquer l'une des formes de la méthode, forme choisie d'après la classe de l'instance repérée.

3.2 Réalisation en C++

3.2.1 Déclaration

Nous étudions le cas de deux classes représentant respectivement des comptes bancaires (`CB`) et des comptes bancaires avec carte bleue (`CACB`). Pour exprimer en C++ la relation de spécialisation présentée sur le schéma UML de la figure 3.1, on précise l'entête de la classe `CACB` par `class CACB : public virtual CB`. Le mot-clef `public` est une expression de contrôle d'accès posée sur la relation d'héritage. Il est conseillé de l'utiliser systématiquement lorsque la relation d'héritage en C++ traduit une relation de spécialisation en UML. Dans ce cours nous ne sortirons pas de ce cadre : l'héritage ne sera pas utilisé pour traduire d'autres relations entre classes. Le mot-clef `virtual` est utilisé dans l'éventualité d'une spécialisation de ces classes par une hiérarchie d'héritage multiple, pour que cette spécialisation se déroule de manière correcte. Il est conseillé de le préciser car lorsque l'on développe une classe, on ne maîtrise pas ce qu'elle deviendra par la suite. Les deux classes sont représentées schématiquement ci-dessous.

```
// Compte bancaire
class CB
{private:
  float solde;
 public:
  CB();
  virtual ~CB();
};

// Compte bancaire avec carte bleue
class CACB : public virtual  CB
{private:
```

```
  int numCB; Date dateExp;
 public:
  CACB();
  virtual ~CACB();
};
```

3.2.2 Construction/Destruction

Ordre d'appel Lorsqu'un objet est créé, par exemple par l'expression `CACB *pc = new CACB();` ou encore par l'expression `CACB c;`, les constructeurs sont appelés en respectant l'ordre suivant :

- les constructeurs des classes sont appelés du haut vers le bas de la hiérarchie d'héritage (depuis les super-classes vers les sous-classes),
- les constructeurs sont appelés pour les attributs dont le type est une classe (pas un pointeur sur une classe), et ceci juste avant le constructeur de la classe qui déclare l'attribut.

Ainsi, dans notre exemple, les constructeurs sont appelés dans l'ordre suivant :

`CB()`
`Date()` ... pour l'attribut `dateExp`
`CACB()`

Les destructeurs sont appelés en sens inverse :

`~CACB()`
`~Date()` ... pour l'attribut `dateExp`
`~CB()`

Passage de paramètres aux constructeurs Pour transmettre des paramètres à l'un des constructeurs de la super-classe, on n'opère pas comme en Java avec un appel à `super`. Supposons par exemple que la classe `CB` dispose d'un constructeur qui permette d'initialiser le solde à la création du compte.

```
//...... CB.h .......          //...... CB.cc

class CB
{
private:
  float solde;                  CB::CB(float s){solde=s;}
public:
  CB(float s);
};
```

L'écriture d'un constructeur dans la classe `CACB` qui permette d'initialiser les attributs va se présenter comme suit. Il faut noter que le passage de paramètres ne se fait pas à l'intérieur du corps du constructeur, mais s'écrit dans son entête.

```
//...... CACB.h .......              //...... CACB.cc
class CACB : public virtual  CB
{
private:
  int numCB;
  Date dateExp;
```

```
public:                                  CACB::CACB(float s, int n, Date d):CB(s)
  CACB(float s, int n, Date d);            {numCB=n; dateExp=d;}
};
```

Initialisation des attributs Pendant que nous étudions les constructeurs, faisons un petit détour par une particularité de C++ en ce qui concerne l'initialisation des attributs. La technique la plus connue consiste à la faire à l'intérieur des corps des constructeurs. C'est le cas dans les exemples précédents, et une expression telle que `dateExp=d;` a la sémantique de l'affectation (on appelle l'opérateur d'affectation connu sur la classe `Date` qui effectue, sauf redéfinition expresse par l'utilisateur, une affectation champ par champ).

Une variante d'écriture consiste à écrire, à la même place que l'expression de transmission des paramètres au constructeur de la super-classe, une expression indiquant quelle valeur on veut donner à un attribut :
`CACB::CACB(float s, int n, Date d):CB(s),numCB(n),dateExp(d){}` . Dans ce cas, la sémantique est celle de la copie : on appelle le constructeur par copie de la classe `Date` pour l'attribut `dateExp`.

Dans la majorité des cas, l'affectation et la copie ont été définies de la même manière sur la classe, le résultat est donc le même, mais il peut se trouver des cas où le comportement sera différent, par exemple parce que l'on a redéfini le constructeur par copie mais pas l'opérateur d'affectation, ou parce qu'on les a définis avec des sémantiques différentes.

3.2.3 Méthodes

Liaison C++, contrairement à Java, propose deux modes de liaison des méthodes, une liaison statique et une liaison dynamique. La liaison statique fixe la méthode à appeler à la compilation, la liaison dynamique la fixe à l'exécution. La liaison statique est malheureusement la situation par défaut. Lorsque l'on souhaite que la liaison soit dynamique, on doit faire précéder les déclarations de méthodes par le mot-clef `virtual`.

Nous montrons au travers de l'exemple ci-dessous la différence entre les deux modes de liaison. Commençons par la liaison dynamique. Nous ajoutons une méthode, appelée par exemple une fois l'an, et qui débite les frais de gestion des comptes. Pour un compte ordinaire, le montant de ces frais est placé dans la variable de classe `fraisGestion`. Pour un compte avec carte bleue, on ajoute à ces frais le montant d'une autre variable de classe, en l'occurrence `fraisCB`.

```
//...... CB.h .......                //...... CB.cc
class CB
{
private:
  float solde;
  static float fraisGestion;             float CB::fraisGestion=10;
public:
  ...
  virtual void changeAvec(float f);        void CB::changeAvec(float f){solde+=f;}
  virtual float getSolde()const;         float CB::getSolde()const{return solde;}
  static float getFraisGestion();        float CB::getFraisGestion()
                                         {return fraisGestion;}
  virtual void debitFrais();             void CB::debitFrais()
 };                                      {changeAvec(-fraisGestion);}


//...... CACB.h .......               //...... CACB.cc
```

```
class CACB : public virtual  CB
{
private:
  ...
  static float fraisCB;                 float CACB::fraisCB=5;
public:                                 void CACB::debitFrais()
  ...                                   {changeAvec(-(getFraisGestion()+fraisCB));}
  virtual void debitFrais();
};
```

Supposons que dans la suite des événements on crée un tableau `dossierComptes` représentant par exemple l'ensemble des comptes de la banque. Ce tableau, pour lequel le type statique des éléments est `CB*`, est garni de divers comptes, dont certains sont ordinaires, et d'autres avec carte bleue. `dossierComptes[i]` est une variable polymorphe.

```
CB **dossierComptes = new CB*[2];
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(74,1212,d);

for (int i=0; i<2; i++)
  dossierComptes[i]->debitFrais();
```

L'effet de la liaison dynamique est le suivant : les méthodes appelées sont recherchées en commençant par la classe à laquelle appartient l'instance.

`dossierComptes[0]->debitFrais();` ⟶ appel de `debitFrais` de CB
`dossierComptes[1]->debitFrais();` ⟶ appel de `debitFrais` de CACB

Si le mot-clef `virtual` avait été omis dans la déclaration des méthodes, la liaison se serait effectuée statiquement, donc en considérant le type de la variable `dossierComptes`. Le résultat aurait l'effet désastreux (uniquement pour le banquier) d'appeler dans les deux cas la méthode `debitFrais` de `CB`, donc une méthode inadaptée pour `dossierComptes[1]`.

`dossierComptes[0]->debitFrais();` ⟶ appel de `debitFrais` de CB
`dossierComptes[1]->debitFrais();` ⟶ appel de `debitFrais` de CB!!!

Pas de liaison dynamique dans les constructeurs ! Même en utilisant le mot-clef `virtual`, C++ déroge au principe de la liaison dynamique dans le cas de l'appel de méthodes dans les constructeurs. L'explication fournie est que pendant l'exécution du constructeur, l'objet n'est pas encore tout à fait construit et ne serait pas en mesure d'exécuter correctement un comportement qui nécessite qu'il soit intégralement initialisé. Elle peut s'admettre mais c'est souvent assez fâcheux et d'ailleurs Java n'a pas fait ce choix.

Examinons une autre version du constructeur de la classe `CB` appelant la méthode de débit des frais :

```
CB::CB(float s){solde=s; debitFrais();}
```

L'absence de liaison dynamique aura pour effet que dans les deux créations suivantes, la méthode `debitFrais` de CB sera toujours celle qui sera appelée.

```
dossierComptes[0] = new CB(200);
dossierComptes[1] = new CACB(74,1212,d);
```

Le conseil que l'on peut donner est donc d'éviter d'appeler des méthodes redéfinies dans un constructeur.

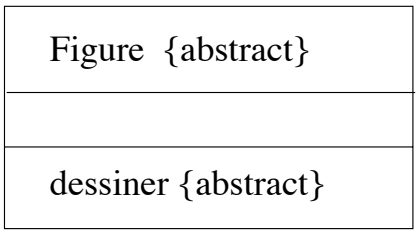


FIGURE 3.2 – Une classe et une méthode abstraite

Appel de la méthode héritée Un autre point qui va différer de Java concerne la désignation de la méthode héritée. En Java cette désignation se faisait grâce au mot-clef `super`, C++ ne dispose pas de ce mot-clef et se sert à la place d'un mécanisme de désignation explicite qui se base sur l'opérateur de portée `::`.

Par exemple, si on veut écrire la méthode `debitFrais` de la classe `CACB` en appelant celle de `CB`, cela prendra la forme présentée ci-dessous.

```
void CACB::debitFrais()
{
  CB::debitFrais();
  changeAvec(-fraisCB);
}
```

Cette désignation explicite a quelques inconvénients, car on peut appeler une méthode située beaucoup plus haut dans la hiérarchie de classes, qui peut ne pas être la version la plus spécialisée. On risque ainsi de ne pas respecter la spécialisation telle qu'elle a été pensée dans le programme.

Redéfinition de méthode En C++ une méthode en redéfinit une autre si elle a même nom, même liste de types d'arguments et en général même type de retour. D'après la documentation, lorsque le type de retour est pointeur ou référence vers une classe, il peut alors être spécialisé. Par exemple, en théorie une méthode de type `virtual CB* debitFrais()` dans `CB` peut être redéfinie par une méthode de type `virtual CACB* debitFrais()`. Ce n'est pas implémenté sur tous les compilateurs.

3.2.4 Classes et méthodes abstraites

Il n'y a pas en C++ de syntaxe particulière pour préciser qu'une classe est abstraite, c'est-à-dire qu'elle n'aura pas d'instances propres (elle peut en avoir par l'intermédiaire de ses sous-classes). Par contre une méthode abstraite est signalée dans le code par la syntaxe ci-dessous. Il n'y aura pas d'implémentation de cette méthode dans le fichier `.cc`. En dialecte C++, la méthode est dite « virtuelle pure ». La traduction du diagramme UML de la figure 3.2 sera donc :

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

Tout comme en Java, on ne peut plus ensuite créer d'instance propre de cette classe. Ainsi on peut déclarer une variable de type pointeur ou référence de figure, comme `Figure *f` mais l'instruction `Figure *f=new Figure();` échouera.

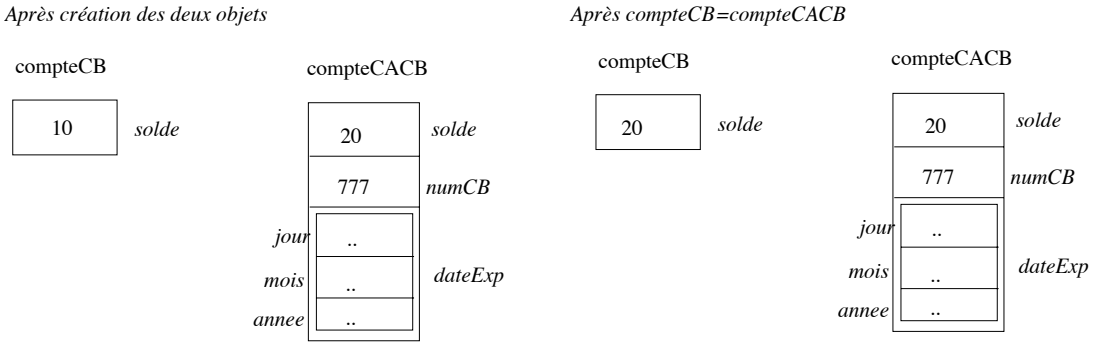


FIGURE 3.3 – Affectation entre objets (1)

3.2.5 Coercition/Affectation

Il faut bien différencier en C++ l'affectation entre variables dont le type est une classe qui appelle l'opérateur `operator=` de la classe et l'affectation entre pointeurs ou références vers des objets (la seule possible en Java).

Affectations entre variables dont le type est une classe Ce type d'affectation appelle l'opérateur `operator=` de la classe. Cet opérateur peut avoir été redéfini dans la classe, sinon par défaut il effectue une copie champ par champ. Dans le cas général, l'affectation d'un compte avec carte bleue à un compte bancaire ordinaire est acceptée, et les champs communs, ici le solde, sont copiés (sémantique par défaut de l'affectation (voir figure 3.3). Par contre l'affectation inverse est interdite.

```
CB compteCB(10); CACB compteCACB(20,777,d);
compteCB = compteCACB;
// INTERDIT PAR DEFAUT ..... compteCACB = compteCB;
```

On peut bien sûr rendre possible cette affectation inverse en écrivant un opérateur d'affectation adapté.

```
class CACB : virtual public CB
{
 public:
  virtual CACB& operator=(const CB&);
};
........
CACB& CACB::operator=(const CB& compte)
{
  if (this != &compte)
     {changeAvec(compte.getSolde()); numCB=0; dateExp=Date();}
  return *this;
}
```

L'affectation devient possible et, réalisée juste après la création des deux objets ci-dessus, a le résultat présenté sur la figure 3.4.

Affectation entre pointeurs L'affectation d'un pointeur de compte bancaire avec un pointeur de compte bancaire avec carte bleue est autorisée mais pas l'inverse. Certains auteurs parlent d'affectation polymorphe pour désigner cette opération.

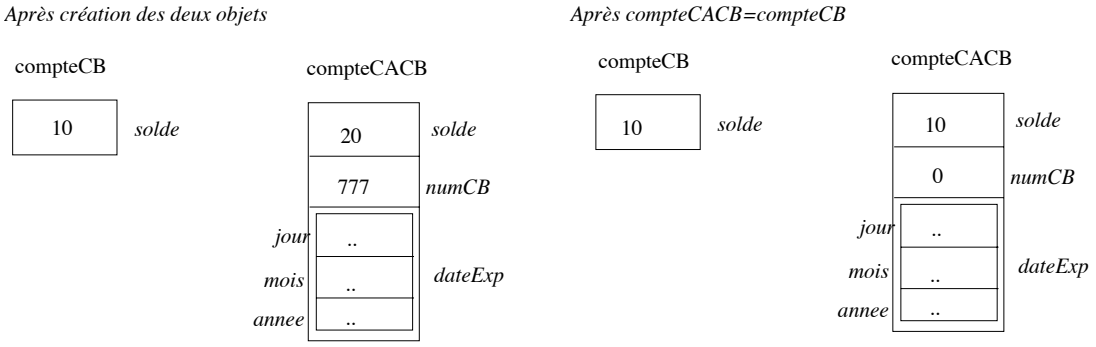


FIGURE 3.4 – Affectation entre objets (2)

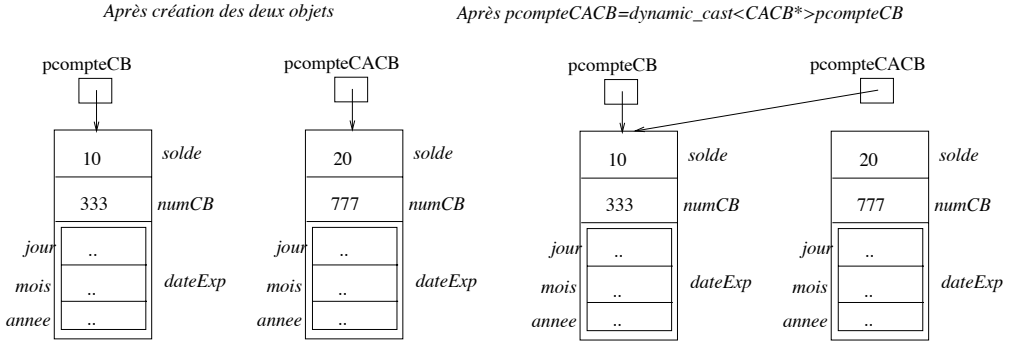


FIGURE 3.5 – Affectation entre pointeurs vers des objets

```
CB *pcompteCB = new CB(10); CACB *pcompteCACB = new CACB(20,777,d);
pcompteCB = pcompteCACB;
//INTERDIT ....  pcompteCACB = pcompteCB;
```

Lorsque cela prend un sens car le pointeur vers `CB` repère en réalité une instance de `CACB`, on peut comme en Java réaliser une coercition (un *type cast*) grâce à la syntaxe suivante (il est déconseillé dans le cadre de la programmation par objet en C++ d'utiliser toute autre syntaxe). L'opérateur `dynamic_cast` effectue une vérification de type à l'exécution et retourne `NULL` si la coercition est incorrecte (le pointeur n'est pas du type attendu).

```
CB *pcompteCB = new CACB(10,333,d);
CACB *pcompteCACB = new CACB(20,777,d);
pcompteCACB = dynamic_cast<CACB*>(pcompteCB);
```

La figure 3.5 montre l'effet de ces dernières instructions.

Chapitre 4

Initialisation, affectation et conversion

C++ entreprend de manière implicite un grand nombre d'actions. On l'a vu déjà avec tout ce qui concerne la création, la copie ou la destruction des objets. Les initialisations, les affectations et la conversion en sont un autre exemple que nous allons détailler dans ce cours. D'un côté cela peut accélérer la programmation, de l'autre cela peut avoir le désavantage de faire perdre au programmeur la maîtrise de la situation.

Ce chapitre approfondit ces différents points en utilisant la classe `Date` décrite ci-dessous.

```
//------------------ Date.h ------------------------------
#ifndef Date_h
#define Date_h
class Date
{
private:
int annee, mois, jour;
public:
Date();
Date(int,int,int);
virtual ~Date();
virtual int getAnnee()const;
virtual int getMois()const;
virtual int getJour()const;
virtual void setAnnee(int);
virtual void setMois(int);
virtual void setJour(int);
virtual void affiche(ostream&)const;
virtual bool operator<(Date d)const; // parametre volontairement passe par copie
};
#endif

//------------------ Date.cc -------------------------------
#include <iostream>
#include <string>
using namespace std;
```

```
#include "Date.h"

Date::Date(){annee=2000; mois=12; jour=31;}
Date::Date(int a,int m,int j){annee=a; mois=m; jour=j;}
Date::~Date(){}
int Date::getAnnee()const{return annee;}
int Date::getMois()const{return mois;}
int Date::getJour()const{return jour;}
void Date::setAnnee(int a){annee=a;}
void Date::setMois(int m){mois=m;}
void Date::setJour(int j){jour=j;}
void Date::affiche(ostream&)const
{cout << jour << "/" << mois << "/" << annee << endl;}
bool Date::operator<(Date d)const
{
return
( (this->annee < d.annee) ||
  (this->annee == d.annee && this->mois < d.mois) ||
  (this->annee == d.annee && this->mois == d.mois && this->jour < d.jour) );
}

//------------------ mainDate.cc ------------------------------
#include <iostream>
#include <string>
using namespace std;
#include "Date.h"

int main()
{
Date *d1=new Date(2004,12,31);
d1->affiche(cout);
Date *d2=new Date(2004,10,31);
d2->affiche(cout);
bool inf = *d2 < *d1;
cout << inf << endl;
}
```

4.1 Initialisation versus affectation

Le symbole = devrait faire référence de manière constante à l'opérateur d'affectation. Il n'en est rien ! Aussi étrangement que cela puisse paraître, dans la deuxième ligne ci-dessous, c'est le constructeur par copie qui est appelé et non l'opérateur = (que vous ayez ou non écrit un opérateur = associé à la classe).

```
Date d1;
Date d2=d1;
```

Ces deux lignes sont donc équivalentes à : `Date d1, d2(d1);`
et non pas à : `Date d1, d2; d2=d1;`

4.2 Les mystères de la conversion élucidés

4.2.1 Conversion standard d'un type (classe) vers un super-type (super-classe)

Ce sujet a été abordé dans le cours sur l'héritage, mais il est bien de le réexaminer sous l'angle de la conversion, cette fois avec les types de la figure 4.1.

```
int main()
{
DateAnniversaire *da = new DateAnniversaire();
DateAnniversaire *dagh = new DateAnniversaireGrandHomme();
DateAnniversaireGrandHomme *dagh2 = new DateAnniversaireGrandHomme();

//conversions standards sous-type vers super-type
da=dagh2; // oui : affectation polymorphe
//dagh2=dagh; non ! il faut effectuer une coercition explicite
dagh2=dynamic_cast<DateAnniversaireGrandHomme *>(dagh); // coercition explicite

//affectation
*da=*dagh2; //ok mais l'objet est tronque : on copie la partie DateAnniversaire
// *dagh2=*da; non sauf si on definit un operateur = dans DateAnniversaireGrandHomme
// avec comme parametre un DateAnniversaire
}
```

4.2.2 Conversion basée sur un constructeur

Les constructeurs qui ne prennent qu'un paramètre doivent attirer toute notre attention dans les affaires de conversion : en effet ils sont considérés comme des opérateurs de conversion.

Prenons par exemple l'ajout d'un constructeur en apparence très anodin à la classe `Date`, qui accepte une chaîne de caractères comme paramètre et en extrait l'année (4 premiers caractères), le mois (les deux caractères suivants) et le jour (les deux derniers caractères).

```
//----- Dans Date.h --------
class Date{...
Date(string s);
...
};

//----- Dans Date.cc --------
Date::Date(string s) // s de format 'yyyymmdd'
{
  annee=1000*(s.at(0)-'0')+100*(s.at(1)-'0')+10*(s.at(2)-'0')+(s.at(3)-'0');
  mois=10*(s.at(4)-'0')+(s.at(5)-'0');
  jour=10*(s.at(6)-'0')+(s.at(7)-'0');
}
```

Dans un premier temps, voyons l'usage normal (explicite) de ce constructeur dans le programme principal `mainDate` que nous complétons.

```
//utilisation explicite du constructeur comme opérateur de conversion
Date stSylvestre("20041231"); // création d'un objet, allocation automatique
stSylvestre.affiche(cout);
cout << (*d2 < Date("20041231")) << endl; // création d'un objet volatile
```

De manière moins évidente, dans certaines situations où l'on attend une date et où une chaîne de caractères est disponible, le compilateur utilise le constructeur pour transformer la chaîne en date et autoriser l'opération.

```
//utilisation implicite du constructeur comme opérateur de conversion
string s="20041231";
cout << (*d2 < s) << endl;
```

Les concepteurs de C++, dans leur grande sagesse, ont cependant prévu que l'on pouvait rendre impossible cette utilisation implicite du constructeur comme opérateur de conversion : il suffit pour cela d'ajouter le mot-clef `explicit` devant tout constructeur dont on veut interdire l'usage pour la conversion implicite.

```
class Date
{
.....
public:
....
explicit  Date(string s);
....
};
```

4.2.3 Opérateurs de conversion

Le procédé que nous avons décrit précédemment n'est pas toujours applicable :

— lorsque l'on veut convertir vers un type de base (qui n'est pas une classe et ne peut avoir de constructeur),

— lorsque l'on veut convertir vers une classe qui est déjà définie et que l'on ne veut pas modifier.

Dans le cas de la classe `Date`, maintenant que nous avons pris goût aux conversions des chaînes de caractères en dates, nous pouvons désirer faire l'opération inverse, c'est-à-dire convertir des dates en chaînes de caractères.

Pour cela, nous introduisons un opérateur de conversion dans la classe `Date`.

```
//----- Dans Date.h --------
class Date{...
public:
....
//opérateur de conversion de Date vers string
virtual operator string()const;
...
};

//----- Dans Date.cc --------
Date::operator string()const
{
```

```
string s="";
int reste;
int chiffreMilleAnnee=annee/1000;
reste=annee%1000;
int chiffreCentAnnee=reste/100;
reste=reste%100;
int chiffreDizAnnee=reste/10;
reste=reste%10;
int chiffreUnitAnnee=reste;
s+= (char)(chiffreMilleAnnee+((int)'0'));
s+=(char)(chiffreCentAnnee+((int)'0'));
s+=(char)(chiffreDizAnnee+((int)'0'));
s+=(char)(chiffreUnitAnnee+((int)'0'));
int chiffreDizMois=mois/10;
reste=mois%10;
int chiffreUnitMois=reste;
s+=(char)(chiffreDizMois+((int)'0'));
s+=(char)(chiffreUnitMois+((int)'0'));
int chiffreDizJour=jour/10;
reste=jour%10;
int chiffreUnitJour=reste;
s+=(char)(chiffreDizJour+((int)'0'));
s+=(char)(chiffreUnitJour+((int)'0'));
return s;
}
```

Etudions-le en action, tout d'abord dans un contexte où on l'appelle de manière explicite, puis de manière implicite dans un nouveau complément de `mainDate`, après ajout préalable de la fonction `f`.

```
// ajout d'une nouvelle fonction

void f(string s){cout << "f" << s << endl;}

//------------- complément
int main()
{
..........
// utilisation explicite de l'opérateur de conversion Date vers string

cout << (string(*d2)) << endl;

// utilisation implicite de l'opérateur de conversion Date vers string

string s2; s2=stSylvestre; // avec l'opérateur d'affectation de la classe string

f(stSylvestre); //avec l'appel de f qui attend une string
```

4.2.4 Ambiguïtés

Si les exemples précédents vous ont plutôt convaincus du bien-fondé de cette facilité à définir des conversions, examinez à présent l'exemple suivant. On y crée deux classes munies chacune d'un constructeur prenant en paramètre une date (donc potentiellement un opérateur de conversion des dates vers les objets de la classe concernée). Puis on écrit deux fonctions de même nom, prenant respectivement comme paramètre une personne, une voiture ou une date. A première vue ... pas de problème puisque le principe de la surcharge statique veut que l'on puisse définir plusieurs fonctions ou méthodes de même nom à condition que les listes de types de paramètres diffèrent dans les signatures, ce qui est le cas ici.

```
class  Personne
{
private:
Date anneeNaissance;
public:
Personne(Date d){anneeNaissance=d;}
};

class Voiture
{
private:
Date anneeConstruction;
public :
Voiture(Date d){anneeConstruction=d;}
};

void ecritAnnee(Personne p){cout << "ecrit annee personne"
<< p.getAnneeNaissance() << endl;}

void ecritAnnee(Voiture p){cout << "ecrit annee voiture"
<< p. getAnneeConstruction() << endl;}
```

Puis voyons ce qui se passe dans un programme où l'on appelle `ecritAnnee` avec une date.

```
ecritAnnee(stSylvestre);
```

Le compilateur peut convertir s'il n'existe que l'une des deux fonctions, mais lorsque les deux coexistent, il a autant d'efforts (de conversion) à faire pour l'un des deux appels que pour l'autre. Résultat, il s'arrête et nous fait part de sa perplexité par un message d'erreur signalant une ambiguité.

Pour désambiguiser, une seule solution : écrire une fonction dont la signature s'apparie exactement avec la forme d'appel.

```
void ecritAnnee(Date d){cout << "ecrit annee Date" << endl;}
```

4.2.5 Régle et moralité

Au-delà des cas d'ambiguïté, qui sont faciles à détecter puisqu'il se produit des erreurs de compilation, il faut savoir la priorité avec laquelle le compilateur effectue ses choix :

1. s'il y a un appariement exact, il est utilisé. Des conversions très simples peuvent être effectuées : tableau vers pointeur, nom de fonction vers pointeur de fonction ou type `T` vers `const T`.
2. conversion vers un type englobant (`bool` vers `int`, `short` vers `int`, `float` vers `double`, etc.)
3. conversion standard : `int` vers `double` ou l'inverse, `T*` vers `void*`, pointeur vers une classe se transforme en pointeur vers une super-classe (affectation polymorphe)
4. utiliser les conversions définies par l'utilisateur (constructeurs et opérateurs de conversion)
5. utiliser l'ellipse (trois points `...` qui servent à dire que la liste des paramètres n'est pas connue) dans une déclaration de fonction

Il y a ambiguïté lorsque deux conversions de même niveau de priorité se produisent. La morale de l'histoire est qu'il faut être extrêmement prudent dans l'utilisation des conversions définies par l'utilisateur (on ne peut rien faire pour les autres...) et penser à des problèmes de cet ordre lorsqu'un programme ne compile pas ou n'effectue pas ce qui est attendu.

4.3 Surcharge versus redéfinition

Pendant que nous examinons les règles avec lesquelles le compilateur choisit une méthode ou une fonction, il est bien de regarder en détails un exemple où surcharge statique (choix à la compilation) et redéfinition (choix à l'exécution) cohabitent, assez mal d'ailleurs dans le dernier cas.

Le programme suivant (écrit de manière très simplifiée, exceptionnellement sans séparer les fichiers header des fichiers d'implémentation) met en oeuvre les schémas présentés dans la figure 4.1. Nous décrivons tout d'abord le code des classes utilisées dans cet exemple.

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

class ObjetCelebration
{
};

class Statue :  virtual public ObjetCelebration
{
};

class Plat
{
};

class Restaurant
{
};

class Date
```

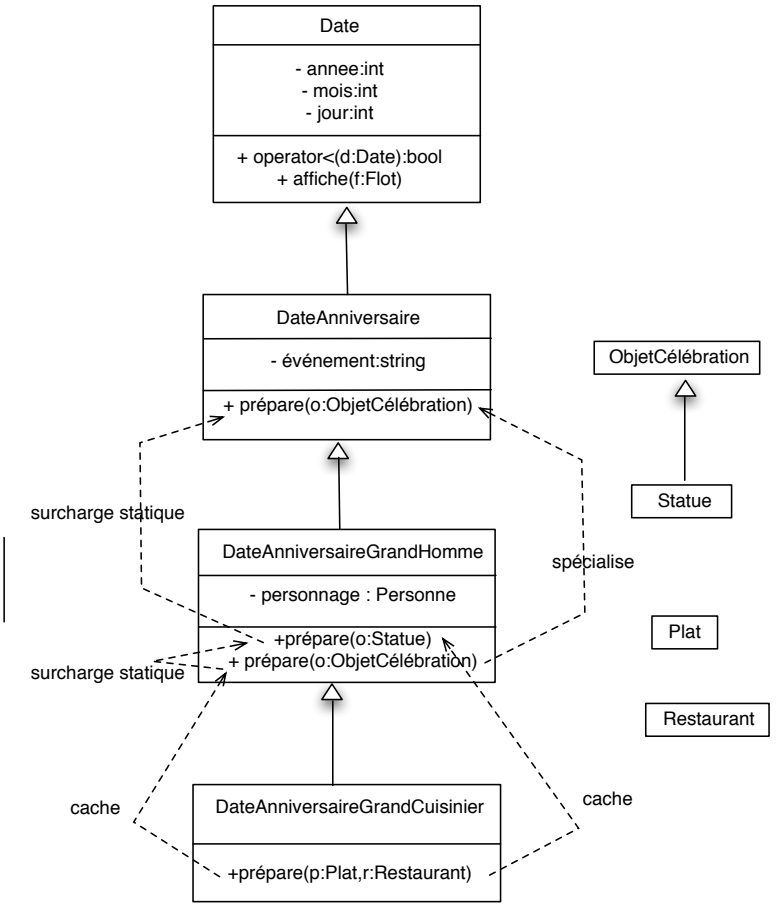


FIGURE 4.1 – Schéma de surcharge, définition et masquage

```
{
public:
virtual ~Date(){}
};

class DateAnniversaire : virtual public Date
{
public:
virtual void prepare(ObjetCelebration& o){cout << "da-prepare-objet" << endl;}
};

class DateAnniversaireGrandHomme : virtual public DateAnniversaire
{
public:
virtual void prepare(Statue&){cout << "dagh-prepare-statue" << endl;}
virtual void prepare(ObjetCelebration&){cout << "dagh-prepare-objet" << endl;}
};

class DateAnniversaireGrandCuisinier : virtual public DateAnniversaireGrandHomme
{
public:
virtual void prepare(Plat&,Restaurant&){cout << "dagh-prepare-plat-restau" << endl;}
// on les ajoute si on veut continuer a en heriter
//virtual void prepare(ObjetCelebration&){cout << "dagh-prepare-objet" << endl;}
//virtual void prepare(Statue&){cout << "dagh-prepare-statue" << endl;}
};
```

Dans les extraits de programme suivant on peut voir quelques exemples des effets de la redéfinition et de la surcharge.

Le compilateur examine le type statique (le type de la variable) pour choisir une signature compatible dans la classe. A l'exécution, le type de l'objet (déterminé par le `new`) est utilisé pour choisir la méthode ayant cette signature la plus spécialisée pour l'objet.

```
Restaurant r; Plat p; ObjetCelebration coupe; Statue s;
//----- appel de la méthode racine - choix de prepare(ObjetCelebration)
DateAnniversaire *da = new DateAnniversaire();
da->prepare(coupe); // da-prepare-objet

DateAnniversaire *dagh = new DateAnniversaireGrandHomme();
//----- appel de la méthode spécialisée - choix de prepare(ObjetCelebration)
dagh->prepare(coupe); // dagh-prepare-objet
//----- appel de la méthode spécialisée - choix de prepare(ObjetCelebration)
// car prepare(Statue) est une signature qui n'existe pas dans DateAnniversaire
dagh->prepare(s); // dagh-prepare-objet

DateAnniversaireGrandHomme *dagh2 = new DateAnniversaireGrandHomme();
//----- appel de la méthode spécialisée - choix de prepare(ObjetCelebration)
dagh2->prepare(coupe); // dagh-prepare-objet
//----- appel de la méthode surchargée- choix de prepare(Statue)
dagh2->prepare(s); // dagh-prepare-statue
```

Il y a cependant une petite anicroche dans ce joli mécanisme : la méthode `prepare(Plat, Restaurant)` devrait simplement surcharger, c'est-à-dire cohabiter avec, les autres méthodes (héritées) `prepare`. Ce serait le cas en Java. En C++, cette méthode `prepare(Plat , Restaurant)` cache en réalité et interdit l'accès à ces méthodes héritées. C'est fort malheureux !

```
DateAnniversaireGrandCuisinier *dagc = new DateAnniversaireGrandCuisinier();
//----- appel de la méthode surchargée
dagc->prepare(p,r); // dagh-prepare-plat-restau
//essai d'appel des méthodes héritées, cachées si elles ne sont pas redéfinies
//dagc->prepare(coupe);
//dagc->prepare(s);
}
```

Chapitre 5

Attributs et méthodes de classes

Les attributs et les méthodes dites « de classe » expriment, selon les cas, des données et des opérations :

— relatives à la classe toute entière et non à une instance particulière,

— partagées par toutes les instances de la classe (ce qui comprend les instances des sous-classes).

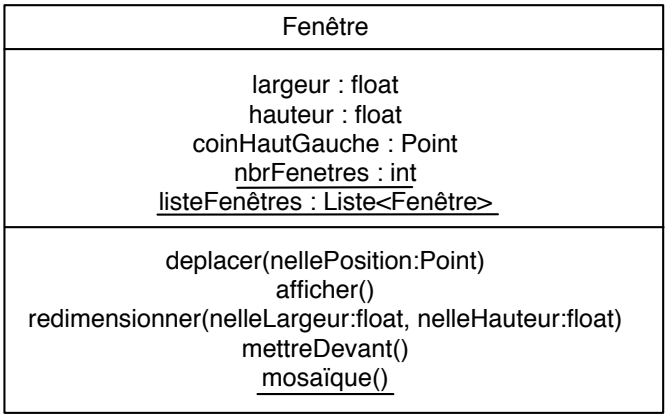


FIGURE 5.1 – La classe `Fenetre` en UML

Elles s'apparentent pour les unes aux variables globales, et pour les autres, aux fonctions, puisqu'elles ne concernent pas une instance donnée.

Quelques exemples d'attributs de classe.

classe *Carré*
variable de classe *NombreCôtés*, constante

classe *Français*
variable de classe *PrésidentDeLaRépublique*

classe *Date*
variable de classe *Mois*, contient la collection des noms des mois de l'année

Quelques exemples de méthodes de classes

— Affichage des variables de classe

— Création d'une instance prototypique de la classe

— Calculs sur l'ensemble des instances (statistiques ...)
plus généralement gestion de l'ensemble des instances
... Mais on peut aussi écrire une classe "gestionnaire", c'est même conseillé !

En C++, attributs et méthodes de classe sont précédés par le mot-clef `static` et en UML, ils apparaissent soulignés (voir figure 5.1).

Une classe Fenêtre

attributs d'instance	Largeur, Hauteur : CoinHautGauche :	flottant Point
attributs de classe	NbrFenêtres : ListeFenêtres :	entier Liste<Fenêtre>
méthodes d'instance	Deplacer(...) Afficher(...) Redimensionner(...) MettreDevant(...)	
méthodes de classe	Mosaïque()	

```
//------------- Fenetre.h ----------------------
class Fenetre
{
private :
// attributs d'instance
float Largeur, Hauteur;
Point CoinHautGauche;

// Declaration des attributs de classe
static vector<Fenetre> ListeFenetre;
public :
static int NbrFenetres;  // exceptionnellement public pour montrer l'utilisation

// Methodes d'instance
virtual void Deplacer(Point&);
virtual void Afficher(...);
virtual void Redimensionner(...);
virtual void MettreDevant(...);

// Methodes de classe
static void Mosaique();

// Miscellaneous
Fenetre();
virtual ~Fenetre();
};
```

Le fichier d'implémentation (d'extension .cc) doit contenir les définitions des attributs de classe, même si on ne les initialise pas.

```
//------------- Fenetre.cc ----------------------
```

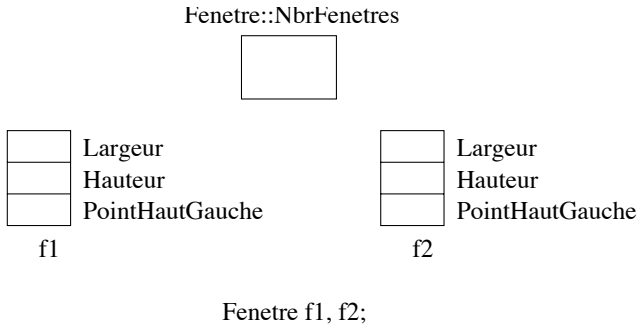


FIGURE 5.2 – Fenêtres graphiques

```
// Definition et initialisation eventuelle
int Fenetre::NbrFenetres=0;
vector<Fenetre> Fenetre::ListeFenetre;

void Fenetre::Mosaique() {}
```

En dehors de la portée de la classe (par exemple dans un `main`), on utilise ces variables et ces méthodes, lorsqu'elles sont accessibles, en les préfixant par le nom de la classe suivi de l'opérateur de portée (`::`).

```
//-------------Main Fenetre.cc -----------------------

int main()
{
  Fenetre::Mosaique();
  cout << Fenetre::NbrFenetres;
}
```

Comme l'illustre la figure 5.2 on trouve un attribut de classe pour toutes les instances et autant d'attributs d'instance que d'instances.

L'exemple choisi montre bien l'aspect souvent inabouti d'une conception contenant des attributs et des méthodes de classe : il serait judicieux d'introduire une classe gestionnaire de fenêtre qui recevrait toutes les caractéristiques statiques de la classe fenêtre.

Notez que ni les attributs de classe ni les méthodes de classe ne sont à proprement parler hérités. Les méthodes de classe ne sont donc pas `virtual` (pas de liaison dynamique possible) et `this` n'a pas de sens à l'intérieur.

Chapitre 6

Généricité paramétrique

6.1 Aspects conceptuels

DÃ‰FINITION 1 (GÉNÉRICITÉ PARAMÉTRIQUE) *La généricité paramétrique peut se définir comme la possibilité d'énoncer des descriptions de classes ou de fonctions dans lesquelles certains éléments restent formels (ce sont des paramètres).*

On parle alors de « modèle de classe » plutôt que de classe, et de « modèle de fonction » plutôt que de fonction. En dialecte C++, on utilise les termes de « template » de classe ou de fonction. Les paramètres peuvent être des types, des classes, des fonctions ou des valeurs.

EXEMPLE 3 (MODÈLE DE PILE) *Pour éviter de multiplier les définitions de* `Pile` *adaptées chacune à un type spécifique d'éléments stockés, il est ainsi utile de définir un* `modèle de pile`*, en faisant apparaître le type T des éléments stockés comme un paramètre formel.*

EXEMPLE 4 (MODÈLE DE FONCTION DE RECHERCHE) *Pour éviter de multiplier les définitions de* `fonction de recherche` *d'un élément dans un tableau, il est intéressant de définir un* `modèle de fonction de recherche` *d'un élément d'un type formel T dans un tableau d'éléments de type T.*

EXEMPLE 5 (MODÈLE D'ASSOCIATION) *Le concept d'association, souvent présent dans les bibliothèques proposant des structures de données (des collections), représente un couple d'éléments dont le premier est une clef d'accès au second. Là encore, il est judicieux de déclarer un modèle d'association, paramétré par le type de la clef et le type de la valeur.*

On parlera d'*instanciation* ou de *spécialisation* pour le procédé qui consiste à lier les paramètres formels à des paramètres réels, par exemple en liant T à `int` pour obtenir une pile d'entiers.

La figure 6.1 présente en UML, dans la partie haute deux modèles de classe (noter les paramètres dans des boîtes en pointillé), et dans la partie basse trois instanciations. Nous montrons sur le modèle de `Pile` les deux notations possibles pour l'instanciation : l'écriture entre chevrons du paramètre réel (pour la pile d'entiers) ou l'utilisation d'un lien de dépendance stéréotypé entre la classe (pile de réels) et le modèle de classe. Dans le dernier cas, le lien de dépendance porte le paramètre réel (`float`).

La généricité paramétrique existe en Eiffel et en C++, et à présent en Java 5 (tiger). Précédemment, en Java, on définissait des structures de données telles que la pile (`stack`) où le type des éléments stockés est le type le plus général (`Object`). Cette solution n'est

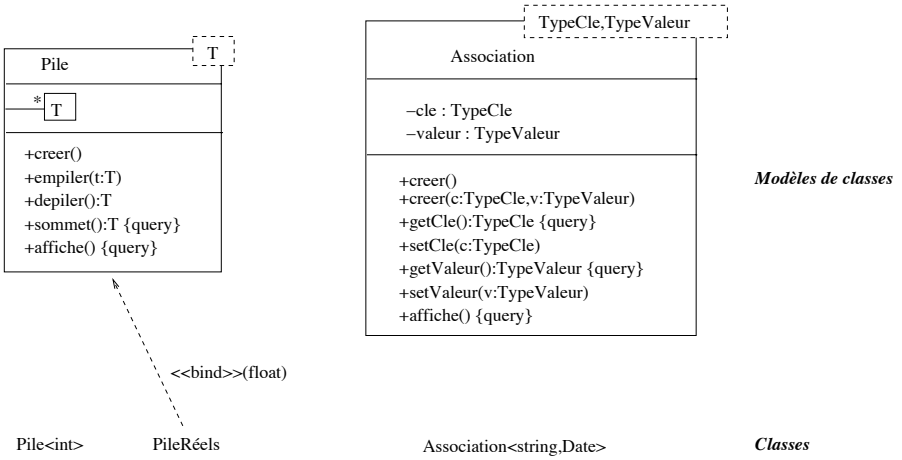


FIGURE 6.1 – Modèles de classes et leurs instanciations notés en UML

pas tout à fait satisfaisante, car elle impose souvent d'utiliser de nombreuses coercitions de type (`cast`).

En C++, la bibliothèque des collections de données (*Standard Template Library*) est implémentée à l'aide de modèles de classes.

6.2 Modèles de fonctions

Nous introduisons tout d'abord la syntaxe utilisée par C++ pour écrire des modèles de fonctions. Nous imaginons que nous souhaitions écrire une fonction qui permette d'afficher sur un flot de sortie standard les éléments d'un tableau entre deux accolades. Cette fonction s'écrirait, dans le cas des tableaux d'entiers, de la manière suivante.

```
void affiche(ostream& os, int t[], int taillet)
{
  os << "{ ";
  for (int i=0; i<taillet; i++) os << t[i] << " ";
  os << "}" << endl;
}
```

Pour l'adapter à un tableau contenant des éléments d'un autre type `TypeElt`, il suffirait de changer, dans la signature de la fonction, `int t[]` par `TypeElt t[]`.

Déclaration C++ va nous permettre de définir cette fonction de manière générale pour tout type `TypeElt` grâce à la syntaxe présentée ci-dessous.

```
template<typename TypeElt>
void afficheT(ostream& os, TypeElt t[], int taillet)
{
  os << "{ ";
  for (int i=0; i<taillet; i++) os << t[i] << " ";
  os << "}" << endl;
}
```

Dans l'entête ajoutée à la fonction, le mot-clef `template` signale que nous sommes en train de définir un modèle, le mot-clef `typename` introduit un paramètre formel de type, ici `TypeElt`. Ce paramètre devra donc être instancié par un type de C++ (classe, type primitif, tableau, pointeur, etc.).

Instanciation L'utilisation de ce modèle de fonction peut se faire en faisant apparaître de manière explicite le paramètre réel de type.

```
int tab[]={2,4,6,8};
afficheT<int>(cout,tab,4);

string *tab2=new string[2]; tab2[0]="lundi"; tab2[1]="mardi";
afficheT<string>(cout,tab2,2);
```

Une autre solution consiste à laisser le compilateur inférer le type réel à partir de la forme d'appel de la fonction. On peut ainsi réécrire les deux appels précédents sans mentionner explicitement l'instanciation effectuée.

```
afficheT(cout,tab,4);
afficheT(cout,tab2,2);
```

Cette deuxième solution ne fonctionne que dans les cas où le paramètre de type peut se déduire de la forme d'appel. Définissons un modèle de fonction `saisit` avec le type des éléments et la taille du tableau comme paramètres du modèle tandis que seul le tableau reste paramètre de la fonction.

```
template<typename TypeElt, int taillet>
void saisit(TypeElt t[])
{
  for (int i=0; i<taillet; i++) cin >> t[i];
}
```

La forme d'appel `saisit(tab);` est insuffisante pour déduire le deuxième paramètre du modèle, et on devra donc utiliser l'appel explicite `saisit<int,4>(tab);`

Expression de contraintes Contrairement à d'autres langages, notamment UML, Eiffel, Pizza et GJ, C++ ne permet pas d'exprimer de contraintes sur les paramètres de modèles.

Dans les modèles de fonction précédents, les opérateurs d'insertion `«(ostream&,const TypeElt&)` et d'extraction `»(istream&,TypeElt&)` doivent exister pour que les instanciations soient compilables, mais il n'y a pas de moyen d'écrire cette contrainte dans le code source.

6.3 Modèles de classes

Déclarations et définitions Nous étudions la déclaration du modèle de classe `Association` (ou en raccourci `Assoc`) présenté dans la figure 6.1. La projection en C++ de ce modèle de classe est introduite, comme dans le cas des fonctions, par une directive `template` suivie des paramètres du modèle qui sont ici le type de la clef et le type de la valeur. Le modèle de classe peut être accompagné de modèles de fonction, comme ici par un modèle d'opérateur surchargeant `operator«` pour l'insertion dans un flot. Le fichier *header* prend ainsi la forme suivante.

```
//.............................. Assoc.h ........................
#ifndef  Assoc_h
#define   Assoc_h

template<typename TypeCle, typename TypeValeur>
```

```
class Assoc{
private:
  TypeCle cle; TypeValeur valeur;
public:
  Assoc();
  Assoc(TypeCle, TypeValeur);
  virtual ~Assoc ();
  virtual TypeCle getCle()const;
  virtual void setCle(TypeCle);
  virtual TypeValeur getValeur()const;
  virtual void setValeur(TypeValeur);
  virtual void affiche(ostream&)const;
};

template<typename TypeCle, typename TypeValeur>
ostream& operator<<(ostream&, const Assoc<TypeCle,TypeValeur>&);
#endif
```

Dans la définition de la classe, chaque méthode se trouve également introduite par la directive `template<typename TypeCle, typename TypeValeur>`, et on peut remarquer qu'à part dans les noms des constructeurs, le terme `Assoc` n'apparaît plus jamais seul, il est toujours suivi des noms des paramètres.

```
//................... Assoc.cc ..................................
#include "Assoc.h"

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::Assoc()   {}

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::Assoc(TypeCle c, TypeValeur v)
:cle(c), valeur(v)    {}

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::~Assoc ()   {}

template<typename TypeCle, typename TypeValeur>
TypeCle Assoc<TypeCle,TypeValeur>::getCle()const   {return cle;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::setCle(TypeCle c)   {cle=c;}

template<typename TypeCle, typename TypeValeur>
TypeValeur Assoc<TypeCle,TypeValeur>::getValeur()const   {return valeur;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::setValeur(TypeValeur v)   {valeur=v;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::affiche(ostream &os) const
  {os <<getCle() << ", " <<getValeur();}

template<typename TypeCle, typename TypeValeur>
ostream& operator<<(ostream& os, const Assoc<TypeCle,TypeValeur>& a)
  {a.affiche(os); return os;}
```

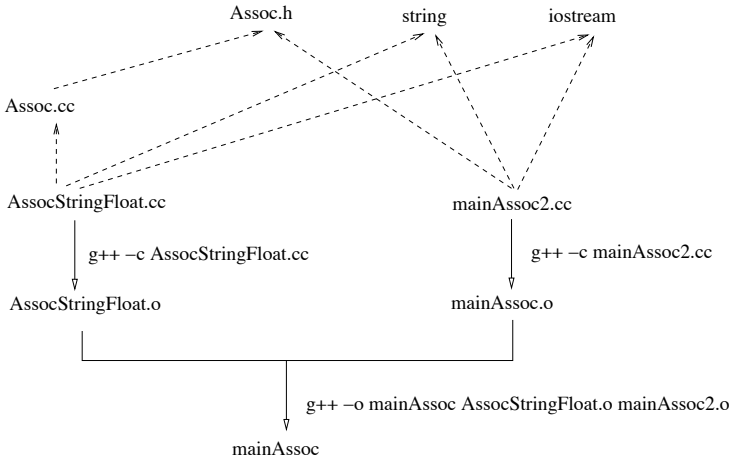


FIGURE 6.2 – Un exemple de compilation et d'édition de liens avec instanciation séparée

Instanciation L'instanciation d'un modèle de classe donne naissance à une classe. Cette instanciation peut se faire dans différents contextes. Le plus évident est lorsque l'on désire créer un objet d'une classe résultant d'une instanciation (l'exemple montre également une instanciation de l'opérateur `operator«`). Notez qu'un même programme peut contenir plusieurs instanciations (avec des paramètres différents) d'un même modèle.

```
// .................. mainAssoc1.cc .......................
#include <iostream>
#include <string>
#include "Assoc.h"

int main()
{
  Assoc<char, int*> a;
  Assoc<string,string>* pass= new Assoc<string,string>("citron","jaune");
  cout << *pass;
}
```

Un nouveau nom de type peut être utilisé pour réduire la taille des expressions.

```
typedef Assoc<string,Date> CarnetAnniversaires;
```

Il est conseillé de prévoir une directive d'instanciation explicite qui permet de vérifier l'instanciation indépendamment de tout programme utilisateur. Ce procédé permet également d'optimiser la compilation. Le mieux est de créer un fichier dédié à cette instanciation.

```
//.............. AssocStringFloat.cc .................
#include <iostream>
#include <string>
#include "Assoc.cc"
template class Assoc<string, float>;
template ostream& operator<<(ostream&, const Assoc<string, float>&);
```

La figure 6.2 propose une mise en œuvre possible de la construction d'un programme avec instanciation du modèle de classe dans un fichier séparé. Le programme principal utilisé (`mainAssoc2.cc`) diffère quelque peu du précédent (il ne peut contenir que des instanciations de `Assoc<string, float>`). Il existe d'autres manières d'inclure les fichiers entre eux (par exemple `Assoc.h` peut contenir `#include<iostream>`) mais le résultat physique de l'inclusion doit suivre le même schéma.

```
// .................. mainAssoc2.cc .......................
#include <iostream>
#include <string>
#include "Assoc.h"

int main()
{ Assoc<string, float> a;
  Assoc<string,float>* pass= new Assoc<string,float>("citron",12);
  cout << *pass;
}
```

Paramétrage par des fonctions et des constantes Nous avons détaillé un exemple dans lequel deux types sont passés comme paramètres du modèle de classe. Des fonctions ou des constantes peuvent également paramétrer un modèle. Nous donnons trois exemples illustratifs.

Exemple 6 (Passage de constante : borne d'une pile) *Nous montrons comment créer et instancier un modèle de classe représentant les piles bornées, paramétré par le type des éléments et par la taille maximale de la pile. Dans la déclaration, le mot-clef* `const` *est optionnel (mais sous-entendu lorsqu'il est absent).*

```
template<typename T, const int capacite>
class PileBornee
{ .....................};

int main()
{...PileBornee<int,10> p;...}
```

Exemple 7 (Passage de constante : valeur par défaut) *Pour écrire le constructeur sans paramètres du modèle de classe* `Assoc`*, il peut être intéressant de déterminer des valeurs « par défaut » des deux types* `TypeCle` *et* `TypeValeur` *afin d'initialiser la clef et la valeur. Une manière de connaître ces valeurs est de les passer comme paramètres du modèle de classe* `Assoc`*.*

```
// ............... AssocVD.h .....................................
template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
class Assoc{
private:
  TypeCle cle; TypeValeur valeur;
public:
  Assoc();
  Assoc(TypeCle, TypeValeur);
  virtual ~Assoc ();
  virtual TypeCle getCle()const;
  virtual void setCle(TypeCle);
  virtual TypeValeur getValeur()const;
  virtual void setValeur(TypeValeur);
  virtual void affiche(ostream&)const;
};

template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
ostream& operator<<(ostream&, const Assoc<TypeCle,TypeValeur,vdc,vdv>&);
```

Nous pouvons alors réécrire ainsi le constructeur de manière à ce qu'il initialise les attributs d'une association.

```
// .......... Dans AssocVD.cc ......................
template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
Assoc<TypeCle,TypeValeur,vdc,vdv>::Assoc()   {cle=vdc; valeur=vdv;}
```

Puis nous pouvons instancier, sur le compilateur actuel il est nécessaire d'introduire une spécialisation de l'opérateur d'insertion. Attention : C++ n'admet pas de chaîne littérale comme paramètre, seulement un pointeur sur un objet externe, ce qui explique les circonvolutions présentées ci-dessous.

```
#include<iostream>
#include<string>
#include"externS.cc"       // contient ...   string s="truc";
#include"AssocVD.cc"

extern string s;const int i=0;

//spécialisation de l'opérateur d'insertion dans un flot
ostream& operator<<(ostream& os, const Assoc<string*,int,&s,0>& a)
 {a.affiche(os); return os;}

int main()
 {Assoc<string*, int, &s, i> a; cout << a;}
```

Exemple 8 (Passage de fonction) *Dans cet exemple, nous déclarons et nous définissons partiellement un modèle de classe destiné à représenter les ensembles partiellement ordonnés. Ce modèle est paramétré par le type des éléments de l'ensemble et par la fonction de comparaison. Dans ce modèle de classe, nous plaçons une méthode retournant vrai si et seulement si deux éléments sont comparables.*

```
// ......... déclaration et définition partielle .............
template<typename T, bool compare(T,T)>
class EnsembleOrdonne
{
public:
  virtual bool comparables(T t1, T t2)const;
};

template<typename T, bool compare(T,T)>
bool EnsembleOrdonne<T,compare>::comparables(T t1, T t2)const
 {if (compare(t1,t2) || compare(t2,t1)) return true; else return false;}
```

Nous créons deux spécialisations de ce modèle, l'une représentant les entiers munis de l'ordre naturel (qui est un ordre total) et l'autre représentant les entiers munis de l'ordre partiel de divisibilité.

```
// ......... instanciation ..................................

bool inf(int i1, int i2){return i1<i2;}
bool divise(int i1, int i2){return (i2%i1==0);}

int main()
{ EnsembleOrdonne<int,inf> E1;
  cout << E1.comparables(3,4);

  EnsembleOrdonne<int,divise> E2;
```

```
  cout << E2.comparables(8,4);
  cout << E2.comparables(4,8);
  cout << E2.comparables(5,4);
}
```

Variables de classe Lorsque l'on instancie un modèle de classe disposant d'une variable de classe (introduite par `static` en C++), on obtient autant de variables de classe que de spécialisations du modèle.

Par exemple, si on désire ajouter au modèle de classe `Assoc` une variable de classe `nbAssoc` qui sert à comptabiliser le nombre d'associations créées, on écrit tout d'abord dans `Assoc.h` :

```
template<typename TypeCle, typename TypeValeur>
class Assoc{
private:
...........
  static int nbAssoc;
...........
};
```

Puis on ajoute à `Assoc.cc` :

```
template<typename TypeCle, typename TypeValeur>
int Assoc<TypeCle,TypeValeur>::nbAssoc=0;
```

Si on crée deux spécialisations, par exemple grâce aux expressions :

```
typedef Assoc<string, Date> CarnetAnniversaire;
typedef Assoc<string, string> Ass;
```

On dispose alors de deux variables, en l'occurrence `CarnetAnniversaire::nbAssoc` et `Ass::nbAssoc`.

L'utilisation de variables de classe peut être une autre manière de résoudre nos problèmes de valeurs par défaut pour l'association. Ces valeurs peuvent être déclarées dans le modèle comme des variables de classe.

```
template<typename TypeCle, typename TypeValeur>
class Assoc{
private:
..........
  static TypeCle cleDefaut;
  static TypeValeur valeurDefaut;
};
```

Puis elles sont définies et initialisées seulement dans le contexte d'une instanciation.

```
#include<iostream>
#include<string>
#include"Assoc.cc"

template class Assoc<string, float>;
template ostream& operator<<(ostream&, const Assoc<string, float>&);
string Assoc<string, float>::cleDefaut="inconnue";
float Assoc<string, float>::valeurDefaut=0;
```

Cette solution peut être plus légère syntaxiquement que celle qui consistait à passer ces valeurs comme paramètres du modèle de classe.

Modèles et héritage On peut associer héritage et paramétrage de diverses manières.

Exemple 9 (un modèle dérive d'un autre modèle)

```
template <typename T>
class Conteneur{};

template <typename T>
class Liste : public virtual Conteneur<T>{};
```

Exemple 10 (un modèle dérive d'une classe)

```
class Graphe{};

template <typename TypeEtiquette>
class GrapheEtiquete : public virtual Graphe{};
```

Exemple 11 (une classe dérive d'un modèle)

```
class Point : virtual public Assoc<int,int>{};
```

Spécialisation d'une fonction ou d'une méthode Il est souvent utile de redéfinir un modèle de fonction ou une méthode d'un modèle de classe pour un cas particulier, par exemple :

- on veut redéfinir le modèle de recherche dans un tableau pour le cas particulier des tableaux de caractères où l'on désire ne pas tenir compte des majuscules et des minuscules ;
- on veut redéfinir la méthode `affiche` d'association pour le cas où le type des clés est un pointeur sur une chaîne et le type des valeurs est un réel.

Pour résoudre ces problèmes, on fait cohabiter le modèle de fonction ou de méthode avec la forme spécialisée. Par exemple, nous résoudrons notre second problème ainsi.

```
//............ Assoc.cc contient toujours .........
template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::affiche(ostream &os) const
  {os <<getCle() << ", " <<getValeur();}

//............ AssocPtrStringFloat.cc contiendrait les spécialisations
template class Assoc<string*, float>;
template ostream& operator<<(ostream&, const Assoc<string*, float>&);

void Assoc<string*, float>::affiche(ostream &os) const
  {os << "clef : " << *getCle() << "-- valeur :" << getValeur();}
```

La définition d'un modèle peut même être incomplète lorsqu'il n'est possible d'écrire une méthode que dans le cadre d'une spécialisation.

Chapitre 7

STL : la librairie standard C++

7.1 Introduction

STL (pour *Standard Template Library*) est une librairie de classes et fonctions utilitaires :
— à l'origine définie par A. Stepanov et M. Lee chez Hewlett Packard,
— adoptée en 1994 par le comité de standardisation de C++ pour être intégrée au langage.
Elle est intéressante à deux titres :
— les programmes sont portables sur tous les compilateurs C++ respectant la norme *International Standard ISO/IEC 14882, sept. 98*. Dans la mesure où STL est intégrée au langage, elle est amenée à suivre les évolutions du langage.
— les algorithmes et structures de données proposés sont efficaces et facilement applicables.
Elle propose les composants suivants :
— des « conteneurs » (collections d'éléments),
— des « itérateurs » (pointeurs généralisés sur les conteneurs),
— des algorithmes génériques,
— des classes représentant les fonctions les plus utilisées (destinées à servir de paramètres aux fonctions génériques),
— des « adaptateurs » qui proposent essentiellement des spécialisations des conteneurs, itérateurs et classes-fonctions,
— des « allocateurs » qui permettent de gérer l'espace de stockage.

Les divers *include* à effectuer sont indiqués dans les exemples. D'une façon générale, le fichier porte le même nom que la classe. Voir l'annexe pour plus de détails.

[ROB 99] décrit en détail comment est faite STL et comment l'utiliser. Vous trouverez aussi un certain nombre d'informations (inégales) sur le web, dont une documentation en ligne à l'adresse `www.sgi.com/tech/stl`.

7.2 Conteneurs

Les conteneurs sont des collections de données paramétrées par le type des éléments stockés et optionnellement une fonction de comparaison (pour les collections qui trient leurs éléments) ainsi qu'un mode d'allocation. Il y en a deux grandes catégories (figure 7.1),
— les *séquences*, qui stockent séquentiellement les éléments et y accèdent séquentiellement ou directement suivant le cas,
— les *conteneurs associatifs*, qui associent une clé à chaque objet stocké, ces clés servant à récupérer efficacement les objets.
Ils partagent un certain nombre d'opérations, telles que :
— divers constructeurs (dont un sans paramètres),
— `begin()` ou `end()` qui retournent respectivement un itérateur sur la position qui repère le début de la séquence, ou juste après le dernier élément,
— `int size()` qui retourne le nombre d'éléments stockés,

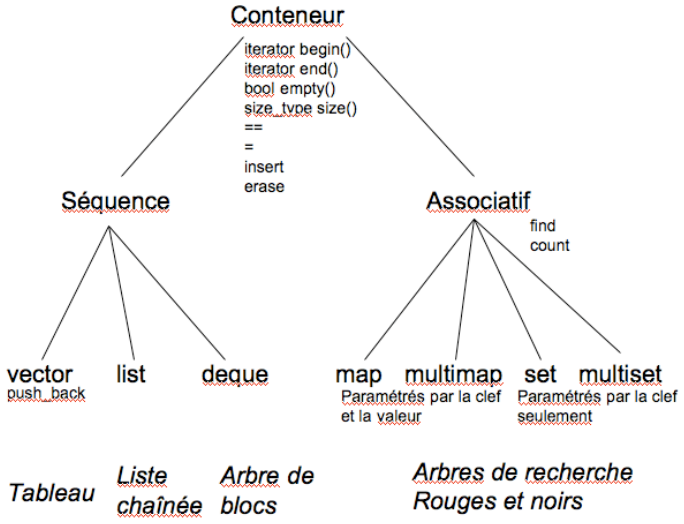


FIGURE 7.1 – Les conteneurs avec quelques méthodes

— `bool empty()` qui retourne vrai ssi le conteneur est vide,
— divers opérateurs de comparaison

D'autres opérations sont assez communément partagées, avec des signatures et des effets parfois différents suivant le conteneur :

— insertion d'éléments avec `insert(...)`, `push_back(...)`, `push_front(...)`,
— effacement d'éléments avec `erase(...)`, `clear(...)`, `pop_back()`, `pop_front()`,
— accès avec `operator[](int)`, `at(int)`, `front()`, `back()`.

7.2.1 Séquences

vector

Un vecteur est implémenté à l'aide d'un bloc de mémoire contigu, semblable à un tableau, qui s'agrandit automatiquement lorsque sa taille n'est pas suffisante (recopies internes).

Complexité des opérations :

— ajout et effacement à la fin en temps constant si on considère la complexité amortie.
— ajout et effacement internes linéaires dans la taille du tableau (recopies internes).

Un exemple

```
#include<iostream>
#include<vector>

void main()
{
  vector<int> v;
  v.push_back(4);
  v.push_back(7);
  v.push_back(3);
  v.push_back(8);

  cout << "taille=" << v.size() << endl;

  cout <<"------ affichage  du vecteur --------------"<< endl;
  for (int ii=0; ii<v.size(); ii++)
```

```
    cout << v[ii] << " ";
}
```

list

La liste est implémentée à l'aide d'un ensemble de cellules doublement chaînées. Elle occupe donc un peu plus de place que le vecteur pour un même ensemble d'éléments.

— Complexité des opérations : ajout et effacement en temps constant à n'importe quelle position.

Un exemple

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

int main()
{
  list<float> liste;
  liste.push_back(4);
  liste.push_back(7);
  liste.push_back(3);
  liste.push_back(8);
  cout << "taille liste=" << liste.size() << endl;
}
```

deque

Cette structure est une sorte de compromis entre les listes et les tableaux.

Les queues « à double entrée » sont implémentées à l'aide d'un tableau qui référence n tableaux secondaires stockant les éléments eux-mêmes. Dans les tableaux secondaires les éléments sont insérés à partir du centre. Lorsqu'un tableau secondaire est rempli (au moins du centre vers une extrémité), un autre tableau secondaire est créé. Les tableaux secondaires sont insérés dans le tableau principal en partant aussi du centre.

Cela permet des complexités constantes d'ajout en début et en fin, tout en occupant moins d'espace que la liste. Par contre l'insertion au milieu est toujours linéaire.

Ces éléments peuvent être accédés par l'opérateur [ ], ce qui n'était pas le cas pour les listes.

Un exemple

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

int main()
{
  deque<int> li;
  li.push_back(4);
  li.push_back(7);
  li.push_back(3);
  li.push_back(8);
  li.pop_front();

  cout << "-------- deque --------------"<< endl;
  for (int d=0; d<li.size(); d++)
    cout << li[d] << " ";
  cout << endl;
}
```

7.2.2 Conteneurs associatifs

Dans ces structures, les éléments sont identifiés (et récupérés) par une clé. Pour des raisons d'efficacité, ces conteneurs sont implémentés à l'aide d'arbres rouges et noirs (arbres de recherche équilibrés), dans lesquels l'insertion et la récupération sont en $log(n)$.

map

C'est la forme la plus classique, où les clés et les valeurs sont distinctes, et les clés sont uniques. Une `map` est paramétrée par le type des clés, le type des valeurs et optionnellement par la fonction de comparaison des clés et un allocateur. Elle contient des instances de la classe paramétrée `pair`, instanciée avec le type des clés et le type des valeurs.

```
#include<iostream.h>
#include<string>
#include<map>
#include<algo.h>
#include <pair.h>

// Attention : la classe Date est ecrite en version "condensée" pour raccourcir le texte
class Date
{
private:
  int jour, mois, annee;

public:
  Date(int j=1, int m=1, int a=2001){jour=j; mois=m; annee=a;}
  virtual ~Date(){}
  virtual void affiche(ostream& os)const{os<<jour<<"/"<<mois<<"/"<<annee;}
};

ostream& operator<<(ostream& os, const Date& D){D.affiche(os); return os;}

ostream& operator<<(ostream& os, const pair<const string,Date>& p)
{os<<p.first<<" "<<p.second; return os;}

void main()
{
  map<string,Date > Anniversaires;
  Anniversaires["Julie"]=Date(8,5,1988);
  Anniversaires["Cecilia"]=Date(18,12,1991);
  Anniversaires.insert(pair<const string,Date>("Eloise", Date(11,4,1998)));
}
```

multimap

Dans cette variante, les clés sont multiples.

```
#include<iostream.h>
#include<string>
#include<multimap.h>
#include<algo.h>
#include <pair.h>

// .......... avec la meme classe Date

void main()
```

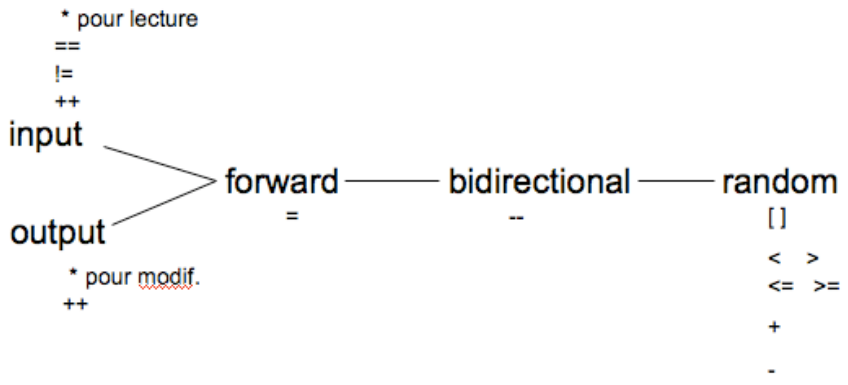


FIGURE 7.2 – Les itérateurs et leurs opérations

```
{
  multimap<string,Date> AnniversairesBis;
  AnniversairesBis.insert(pair<const string,Date>("Julie",Date(8,5,1988)));
  AnniversairesBis.insert(pair<const string,Date>("Cecilia",Date(18,12,1991)));
  AnniversairesBis.insert(pair<const string,Date>("Eloise", Date(11,4,1998)));
  AnniversairesBis.insert(pair<const string,Date>("Julie",Date(12,3,1994)));
}
```

set

Dans le cas des ensembles, clés et valeurs sont confondues. Un exemple est donné plus loin, dans la section 7.4.

multiset

C'est une variante des ensembles qui accepte la répétition des éléments (on appelle parfois ceci une famille).

7.3 Itérateurs

Les itérateurs sont des pointeurs généralisés sur les conteneurs. Un itérateur est un objet qui retourne successivement une référence vers chaque élément d'un conteneur.

L'essentiel des algorithmes appliqués aux conteneurs les utilisent.

Il y en a cinq sortes (figure 7.2) :

— les `input iterator`, qui autorisent le déréférencement (`*` et `->`) pour la lecture seule, l'incrémentation (`++`), les comparaisons (`==` et `!=`),

— les `output iterator`, qui autorisent le déréférencement (`*` et `->`) pour l'écriture seule, et l'incrémentation (`++`),

— les `forward iterator`, qui autorisent en plus des opérations précédentes l'affectation (`=`),

— les `bidirectionnal iterator`, qui autorisent en plus des opérations précédentes la décrémentation (`--`),

— les `random iterator`, qui autorisent en plus des opérations précédentes l'accès direct à une position (`[]`) les comparaisons (`<`, `>`, `>=`, `<=`), et l'avancée ou le recul de n positions.

Chaque conteneur `C` dispose de son propre type d'itérateur, défini dans la classe, donc utilisable par l'expression `C::iterator`, ou `C::const_iterator`. La figure 7.3 présente schématiquement ces relations.

— les `input iterator` et les `output iterator` servent de base pour les itérateurs de *flot* : `istream_iterator`, `ostream_iterator` ;

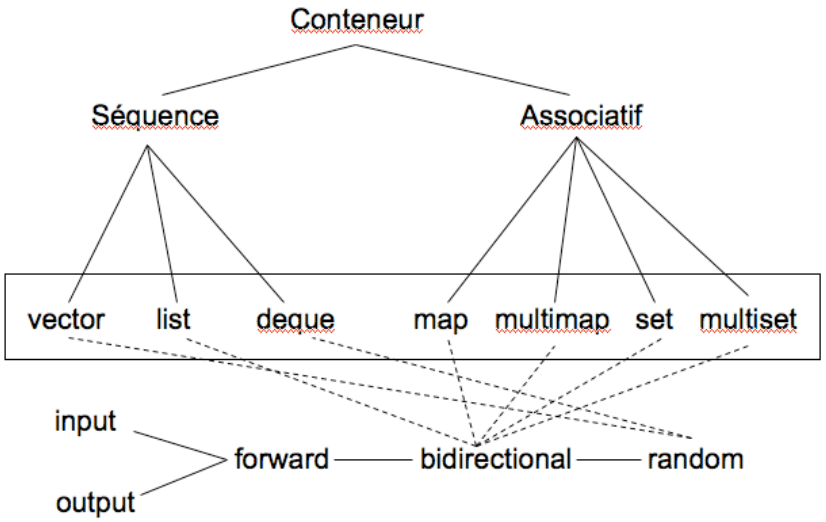


FIGURE 7.3 – Les liens entre conteneurs et itérateurs

— les `bidirectionnal iterator` servent pour les *conteneurs associatifs* et les *listes*, un accès direct par le numéro d'élément ne serait en effet pas efficace dans une telle structure et il n'est donc pas disponible ;
— les `random iterator` servent pour les *vecteurs* et les *queues à double entrée (deque)*.

Exemple 1 - pour insérer et afficher les éléments d'un vecteur/d'un ensemble

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

void main()
{
  vector<int> v;
  vector<int>::iterator i; // est un random iterator
  v.insert(v.end(), 4);
  v.insert(v.end(),7);
  v.insert(v.end(),3);
  v.insert(v.end(),8);

  cout <<"------ affichage du vecteur --------------"<< endl;
  for (i=v.begin(); i<v.end(); i++)
    cout << *i << " ";
  cout << "taille=" << v.size() << endl;

  cout <<"------ affichage du vecteur --------------"<< endl;
  vector<int>::iterator iter=v.begin();
  for (int ii=0; ii<v.size(); ii++)
    cout << iter[ii] << " ";

  set<int> s;
  set<int>::iterator is;  // est un bidirectionnal iterator
  s.insert(s.end(),4);
  s.insert(s.end(),7);
  s.insert(s.end(),3);
```

```
  s.insert(s.end(),8);

  cout <<"------ affichage de l'ensemble --------------"<< endl;
  for (is=s.begin(); is!=s.end(); is++)
    cout << *is << " ";
  cout << endl;
}
```

Exemple 2 - pour accéder aux éléments d'une `map`

```
void main()
{
.... // suite du programme de la section 2.2.1 (map).

  cout<<"--- acces a la cle et la valeur d'un element ---"<<endl;

// la fonction find retourne Anniversaires.end() si la cle n'est pas trouvee
// ce pourrait etre teste ici ...

  cout << (*(Anniversaires.find("Julie"))).first<<" "
       << (*(Anniversaires.find("Julie"))).second <<endl;
}
```

7.4 Algorithmes

La librairie offre une grande variété d'algorithmes :
- numériques (min, max, sommes partielles, accumulations, produits, etc.)
- tris
- modifiant les conteneurs (remplissage, copie, echanges, union, intersection, etc.)
- sans modification (recherche, application de fonction, inclusion, etc.)

Leur particularité est que ces algorithmes ne sont pas écrits spécifiquement pour tel ou tel conteneur, mais prennent comme paramètres des itérateurs et des fonctions.

Quelques exemples sont détaillés ci-dessous.

Numérique (accumule) Cet algorithme permet d'appliquer cumulativement une fonction binaire f aux éléments du conteneur. Par défaut cette fonction est l'addition. Dans l'exemple ci-dessous, le premier appel de `accumulate` effectue $v[3] + (v[2] + (v[1] + (v[0] + 0)))$.

```
#include<iostream>
#include<vector>
#include<algo>
#include<set>

void main()
{
  vector<int> v;
  v.insert(v.end(), 4);
  v.insert(v.end(),7);
  v.insert(v.end(),3);
  v.insert(v.end(),8);
  cout << accumulate(v.begin(),v.end(),0) << endl;

  set<int> s;
  s.insert(s.end(),4);
```

```
  s.insert(s.end(),7);
  s.insert(s.end(),3);
  s.insert(s.end(),8);
  cout << accumulate(s.begin(),s.end(),0) << endl;
}
```

Tri (quicksort en nlog(n))

```
... // ----- tri du vecteur
  sort(v.begin(),v.end());
... // ----- tri de la queue
  sort(li.begin(),li.end());
```

On peut également passer comme paramètre la fonction de comparaison à utiliser pour le tri (sinon c'est `less<>` qui sera présentée dans la section suivante).

Modificateur (affichage par copie dans un flot)

```
  // declaration d'un iterateur de flot (stream iterator)
  // qui permet de manipuler un flot de la meme maniere qu'un conteneur
  // l'espace sert de separateur

  ostream_iterator<int> outInt(cout, " ");
  copy(v.begin(),v.end(),outInt);   // pour le vecteur precedent
  copy(li.begin(),li.end(),outInt); // pour la queue precedente
  copy(s.begin(),s.end(),outInt);   // pour l'ensemble precedent
  copy(AnniversairesBis.begin(), AnniversairesBis.end(), outDate); // pour la multimap
```

Modificateur (intersection d'ensembles)

```
  cout <<"------ intersection de deux ensembles --------------"<< endl;
  set<int> s1;
  s1.insert(s1.end(),4);
  s1.insert(s1.end(),7);
  s1.insert(s1.end(),3);
  s1.insert(s1.end(),8);
  s1.insert(s1.end(),8); // sans effet !

  cout << "s1 = ";
  copy(s1.begin(),s1.end(),outInt); //outInt precedemment defini
  cout << endl;

  set<int> s2;
  s2.insert(s2.end(),4);
  s2.insert(s2.end(),9);
  s2.insert(s2.end(),5);
  s2.insert(s2.end(),8);
  cout << "s2 = ";
  copy(s2.begin(),s2.end(),outInt);
  cout << endl;

  vector<int> s3(4);
  vector<int>::iterator fin
    = set_intersection(s1.begin(),s1.end(),s2.begin(),s2.end(),s3.begin());
  cout << "partie utile de s3 = ";
```

```
  copy(s3.begin(),fin,outInt);
  cout << "taille="<< s3.size()<<endl;
  cout << "integralite de s3 = ";
  copy(s3.begin(),s3.end(),outInt);
  cout << endl;
```

Non Modificateur (application d'une fonction à tous les éléments) La séquence n'est pas modifiée ... mais l'état de ses éléments peut l'être.

```
void ajoute2(int& i)
{i+=2;}

....... // dans le main ........ pour la queue precedente .....
       // chaque élément est augmenté de 2

  for_each(li.begin(),li.end(),ajoute2);
```

7.5 Classes-Fonctions

7.5.1 Classes existantes

Ce sont des classes munies d'un opérateur `operator()`. Elles généralisent la notion de fonction. Il existe deux classes générales principales :
- `unary_function`, qui définit deux types associés `argument_type` et `result_type`,
- `binary_function`,qui définit trois types associés `first_argument_type`, `second_argument_type` et `result_type`.

Les autres sont des sous-classes de celles-là, par exemple `plus, minus, multiplies, divides, ..., less, greater, ...`
Vous pouvez les voir dans le fichier `stl_function.h`. Voici pour exemple un extrait du code de plus :

```
template <typename T>
class plus : public binary_function<T,T,T> {
  T operator()(const T& x, const T& y) const { return x + y; }
};
```

Elles sont très utilisées pour passer des paramètres aux algorithmes ou aux classes paramétrées (on ne peut pas passer d'opérateur comme paramètre).
Par exemple :

Utilisation dans une variante de `accumulate`

```
  cout << accumulate(s.begin(),s.end(),1,multiplies<int>()) << endl;
```

Utilisation avec `bind2nd`

`bind2nd` est une fonction qui prend comme arguments une fonction binaire et une valeur du même type que le deuxième argument de la fonction binaire. Son effet est de fixer la valeur de ce deuxième argument. Par exemple `bind2nd(less<int>(),5)` produit une fonction unaire qui repond vrai si son argument est inférieur à 5.
Dans l'exemple ci-dessous, on veut par exemple remplacer dans le vecteur toutes les valeurs inférieures à 5 par 5

```
  // à partir de (x,y)-> x<y, crée x -> x<5
  replace_if(v.begin(),v.end(),bind2nd(less<int>(),5),5);
```

7.5.2 Classes définies par l'utilisateur

On peut aussi en définir de nouvelles, par exemple un équivalent de `ajoute2` :

```
class Ajoute2 : public unary_function<int,int>
{public:
  Ajoute2(){}
  virtual int operator()(int& i){i+=2; return i;}
};
........
// exemple d'utilisation directe
void main()
{Ajoute2 x; int i=28;
 cout<<" i avant: "<<i<<endl;
 cout<<"retour de Ajoute2: "<<x(i)<<endl;
 cout<<"i apres: "<<i<<endl;
}

//exemple d'utilisation en paramètre d'un algorithme
  for_each(li.begin(),li.end(),Ajoute2());
```

7.6 Adaptateurs

Ce sont des spécialisations des conteneurs, des itérateurs et des classes fonctions.

7.6.1 Adaptateurs de conteneurs

Il en existe trois, dont les noms sont assez explicites, `stack`, `queue` et `priority_queue`.

7.6.2 Adaptateurs d'itérateurs

Ils permettent de modifier le comportement des itérateurs existants en obligeant des insertions toujours en tête ou en queue d'un conteneur, ou en parcourant le conteneur de la fin vers le début, plutôt que dans le sens habituel qui est inverse.

7.6.3 Adaptateurs de classes-fonctions

`not1` et `not2` permettent d'obtenir la négation d'un prédicat unaire ou binaire.

`ptr_fun` permet de transformer une fonction ordinaire en instance de classe-fonction sans effort de programmation.

```
int affiche2(int i)
{
  cout << i+2;
  return i;
}

// ... dans le main, sur la deque li ...
  for_each(li.begin(),li.end(),ptr_fun(affiche2));
```

7.7 Allocateurs

La librairie STL permet également de définir et d'utiliser différents modèles de gestion de la mémoire, en particulier la classe paramétrée `auto_ptr` qui implémente une sorte de ramasse-miettes.

7.8 Annexe : les fichiers à inclure

Ceux de la distribution standard :

```
#include <algorithm>
#include <deque>
#include <functional> //objets fonctions prédéfinis: unary_function, binary_function,..et less
#include <iterator> //itérateurs, fonctions, adaptateurs
#include <list>
#include <map> //map et multimap
#include <numeric> //fonctions numériques: accumule,...
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <utility>
#include <vector>
```

Le fichier `stl.h`,variable suivant les sites, en regroupe plusieurs et ajoute parfois quelques spécialités locales (consultez-le ; il est en général dans `/usr/include/g++`).

Bibliographie

[ROB 99] Robson R. *Using the STL, the C++ Standard Template Library*, second edition, Springer Verlag, 1999.

[STR 97] Stroustrup B. *Le langage C++*, troisième édition, Campus Press, 1997.

Chapitre 8

Spécialisation/généralisation et héritage multiple

8.1 Définition

Nous avons étudié précédemment l'héritage simple, caractéristique dont doit absolument disposer un langage pour se réclamer de l'approche par objets. Certains langages autorisent une version plus générale de l'héritage, communément appelée « héritage multiple », dans laquelle une classe peut avoir plusieurs super-classes directes.

Cela a deux conséquences immédiates :

— le graphe d'héritage a une structure de graphe sans circuit. Il peut avoir une racine, mais ce n'est pas obligatoire (ce n'est pas le cas en C++). Lorsqu'on ferme transitivement le graphe, on obtient un ordre partiel entre les classes ;

— l'ensemble des super-classes d'une classe n'est plus totalement ordonné par la relation d'héritage.

8.2 Discussion des avantages et inconvénients

8.2.1 Une meilleure classification

Lorsqu'on se place du point de vue de l'analyse, la relation de spécialisation/généralisation forme une classification des concepts et naturellement, une classification est multiple, car elle se fait en combinant plusieurs critères ou propriétés, ou valeurs pour ces propriétés. Il est important de ne pas restreindre la hiérarchie de classes à être organisée avec de l'héritage simple pour préserver sa bonne structuration.

multi-classification avec plusieurs critères Nous prenons comme premier exemple de multi-classification le domaine des mathématiques (figure 8.1). Dans ce contexte, les parallélogrammes (dont les côtés opposés sont parallèles) se spécialisent en rectangles (deux côtés successifs forment un angle droit) et losanges (les côtés sont de même longueur). Les carrés sont une spécialisation conjointe à la fois des rectangles et des losanges. Plusieurs critères ont été employés pour obtenir cette classification : longueur des côtés, propriétés des côtés opposés, propriétés des côtés consécutifs.

Le deuxième exemple considère une classification des collections dans un langage de programmation (figure 8.2). Deux critères de classification principaux sont utilisés : la possibilité d'étendre les collections, admise sous deux formes différentes dans les ensembles et les dictionnaires associatifs ; l'indexation des éléments, faite à l'aide d'entiers dans le cas des tableaux classiques et basée sur des clefs dans le cas des dictionnaires associatifs.

multi-classification avec un unique critère Mais l'utilisation d'un seul critère peut également conduire à une multi-classification par exemple lorsque la classification se base sur une

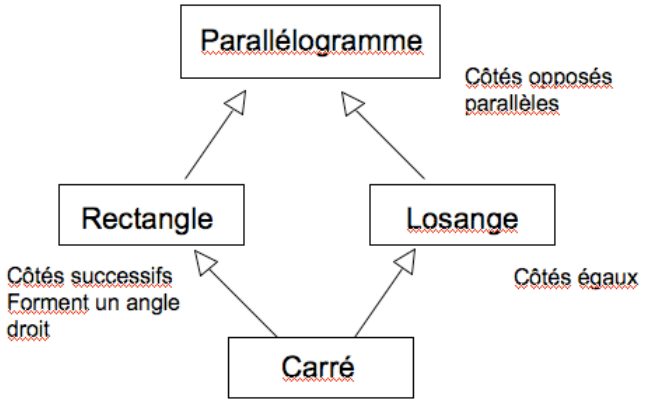


FIGURE 8.1 – multi-classification de quelques polygones

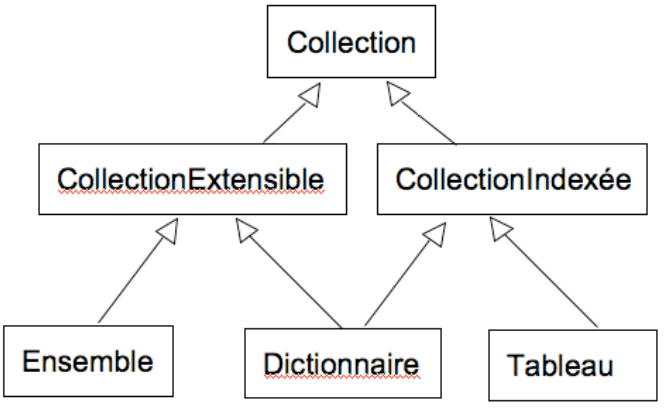


FIGURE 8.2 – multi-classification de quelques collections

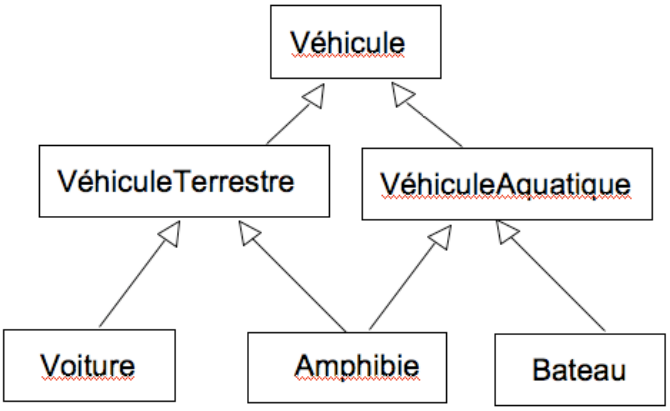


FIGURE 8.3 – multi-classification de quelques véhicules

caractéristique multi-valuée et dont les valeurs peuvent être cumulées par certains objets. La figure 8.3 présente une telle situation. Les véhicules définissent une zone d'utilisation possible. Dans le cas des véhicules terrestres (resp. aquatiques), cette zone d'utilisation est le milieu terrestre (resp. aquatique). Les véhicules amphibies seront naturellement classés comme des véhicules à la fois terrestre et aquatiques, donc sur ce même critères de la zone d'utilisation.

8.2.2 Un meilleur partage

Si l'héritage procure une bonne structuration des classes d'un programme, il sert également de support à la factorisation des attributs et des méthodes, évitant ainsi des redondances dans le code du programme. Cette élimination de la redondance est importante car elle facilite les corrections et les modifications puisqu'un seul point du programme contient une déclaration ou un traitement donné. Elle permet également de réaliser une économie d'écriture.

Un petit exemple vous convaincra aisément qu'une factorisation maximale ne peut être obtenue que grâce à l'héritage multiple.

Sur la partie haute de la figure 8.4, vous pouvez observer trois classes, `A`, `B` et `C`. Les trois classes partagent les propriétés `f` et `g` ; `h` est partagée par `A` et `C` ; `i` est partagée par `B` et `C`. L'héritage multiple offre différentes solutions pour effectuer une factorisation multiple ; deux d'entre elles vous sont proposées au bas de la figure (vous pouvez chercher les autres ...).

La figure 8.5 montre deux possibilités pour organiser les classes avec de l'héritage simple. Ces deux solutions, comme toute solution n'autorisant que l'héritage simple, contiennent de la redondance : dans la première, `i` est répété, dans la deuxième c'est le cas de `h`.

8.2.3 Conflits de valeurs et conflits de noms

Les conflits constituent la difficulté majeure lorsqu'on utilise l'héritage multiple. Cette situation se produit lorsque plusieurs propriétés de même nom sont héritées dans une classe.

Il existe deux types de conflits : les conflits de nom et les conflits de valeur.

Conflit de valeurs Ce cas se produit lorsque les propriétés portent un même sens, par exemple différentes méthodes `seDeplacer` dans les classes de véhicules terrestres et aquatiques : dans la classe des véhicules amphibies, la méthode `seDeplacer` est ambiguë, on ne sait pas quelle méthode choisir entre celle qui vient des véhicules terrestres et celle qui vient des véhicules aquatiques. Généralement, les propriétés qui produisent un conflit de valeur spécialisent une même propriété : dans notre exemple, il serait logique que la classe des véhicules dispose également d'une méthode `seDeplacer`, même abstraite. On dit que les propriétés en conflit ont une origine commune.

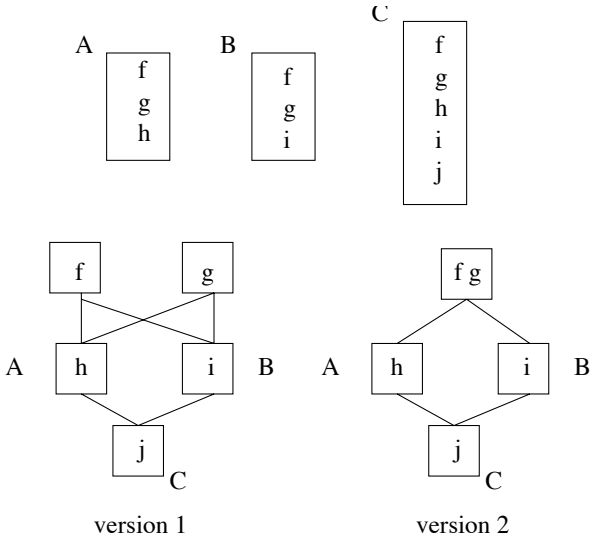


FIGURE 8.4 – Trois classes et deux factorisations maximales

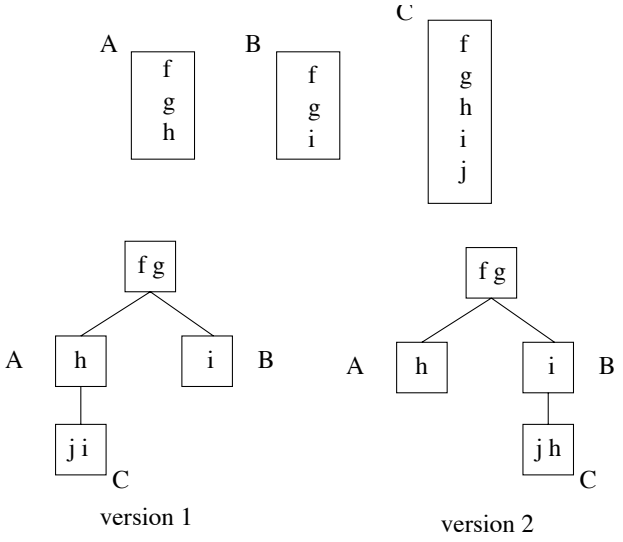


FIGURE 8.5 – Trois classes et deux factorisations en héritage simple

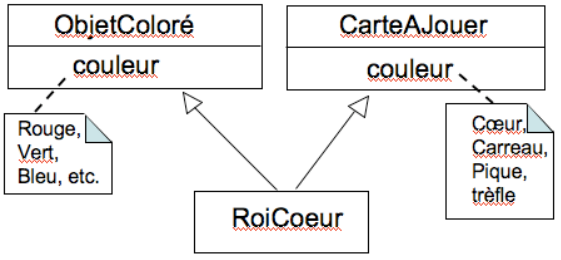


FIGURE 8.6 – Situation de conflit de nom

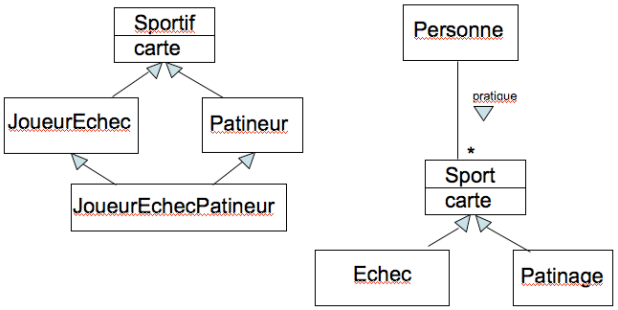


FIGURE 8.7 – héritage multiple répété (gauche), solution sans héritage répété (droite)

Conflit de nom Une situation de conflit de nom est illustrée sur la figure 8.6. Deux propriétés, de sémantiques différentes, sont présentes dans les super-classes d'une classe : `couleur` dans `ObjetColoré` décrit la couleur (chromatique) tandis que `couleur` dans `CarteAJouer` décrit la famille de la carte. En règle générale, lorsqu'il y a conflit de nom, il n'y a pas d'origine commune : pas de superclasse commune qui possède la propriété sur laquelle porte le conflit.

8.2.4 Héritage répété

L'héritage multiple demande de statuer sur le sort des attributs hérités par plusieurs chemins. La figure 8.7 (gauche) présente une modélisation demandant de l'héritage répété : un sportif qui est à la fois joueur d'échec et patineur a besoin de deux cartes de sport, chacune correspondant à l'une de ses activités. Avec de l'héritage répété, l'attribut `carte` est hérité deux fois car il y a deux chemins depuis la sous-classe `JoueurEchecPatineur`. Cette modélisation n'est cependant pas satisfaisante, elle représente typiquement un cas où il faut préférer de l'héritage simple (à droite de la figure) et une association : une personne pratique plusieurs sports, et pour chaque activité pratiquée, dispose d'une carte. La première modélisation est peu satisfaisante car elle entraîne rapidement une forte combinatoire de classes pour représenter toutes les combinaisons possibles de sports pratiqués.

8.2.5 Complexité de la classification

Les détracteurs de l'héritage multiple lui reprochent souvent la complexité de la structure résultat. Il est vrai que de manière générale, la hiérarchie peut devenir plus complexe (contenir plus de classes et plus d'arcs), mais cette notion de complexité est très relative. Une structure d'ordre partiel sans redondance peut contenir plus de liens et être cependant plus lisible qu'une structure arborescente. La figure 8.8, dans laquelle les noms des classes sont omis, en donne une illustration. Les redondances d'attributs dans la structure de droite la rendent peu lisible. La structure de gauche bénéficie au contraire d'une régularité de construction et d'une absence de redondances d'attributs.

8.3 Réalisation en C++

8.3.1 Résolution des conflits par désignation explicite

la résolution des conflits en C++ se réalise en utilisant l'opérateur de résolution de portée :: et en désignant explicitement la classe à partir de laquelle la propriété doit être recherchée.

Les instructions suivantes se soldent par une ambiguïté :

```
A *instA = new A; instA->f();
```

Trois solutions sont envisageables :

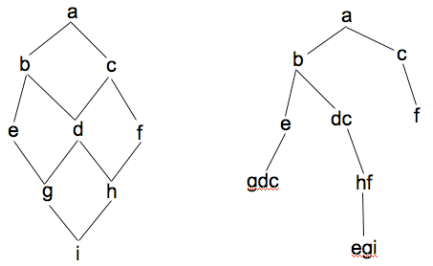


FIGURE 8.8 – Structure avec héritage multiple (gauche), héritage simple (droite)

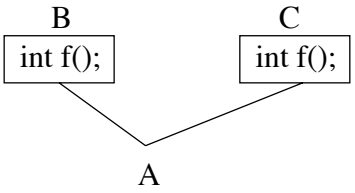


FIGURE 8.9 – Situation de conflit

— écrire `instA->B::f();`
— écrire `instA->C::f();`
— redéfinir `int f()` dans `A`

Ce principe de résolution (qui n'est pas incontournable, le langage Eiffel par exemple propose une élégante solution par renommage) a deux inconvénients majeurs.

Le premier est qu'il demande au programmeur d'avoir connaissance de la hiérarchie, ce qui va à l'encontre d'un certain principe de modularité et de localité selon lequel la connaissance des super-classes directes d'une classe devrait suffire pour l'écrire. Si l'on examine le cas de la figure 8.10, pour l'objet créé par l'instruction `D *instD=new D;`, l'ambiguïté de l'expression `instD->f()` n'est pas levée par `instD->A::f()` !

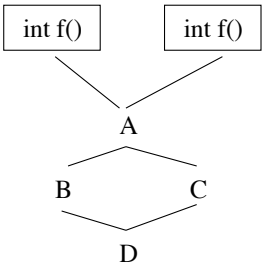


FIGURE 8.10 – Désambiguisation et connaissance de la hiérarchie

Le deuxième inconvénient, plus grave, est que la règle de spécialisation n'est plus assurée. Un programmeur peut passer par-dessus une propriété pour en appeler une autre. La figure 8.11 nous donne un exemple de ce problème : rien ne nous empêche d'écrire `C *instC=new C; instC->A::f();`, appelant ainsi pour les objets de la classe `C` une méthode qui n'est pas adaptée, en l'occurrence pas la plus spécifique.

8.3.2 Héritage répété versus héritage `virtual`

C++ offre la possibilité de faire de l'héritage répété (solution adoptée par défaut) ou de l'héritage non répété (qui demande d'utiliser le mot-clef `virtual` lors de la déclaration de la super-classe).

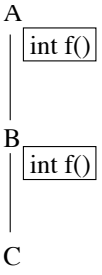


FIGURE 8.11 – Désambiguisation et règle de spécialisation

Nous utilisons la hiérarchie de classes de la figure 8.12 pour illustrer cette section.

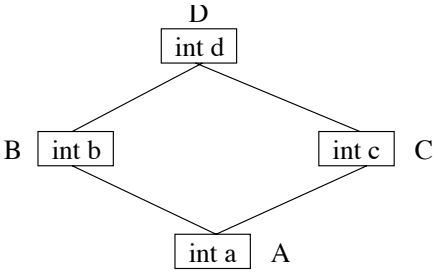


FIGURE 8.12 – Situation d'héritage : une instance de `A` possède-t-elle 1 ou 2 attributs `d` ?

Une instance de la classe `A` créée par le code suivant (héritage répété) aura deux attributs `d` comme le montre la figure 8.13.

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

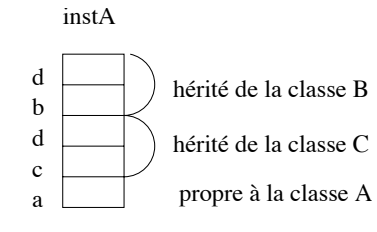


FIGURE 8.13 – Situation d'héritage répété en C++, vue logique de l'instance

Une instance de la classe `A` créée par le code suivant (héritage virtuel) aura un unique attribut `d` comme le montre la figure 8.14.

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

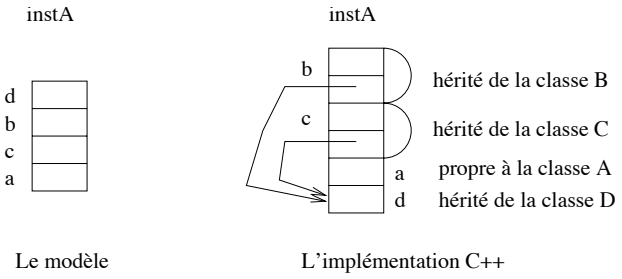


FIGURE 8.14 – Situation d'héritage virtuel en C++, vue logique de l'instance

8.3.3 Transmission des paramètres aux constructeurs

La figure 8.15 présente une particularité du passage de paramètres aux constructeurs d'une super-classe dans le cas de l'héritage virtuel : vous devez noter essentiellement dans le constructeur de la classe `D` la nécessité de réécrire `A(aa)` pour la transmission des paramètres alors que l'on pourrait penser en première approximation que cette transmission est effectuée par l'intermédiaire de `B(aa,bb)` et `C(aa,cc)`. En réalité C++ ignore le passage du paramètre `aa` dans ces deux écritures, supposant qu'elles pourraient être incohérentes : par exemple on aurait pu écrire (par erreur) `B(aa,bb)` et `C(bb,cc)` dans le constructeur de la classe `D`.

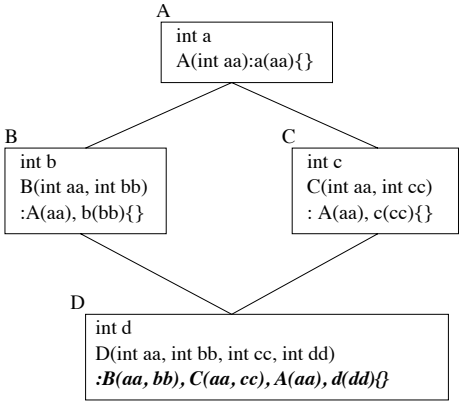


FIGURE 8.15 – Passage de paramètres aux constructeurs (héritage virtuel)

8.3.4 Ordre d'appel des constructeurs et destructeurs

Dans le cas de l'héritage simple, lors de la création d'un objet, les constructeurs des superclasses et de la classe de l'objet sont appelés depuis le haut jusqu'en bas de la hiérarchie. Les destructeurs sont appelés systématiquement dans le sens inverse de celui des constructeurs. Dans le cas de l'héritage multiple, ces superclasses ne sont pas totalement ordonnées par la relation d'héritage : un algorithme particulier de parcours des superclasses est utilisé pour déterminer un ordre total.

Le parcours est appliqué sur un graphe d'héritage dans lequel les classes héritées de manière répétées sont dupliquées autant de fois qu'il y a de chemins y menant depuis la classe de l'objet que l'on cherche à construire (ou à détruire). Cet ordre est donné par l'ordre de dépilement d'un parcours en profondeur d'abord (dans lequel on ne repasse pas par les sommets déjà explorés) partant de la classe de l'objet. Lorsqu'il y a plusieurs superclasses directes, les classes héritées de manière virtuelle sont d'abord explorées, puis les autres, dans chaque catégorie en suivant l'ordre de déclaration. Par exemple (figure 8.16), l'ordre local entre les super-classes de `B` est tout d'abord `E` (héritée de manière virtuelle) puis `Db` (héritée de manière répétée). Pour la classe `A`, on suppose que la classe est ainsi déclarée :

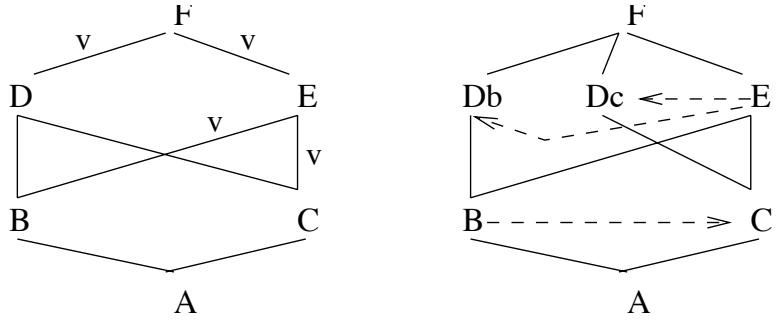
```
class A : public B, public C{.....};
```

Dans l'exemple donné figure 8.16, voici l'évolution de la pile dans un tel parcours :
A
A B
A B E
A B E F
A B E
A B
A B Db
A B
A
A C
A C Dc
A C
A
L'ordre de dépilement est donc : F E Db B Dc C A.

Supposons l'ordre local sur les superclasses directes d'une classe selon la règle : les virtuelles avant les autres, et dans chaque catégorie l'ordre de déclaration prévaut. On peut schématiser l'algorithme ainsi :

```
ordreTotal <- vide
p.empiler(racine)
tant que ( not p.vide()) faire
   si (p.sommet() a des successeurs non marqués
           empiler le plus petit successeur pour l'ordre local de p.sommet()
   sinon
            s=p.depiler()
            ordreTotal <- ordreTotal + s
            marquer s
```

Cet algorithme calcule une extension linéaire de l'ordre inverse de l'ordre induit par l'héritage et assure ainsi qu'une classe est ramassée (ordre de dépilement) avant toutes ses sous-classes.



F E Db B Dc C A

FIGURE 8.16 – une hiérarchie (gauche, l'héritage virtuel est mentionné par v sur le lien d'héritage), duplication des classes héritées de manière répétée (droite, l'ordre local apparaît en flèches pointillées), l'ordre d'appel des constructeurs (bas)

Un autre algorithme, moins efficace consiste à effectuer une descente en profondeur d'abord en respectant l'ordre inverse de déclaration :
A C Dc F E F B Db F E F
Dans cet ordre, on ne garde que les dernières occurrences de chaque sommet, puis on inverse le résultat.

Chapitre 9

Gestion des exceptions

9.1 Introduction

Les mécanismes de gestion d'exceptions ont deux motivations principales.

La première est d'améliorer (dans une certaine mesure que nous détaillerons) la résistance des programmes aux erreurs. Il s'agit au minimum de :

- détecter certaines erreurs,
- prendre en charge leur traitement sans que le programme ne s'arrête brutalement par une erreur majeure, et en le déroutant de son fil normal,
- assurer la cohérence des données après exécution puisqu'en cas d'erreur, il n'y a pas d'arrêt brutal du programme. On peut introduire des procédures de sauvegarde des données, de mise à jour ou de retour vers des versions antérieures.

La deuxième est d'offrir une meilleure réutilisation des opérations en étendant leur domaine de définition puisque les pré-conditions seront généralement vérifiées et prises en charge par le mécanisme de gestion d'exceptions.

Dans l'écriture d'un logiciel, il est intéressant de distinguer les cas généraux d'utilisation des objets et les cas particuliers. Ces derniers sont souvent nombreux et peuvent correspondre à un volume de code important. On retiendra donc que le mécanisme de gestion d'exceptions a pour rôle de séparer le code normal, usuel, du code de gestion des situations exceptionnelles.

9.2 Modélisation

Il y a toujours une ambiguïté dans la notion d'exception à laquelle il est difficile d'échapper :

- une exception peut être un cas particulier dans un programme (prévu pour se produire mais assez rarement),
- une exception peut être une erreur, le non respect du contrat d'une classe ou d'une opération (qui ne devrait pas se produire).

Certaines équipes de programmation privilégient l'un ou l'autre des deux usages. Il n'est pas toujours évident de savoir dans quelle catégorie une exception se trouve.

Cas particulier dans un traitement Nous donnons quelques exemples de cas particuliers dans un programme qui peuvent être traités par un mécanisme de gestion d'exceptions. Nous n'encourageons pas ce style de programmation dont nous considérons qu'il relève de la programmation normale mais il est bon de le connaître.

- On implémente un vecteur par un tableau alloué dynamiquement. Le cas normal de l'ajout d'un élément (tableau non rempli) consiste à insérer l'élément dans le tableau. Le cas exceptionnel se produit lorsque le tableau est plein : il faut en réallouer un plus grand et recopier les valeurs existantes avant d'insérer normalement la nouvelle valeur.
- On lit un fichier, le cas normal consiste à passer à l'enregistrement suivant. Le cas exceptionnel survient lorsque la fin du fichier est atteinte.

Contrat d'une classe ou d'une opération Le contrat d'une opération (méthode ou fonction) se compose essentiellement de deux parties :

- les pré-conditions, assertions qui doivent être vraies avant d'entrer dans le code de l'opération. Voici quelques cas de pré-conditions non respectées.
  - Diviser par `zéro`,
  - `depiler` une pile vide,
  - `setJour` d'une `Date` avec la valeur 32.
- les post-conditions, assertions qui doivent être vraies après le retour de l'opération. Voici quelques cas de post-conditions non respectées.
  - Débordement dans une opération arithmétique,
  - écriture d'un fichier sur une disquette pleine,
  - réservation d'un emplacement dans une mémoire saturée.

Le contrat d'une classe se compose principalement :

- des invariants, assertions vérifiées par les objets d'une classe, très souvent des propriétés sur les valeurs des attributs. Voici quelques exemples d'invariants de classe.
  - Dans une classe `Personne`, l'attribut `age` doit varier entre 0 et 140.
  - Si on stocke dans un attribut la date de naissance et dans un autre attribut l'âge, ils doivent toujours être cohérents.

En l'absence de mécanisme de gestion d'exceptions, deux procédés bien connus sont employés.

- *Traiter le problème dans l'algorithme de l'opération*
  → le code de l'opération est alourdi,
  → le traitement des erreurs est figé dans l'opération.
- *L'opération retourne un code d'erreur*
  → le code appelant l'opération doit se préoccuper du traitement des situations exceptionnelles, et possède le même défaut de lourdeur. Le traitement est cependant moins figé car il est traité plus près du contexte où l'erreur s'est produite.

Cette problématique est déjà ancienne dans le domaine de la programmation, et la plupart des langages y répondent par un traitement des exceptions, qui, comme nous allons le voir, a pour objectif de détourner le programme de son cours normal et de proposer des localisations particulières dans le code pour le traitement des problèmes.

La plupart des langages à objets actuels proposent une évolution de ce modèle dans laquelle les exceptions sont des objets comme les autres, ce qui a plusieurs avantages.

- La récupération des exceptions se fait d'après des types (leurs classes), leur sémantique est donc explicite et facilite la lisibilité du programme.
- Les types d'exceptions peuvent être classés grâce à l'héritage.
- Les objets exceptions peuvent transporter des informations (dans leur attributs) depuis le contexte où l'erreur est survenue jusqu'au contexte où elle est traitée ; certains traitements typiques peuvent être intégrés dans les opérations de la classe exception. Par exemple une exception `livreAbimé` décrivant une erreur dans une opération `rendre` d'une classe bibliothèque pourrait détenir parmi ses attributs la référence du client et comme opération une méthode `facturePénalités`.

9.3 Définition d'exceptions en C++

Nous complétons la classe `Date` déjà vue précédemment par un mécanisme de gestion d'exceptions. Nous rappelons tout d'abord quelques extraits de la classe en question.

```
//--------------- Date .h-------------------------------------------------------
class Date
{
private:
int annee, mois, jour;
public:
Date();
Date(int,int,int);
```

```
virtual ~Date();
virtual int getAnnee()const;
virtual int getMois()const;
virtual int getJour()const;
virtual void setAnnee(int);
virtual void setMois(int);
virtual void setJour(int);
virtual void modifier(int j, int m, int a);
virtual void affiche(ostream&)const;
static int NbrJoursMois(int, int);
};

//--------------- Date .cc-----------------------------------------------------
Date::Date(){annee=2000; mois=12; jour=31;}
Date::Date(int j, int m, int a) { jour = j; mois = m; annee = a;  }
void  Date::modifier(int j, int m, int a) {  jour = j; mois = m; annee = a;  }
Date::~Date(){}
int Date::getAnnee()const{return annee;}
int Date::getMois()const{return mois;}
int Date::getJour()const{return jour;}
void Date::setAnnee(int a){annee=a;}
void Date::setMois(int m){mois=m;}
void Date::setJour(int j){jour=j;}
void Date::affiche(ostream&)const{cout << jour << "/" << mois << "/" << annee << endl;}
int Date::NbrJoursMois(int mois, int annee)
// retourne le nombre de jours du mois
// une annee est bissextile
// pour les siecles, seulement si divisible par 400
// pour les autres annees, si divisible par 4
{switch (mois)
 {
   case 4:case 6:case 9:case 11: return 30;
   case 2: if ((annee%100==0 && annee%400==0)
             || (annee%100!=0 && annee%4==0))
             return 29;
           else return 28;
   default: return 31;
 }}
```

Pour cette classe, différentes erreurs peuvent survenir, nous étudierons trois d'entre elles concernant :

— une valeur de jour erronée,
— une valeur de mois erronée,
— la combinaison des deux erreurs précédentes.

Comme nous l'avons dit précédemment, les erreurs vont être représentées par des classes ordinaires, ce peut même être des classes paramétrées. Nous embarquerons dans leurs attributs les valeurs susceptibles d'être fausses. Nous les organisons dans une hiérarchie de trois classes décrites comme suit. Les opérations consistent ici simplement à afficher un message d'erreur.

```
class ErreurDate
{
 protected: int jour; int mois; int annee;
 public:
  ErreurDate();
  ErreurDate(int, int, int);
  virtual ~ErreurDate();
```

```
  virtual void verdict(ostream& o)const;
};

ErreurDate::ErreurDate(){}
ErreurDate::ErreurDate(int j, int m, int a):jour(j), mois(m), annee(a){}
ErreurDate::~ErreurDate(){}
void ErreurDate::verdict(ostream& o)const
{ o << jour << "/" << mois << "/"<< annee << " ... pas terrible comme date !" << endl;}


class JourIncorrect : virtual public ErreurDate
{
 public:
  JourIncorrect();
  JourIncorrect(int, int, int);
  virtual ~JourIncorrect();
  virtual void verdict(ostream& o) const;
};

JourIncorrect::JourIncorrect(){}
JourIncorrect::JourIncorrect(int j, int m, int a):ErreurDate(j,m,a){}
JourIncorrect::~JourIncorrect(){}
void JourIncorrect::verdict(ostream& o)const
{
 switch (mois)
 {
  case 4:case 6:case 9:case 11: o << "30 jours pour le mois numero " << mois << endl; break;
   case 2: if ((Date::NbrJoursMois(mois, annee)==28))
                o << "28 j. en fevrier les annees non bissextiles !" << endl; break;
   default: o << "31 jours pour le mois numero " << mois << endl;
   }
 }

 class MoisIncorrect : virtual public ErreurDate
{
 public:
  MoisIncorrect(int, int, int);
  virtual ~MoisIncorrect();
  virtual void verdict(ostream& o);
};

MoisIncorrect::MoisIncorrect(int j, int m, int a):ErreurDate(j,m,a){}
MoisIncorrect::~MoisIncorrect(){}
void MoisIncorrect::verdict(ostream& o)
   {o << mois << " est un numero de mois incorrect" << endl;}

 class JourMoisIncorrect
: virtual public JourIncorrect, virtual public MoisIncorrect
{
 public:
  JourMoisIncorrect(int,int,int);
  virtual ~JourMoisIncorrect();
  virtual void verdict(ostream& o);
};

JourMoisIncorrect::JourMoisIncorrect(int j, int m, int a)
```

```
                          :JourIncorrect(j,m,a),MoisIncorrect(j,m,a),ErreurDate(j,m,a){}
JourMoisIncorrect::~JourMoisIncorrect(){}
void JourMoisIncorrect::verdict(ostream& o)
     {o << "Jour et mois sont incorrects" << endl;}
```

9.4 Déclaration et signalement

Le signalement d'une exception se réalise grâce à l'instruction `throw`.
Par exemple, le code de la méthode de modification d'une date peut se réécrire ainsi

```
void  Date::modifier(int j, int m, int a)
{
if (((m<1) || (m>12)) && ((j<1) || (j>NbrJoursMois( m, a))))
      throw JourMoisIncorrect(j,m,a);
  if ((m<1) || (m>12))
      throw MoisIncorrect(j,m,a);
  if ((j<1) || (j>NbrJoursMois( m, a)))
      throw JourIncorrect(j,m,a);
  jour = j; mois = m; annee = a;
  }
```

Si on désire spécifier les exceptions qu'une fonction ou une méthode peut signaler, on complète la signature (pour une méthode de `Date` dans le fichier Date.h et dans le fichier Date.cc). La méthode ne peut alors signaler que des exceptions qui sont des objets des classes mentionnées (ou de leurs sous-classes).

Pour plus de précision, lorsque l'on ne met rien, n'importe quelle exception est susceptible d'être signalée par la méthode :

```
void modifier(int j, int m, int a)
```

Lorsque l'on met une clause de spécification d'exception, cela limite les exceptions signalables par la méthode.

```
void modifier(int j, int m, int a) throw (ErreurDate,ErreurCalendrier)
```

La redéfinition de la méthode dans une sous-classe ne pourra ensuite que restreindre les exceptions déclarées, soit par un retrait, soit par une spécialisation. Dans une sous-classe de `Date` on pourra ainsi redéfinir `modifier` :

```
void modifier(int j, int m, int a) throw (JourIncorrect)
```

Mais on ne pourra pas ajouter une nouvelle exception (qui ne serait pas sous-classe de `ErreurDate` et `ErreurCalendrier`).

9.5 Récupération

La récupération des erreurs s'effectue grâce à la structure de contrôle suivante.

`try` *BlocProtégé*
`catch` *(déclaration1) BlocTraitement 1*
`catch` *(déclaration2) BlocTraitement 2*
...

Le mécanisme peut se décrire de la manière schématique suivante :

▷ `throw(e)` → interruption du programme

▷ Propagation sur les blocs `try .. catch ..` englobants jusqu'au premier `catch` qui contient une déclaration compatible avec `e` et destruction au passage de tous les objets locaux créés dans le bloc `try` ou les fonctions qu'il appelle

▷ Exécution du bloc du `catch` avec liaison de `e` avec le paramètre formel du `catch`

▷ Continuation par défaut : instruction suivant l'instruction `try`

Pour donner un exemple simple, voici un petit programme qui itère tant que la date saisie n'est pas conforme.

```
int main()
{

 Date D;
int i, j, m, a;
bool DateCorrecte = false;

while (!DateCorrecte)
{
  cout << "date ? "; cin >> j >> m >> a;
  try { D.modifier(j, m, a); DateCorrecte = true;}
  catch (JourMoisIncorrect JMI) {JMI.verdict(cout);}
  catch (JourIncorrect JI) {JI.verdict(cout);}
  catch (MoisIncorrect MI) {MI.verdict(cout);}
}
}
```

Mais dans le cas général, plusieurs bloc de récupération peuvent être imbriqués.

```
try
{
  try
  {
    try {... /* levée de Z() */ ...}
    catch(X) {...}
    catch(Y) {...}
    .....
  }
  catch(Z) {...}
  catch(V) {...}
  ..... /* reprise ici */
}
catch(U) {...}
....
```

Et ainsi pour notre exemple, nous pouvons imbriquer deux niveaux de structures de récupération. Notez qu'ici on a de plus décidé de renvoyer une exception au niveau appelant.

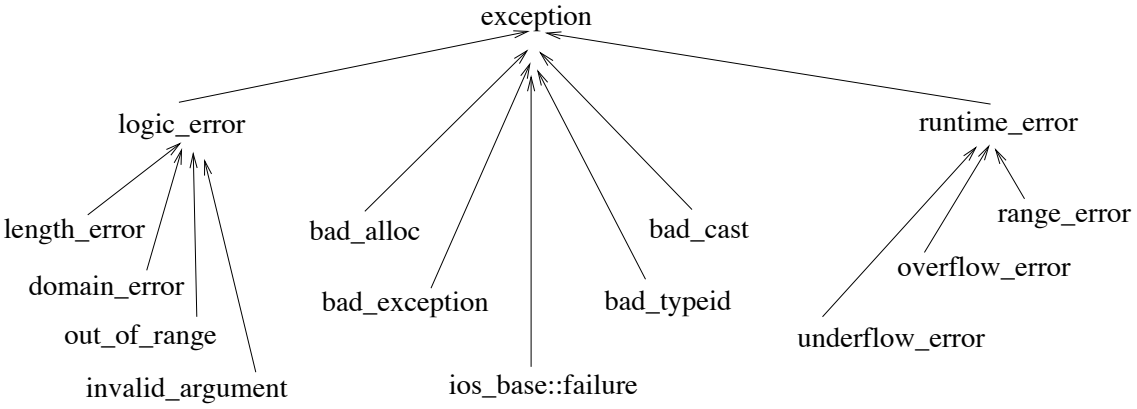
```
void f2(Date& D) throw (ErreurDate)
{
 try { D.modifier(34, 64, 1997); cout << "suite ! ";}
  catch (JourMoisIncorrect JMI) {JMI.verdict(cout);
                                 throw ErreurDate(34, 64, 1997);}
 catch (JourIncorrect JI) {JI.verdict(cout);
                           throw ErreurDate(34, 64, 1997);}
 catch (MoisIncorrect MI) {MI.verdict(cout);
```

```
                                throw ErreurDate(34, 64, 1997);}
}

int main()
{
 Date DD(1, 1, 1998);
 try {f2(DD); cout << "suite ! ";}
 catch (ErreurDate ED) {ED.verdict(cout);}
 }
```

9.6 Exceptions prédéfinies

Comme le montre la figure **??**, C++ prédéfinit assez peu d'exceptions, par comparaison avec Java. Le programmeur est invité à ajouter ses propres erreurs dans la hiérarchie, mais il n'y a pas d'obligation.



Les classes `logic_error` and `runtime_error` sont les plus générales (après `exception`), et représentent respectivement les problèmes dans la logique d'un programme et les erreurs dues aux conditions d'exécution. Leurs sous-classes, d'un niveau encore général, sont destinées à être spécialisées.

9.7 Exceptions du mécanisme de gestion d'exceptions

Deux problèmes du mécanisme de gestion d'exceptions sont gérés par appel d'une fonction que l'on peut modifier.

▷ une exception est signalée qui n'est prévue dans aucun `try..catch..`

la fonction `terminate()` qui appelle `abort` par défaut est appelée

▷ une exception est signalée qui n'est pas prévue dans la déclaration de la fonction

la fonction `unexpected()`, qui appelle `terminate()` par défaut, est appelée

Dans les deux cas, terminaison anormale du programme, mais on peut redéfinir la terminaison, en utilisant les fonctions `set_terminate` et `set_unexpected`

Nous en donnons un petit exemple.

```
// ------------------- déclaration de deux fonctions --------------------------------
void f(){cout << "le programme va terminer brutalement" << endl;}
void fu(){cout << "une erreur due a un probleme de specification d'exception" << endl;}

// ------ association de ces fonctions aux fonctions de terminaison anormales -
set_terminate(f);
set_unexpected(fu);
```

```
//------------------ cas d'appel de terminate
int main()
{Date D; D.modifier(34, 64, 1997);}

//------------------ cas d'appel de unexpected

class ErreurJustePourIllustrerUnexpected{};

void  Date::modifier(int j, int m, int a) throw (ErreurDate)
{
throw ErreurJustePourIllustrerUnexpected();
.........
  }

  int main()
{Date D; D.modifier(34, 64, 1997);}
```

Chapitre 10

Contrôle d'accès statique

10.1 Définition et usage

Les langages à objets à typage statique, famille à laquelle appartient C++, développent différentes stratégies pour contrôler à la compilation le bien fondé d'un envoi de message à un objet.

- Le type statique d'un objet, en particulier, est utilisé pour vérifier que l'objet saura répondre au message (si le type statique est une classe il s'agira de vérifier qu'elle possède bien une méthode de signature conforme à l'envoi de message) ;
- le `const` placé dans la signature d'une méthode permet de vérifier que l'objet receveur est une constante ;
- la généricité paramétrique participe à ce type de contrôle : dans une collection paramétrée par le type de ses éléments, on ne peut ranger que des éléments du bon type.

Le contrôle d'accès statique rentre dans cette même problématique mais porte un regard dual sur l'envoi de message. L'envoi de message implique :

- l'expéditeur (objet courant de la méthode où se trouve l'envoi de message, dont le type statique est pointeur constant de la classe) ;
- le message (nom de méthode, paramètres et retour pour les cas simples) ;
- le receveur (objet sur lequel est appliquée la méthode, cet objet a un type statique, type de la variable qui désigne l'objet).

La vérification effectuée par le contrôle d'accès consiste à répondre, en s'appuyant sur les types statiques, à la question générale :

> l'expéditeur a-t-il le droit d'envoyer le message au receveur ?

Examinons l'exemple suivant.

```
class Revendeur{
private:
 ....
public:
 ....
virtual vector<Produit*> vendre(string ref, int qty);
};

class Personne{
private:
string nom;
vector<Produit *> placard;
public:
```

```
  .....
virtual void achete(Revendeur* r, string ref,int qty);
};

void Personne::achete(Revendeur* r,string ref,int qty)
{
vector<Produit*> v = r->vendre(ref,qty);
........ // ajouter v au placard
}
```

La méthode `achete` contient (partie droite de l'affectation) un message dont
— l'expéditeur est caché sous l'identité `this`
— le receveur est `r`
— le message est `vendre(string,int)`

La vérification effectuée par le contrôle d'accès dans notre exemple consiste à répondre à la question particulière :

une *personne* peut-elle envoyer le message `vendre` à un *revendeur*

Les objectifs sont de différents niveaux :
— masquer l'implémentation, favoriser l'abstraction (ex. sur un objet `Pile`, on n'a accès qu'aux opérations légales pour le type abstrait de données) ;
— faire respecter certaines spécifications du problème (on pourrait ainsi compléter l'exemple précédent avec des classes `Enfant` et `RevendeurAlcool`, et un objet `Enfant` n'aurait pas accès à la méthode `vendre` d'un objet `RevendeurAlcool`).

Avec des conséquences importantes sur les qualités du logiciel :
— réduire la dépendance entre composants logiciels : les accès entre objets sont restreints ;
— faciliter la maintenance : la restriction des accès possibles limite les modifications à apporter en cas d'évolution ;
— faciliter la réutilisation : grâce aux qualités d'abstraction acquises.

10.2 Contrôle d'accès sur les propriétés

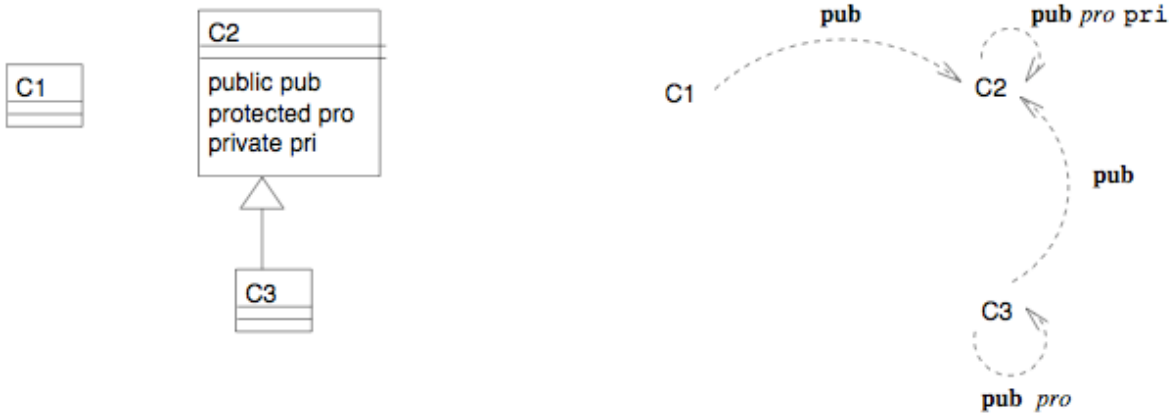


FIGURE 10.1 – Accès aux propriétés en dehors d'autres mécanismes. Une flèche de A vers B étiquetée m se lit "dans une méthode de A, on peut envoyer le message m à un objet de type statique B"

Sans imbrication avec un autre mécanisme, en première approximation :

— les propriétés `public:` sont accessibles depuis le code de n'importe quelle classe ou n'importe quelle fonction. Sur la figure 10.1, toute méthode de `C1,C2,C3` peut accéder à l'attribut `pub` sur un objet dont le type statique est la classe `C2` ;
— les propriétés `private:` ne sont accessibles que dans les autres méthodes de la même classe ;
— les propriétes `protected:` sont accessibles par une classe et ses sous-classes mais seulement sur leurs propres instances ; `C3` n'a pas accès à `pro` sur les objets de type statique `C2`.

Certains autres accès se rajouteront en cas d'héritage déclaré `public` comme nous le décrirons un peu plus loin.

10.3 La déclaration `friend`

Cette directive permet à une classe de donner à une autre classe, à une méthode d'une autre classe ou à une fonction le droit d'accéder à sa partie privée (ou protégée). Dans l'exemple donné ci-dessous, et pour aller du plus acceptable au plus inacceptable, on autorise l'opérateur <<, la méthode `vendre` de la classe `Voiture` et la fonction `main` à accéder à la partie privée de `Personne`.

```
class Personne ;

class Voiture
{
private:
Personne* proprietaire;
public:
Voiture(){}
virtual ~Voiture(){}
virtual void vendre(Personne* p);
};

class Personne
{
private:
string nom;
public:
Personne(){}
virtual ~Personne(){}
friend ostream& operator<<(ostream& os,const Personne& p);
friend int main();
friend  void Voiture::vendre(Personne* p);
};

void Voiture::vendre(Personne* p){cout << "vendue a " << p->nom << endl;
                                                              proprietaire=p;}

ostream& operator<<(ostream& os,const Personne& p)
{os << p.nom << endl;}

int main()
{
Personne p;
p.nom="juju";
cout << p;
}
```

Les limites de `friend` sont les suivantes :
— il faut anticiper (une fois l'interface de la classe écrite, on ne peut plus lui ajouter des amis) ;

— ce n'est pas hérité ;
— ce n'est pas partagé par les classes internes ;
— ce n'est pas symétrique ;
— ce n'est pas transitif.

On peut admettre son utilisation pour des cas restreints et anticipés d'accès, tels que l'écriture des opérateurs qui se fait de manière étroite avec celle de la classe concernée, mais il ne faut surtout pas généraliser son usage.

10.4 Contrôle d'accès sur les liens d'héritage

L'utilisation des mots-clefs `public`, `private` et `protected` s'étend à la clause dans laquelle une classe déclare ses super-classes. Par exemple, pour la figure 10.2, on déclare ainsi la classe `C3` :

```
class C3 : protected virtual C2 {};
```

Comme rien n'est précisé pour la classe `C4`, les accès dans les méthodes de `C3` vers les objets de la classe `C4` ne sont pas précisés.

Ces déclarations permettent à la sous-classe de restreindre, sur ses propres objets, l'accès aux propriétés héritées. Ainsi une propriété `public` héritée dans une sous-classe par un lien d'héritage `protected` devient `protected`.

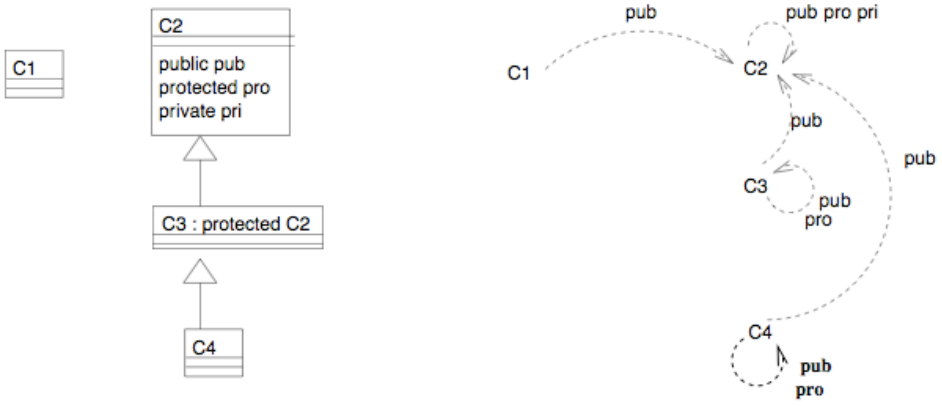


FIGURE 10.2 – Accès aux propriétés avec protected sur un lien d'héritage

La figure 10.3 présente l'intégralité des cas.

De plus, la conversion implicite (affectation polymorphe) devient impossible lorsqu'on restreint la visibilité du lien d'héritage.

```
class PersonnePro : protected virtual Personne{...};

int main()
{
Personne* p;
PersonnePro *pp=p;
}
```

Quand peut-on admettre de l'héritage privé ou protégé ? Par exemple lorsqu'on utilise l'héritage pour des besoins d'implémentation, mais qu'il n'y a pas de spécialisation entre les types abstraits correspondants. Dans le programme suivant, on montre une implémentation de Pile avec un vecteur, comme l'héritage est privé, les méthodes de vecteur ne s'appliquent pas à une pile.

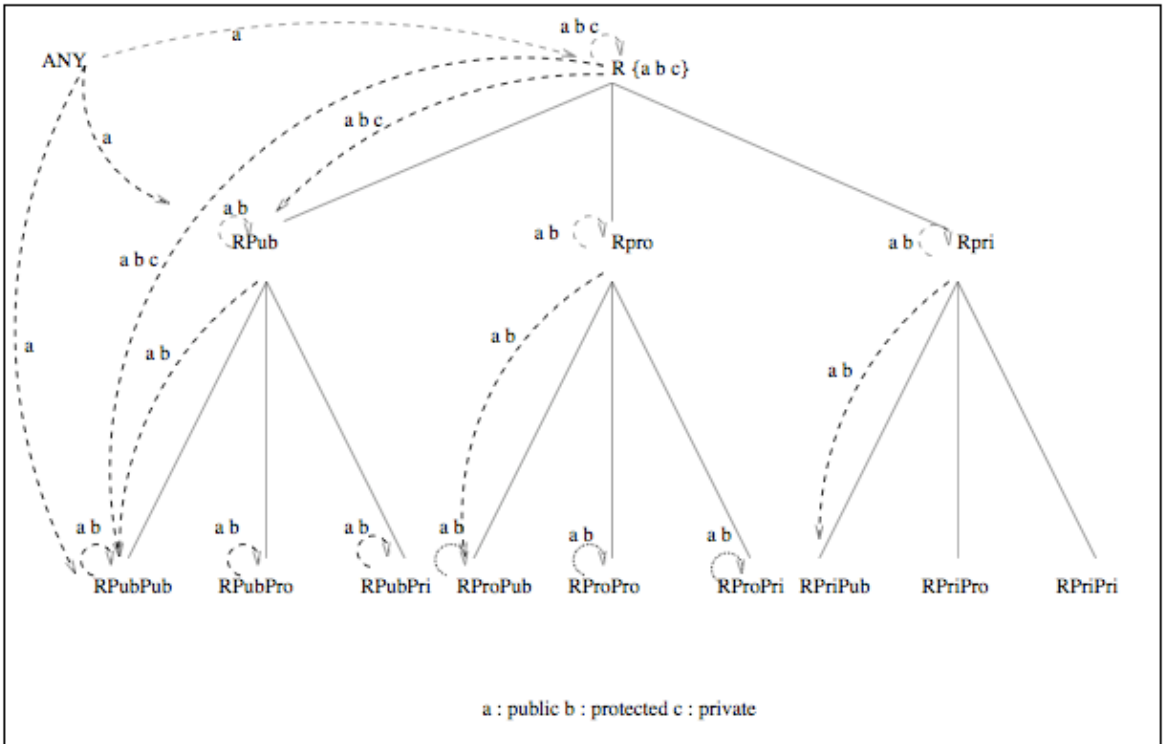


FIGURE 10.3 – Accès aux propriétés et protections sur liens d'héritage ; C_x est dérivée de C avec la protection x

```
template<typename T>
class Pile : private vector<T>
{
........
public:
.........
  virtual void empiler(T t);
};
.........
int main()
{
Pile<int> p;
p.empiler(2);
// (erreur : méthode d'une super-classe privée) p.size();
// (erreur : méthode d'une super-classe privée) p.push_back();
}
```

10.5 La déclaration `using`

On peut, sur une propriété héritée par un lien d'héritage portant une restriction, lever la restriction. Dans l'exemple de la pile, on peut réhabiliter certaines méthodes du vecteur, telle que celle qui procure le nombre d'éléments stockés.

```
template<typename T>
class Pile : private vector<T>
{
public:
```

```
  virtual void empiler(T t);
  using vector<T>::size;
};

int main()
{
Pile<int> p;
p.empiler(2);
cout << p.size() << endl; // maintenant possible
// (toujours erroné : méthode d'une super-classe privée) p.push_back();

}
```

10.6 Contrôles d'accès lors de la redéfinition

Comme le montre l'exemple ci-dessous, on peut modifier les accès dans les redéfinitions. Il n'y a pas de contraintes : f passe de privé à public et h passe de public à privé avec le comportement attendu. Le résultat du programme est le même si f est privé dans B. En même temps, vous constatez qu'il est utile de déclarer `virtual` même les méthodes privées.

```
class A
{
private:
 virtual void f(){cout << "f de A"<<endl;}
public:
 virtual void g(){this->f();}
 virtual void h(){cout << "h de A"; this->f();}
};

class B : public virtual A
{
public:
 virtual void f(){cout << "f de B"<<endl;}
private:
 virtual void h(){cout << "h de B"; this->f();}
};

class Bpro : protected virtual A
{
public:
 virtual void f(){cout << "f de B"<<endl;}
private:
 virtual void h(){cout << "h de B"; this->f();}
};

class Bpri : private virtual A
{
public:
 virtual void f(){cout << "f de B"<<endl;}
private:
 virtual void h(){cout << "h de B"; this->f();}
};

int main()
{
```

```
A *pa=new A();
cout << "cest pour A" << endl;
pa->g();
pa->h();

A *pb=new B();
cout << "cest pour B" << endl;
pb->g();
pb->h();

B *pbb=new B();
cout << "cest pour BB" << endl;
pbb->g();
//(accès interdit cf type statique) pbb->h();

Bpro *pbo=new Bpro();
cout << "cest pour Bpro" << endl;
pbo->f();

Bpri *pbbo=new Bpri();
cout << "cest pour BBpro" << endl;
pbbo->f();
}
```

10.7 Regard sur la modélisation

Après cette introduction sur les aspects techniques, il est important de se demander si tous ces mécanismes sont réellement utiles et expressifs. En ce qui concerne les contrôles d'accès simples pour la protection de l'implémentation d'un type abstrait de données on peut répondre de manière positive. Mais si on examine des spécifications, pourtant simples, issues de problèmes de modélisation, la réponse est moins évidente et curieusement, malgré l'abondance et la complexité de l'imbrication des mécanismes proposés, on est assez vite limité.

A titre d'exemple, nous proposons un exercice de réflexion basé sur une modélisation assez simple (figure 10.4). Il s'agit de trouver une manière de mettre en oeuvre les accès prévus dans le graphe 10.5 et le moins possible d'accès interdits (absents du graphe d'accès).

- la classe `Fournisseur` offre deux méthodes :
  - `fournir` est destinée aux revendeurs exclusivement pour obtenir des produits d'une certaine référence et dans une quantité donnée ;
  - `listeDeFournisseurs` est destinée aux personnes qui désireraient connaître les revendeurs agréés par le fournisseur.
- la classe `Revendeur` offre la méthode `vendre`. Cette méthode est destinée aux personnes sauf dans un cas particulier, les enfants ne doivent y avoir accès que sur la classe `RevendeurDeBonbons`.

Accès à la méthode `vendre` Les accès peuvent être mis en place sans accès superflu comme suit :

- `vendre` est `protected`,
- `Personne` est `friend` des trois classes de revendeur,
- `Enfant` est `friend` seulement de `RevendeurDeBonbon`,
- mais ce procédé laisse accès à toutes les autres méthodes et attributs des revendeurs s'il y en avait.

Accès aux méthodes `fournir` et `listeDeRevendeurs` Comme ces méthodes ne sont pas accessibles à toutes les classes, elles ne peuvent être publiques. Si elles sont privées ou protégées (moins approprié), il faut que les classes qui ont besoin d'y accéder soient `friend` mais cette fois cela permet par exemple à `Personne` d'accéder à `fournir`, ce qui n'est pas prévu dans la spécification.

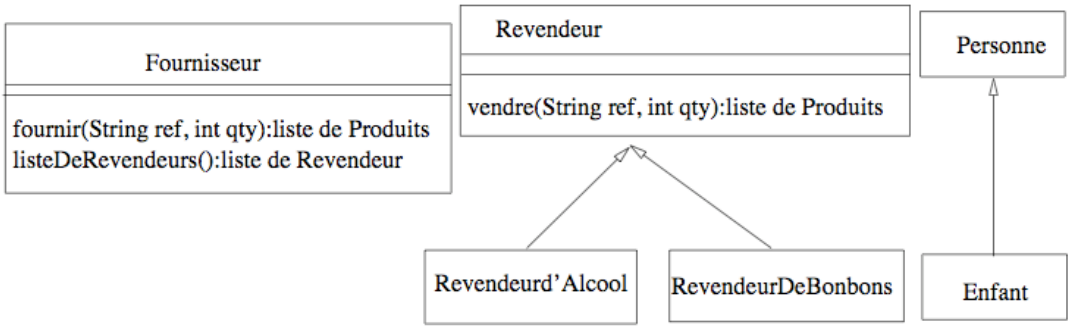


FIGURE 10.4 – Un diagramme de classes

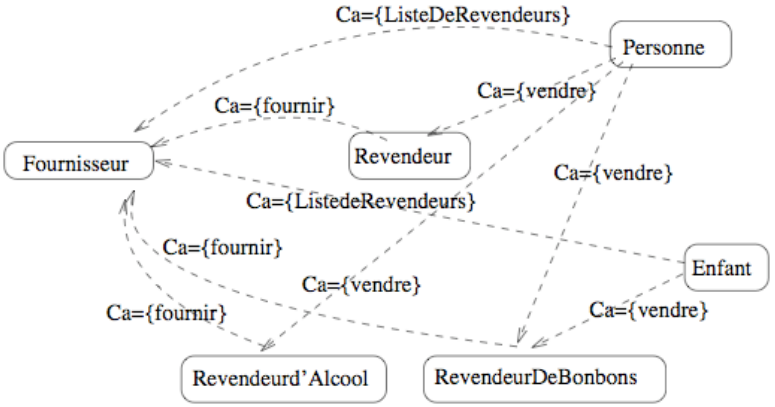


FIGURE 10.5 – Les accès réclamés

Synthèse Sur ce simple exemple il apparaît que malgré toute son artillerie, C++ ne permet pas de mettre en oeuvre facilement des spécifications d'accès.