Programmation (II)

Langage C++

Claude Guéganno
–

6 octobre 2004

Table des matières

1 Introduction **6**
- 1.1 Généralités . . . 6
- 1.2 Caractéristiques . . . 6
- 1.3 Structure d'un programme `C++` . . . 7
- 1.4 Compilation . . . 9
  - 1.4.1 Développement . . . 9
  - 1.4.2 Les étapes de la compilation . . . 9
  - 1.4.3 Exemple . . . 10

2 Les éléments de base **11**
- 2.1 Les identificateurs . . . 11
- 2.2 Les types de base . . . 11
  - 2.2.1 Les caractères (`char`) . . . 11
  - 2.2.2 Les entiers (`int`) . . . 12
  - 2.2.3 Les réels . . . 12
- 2.3 Les constantes . . . 13
  - 2.3.1 Les constantes entières . . . 13
  - 2.3.2 Les constantes réelles . . . 13
  - 2.3.3 Les constantes caractères . . . 14
  - 2.3.4 Les constantes chaînes de caractères . . . 14
  - 2.3.5 Les constantes booléennes . . . 15
- 2.4 Les variables . . . 15
  - 2.4.1 Déclaration d'une variable . . . 15
  - 2.4.2 Définition d'une variable . . . 16

3 Les expressions du `C++` **17**
- 3.1 Les opérateurs . . . 17
  - 3.1.1 Les opérateurs arithmétiques . . . 17
  - 3.1.2 Les opérateurs sur les *bits* . . . 18
  - 3.1.3 Les opérateurs d'affectations . . . 19
  - 3.1.4 Les opérateurs d'incrémentation et de décrémentation 20
  - 3.1.5 L'opérateur de taille . . . 20
  - 3.1.6 L'opérateur de séquence . . . 20
- 3.2 Les conversions . . . 21
- 3.3 Les opérateurs relationnels . . . 21

3.4 Les expressions booléennes 22
3.5 L'opérateur conditionnel 23
3.6 Analyse lexicale 23
3.7 Priorité et associativité des opérateurs 23

4 Entrée et sortie standards (`stdin`,`stdout` et `cin`,`cout`) 25
4.1 Écriture et lecture de caractères 26
4.1.1 Lecture de caractères : `getchar` 26
4.1.2 Écriture de caractères : `putchar` 26
4.1.3 Exemple avec `getchar` et `putchar` 26
4.1.4 Écriture de chaînes de caractères (`puts`) 27
4.1.5 Lecture de chaînes de caractères (`gets`) 27
4.2 Sortie formatée de caractères (`printf`) 27
4.2.1 Utilisation simple 28
4.2.2 Affichage de variables 28
4.2.3 Affichage d'une chaîne de caractères 29
4.3 Entrée formatée de caractères (`scanf`) 29
4.3.1 Lecture d'une variable simple 29
4.3.2 Lecture de plusieurs variables 30
4.4 Les flux d'entrées-sorties, `cin` et `cout` 31
4.4.1 Généralités 31
4.4.2 Mise en œuvre 31
4.4.3 Les manipulateurs 32

5 Les structures de commande 34
5.1 Les blocs d'instructions 34
5.2 Les répétitions 35
5.2.1 L'instruction `while` 35
5.2.2 L'instruction `do` . . .`while` 36
5.2.3 L'instruction `for` 37
5.3 Les choix 39
5.3.1 Le choix simple `if` . . .`else` 39
5.3.2 Le choix multiple `switch` . . .`case` 40

6 Les sous-programmes 42
6.1 Définition d'un sous-programme 42
6.2 Appel d'un sous-programme 43
6.3 Passage de paramètres par valeur 43
6.4 Passage de paramètres par référence 44
6.4.1 Notion de référence 44
6.4.2 Les références utilisées comme paramètres 45
6.5 Paramètres déclarés `const` 46
6.6 Les pointeurs comme paramètres 46
6.7 Les fonctions `inline` 47
6.8 Surcharge de fonctions 47
6.9 Les paramètres «par défaut» 48

7 Les tableaux **49**
7.1 Déclaration et définition d'un tableau 49
7.1.1 Déclaration 49
7.1.2 Accès aux membres 49
7.1.3 Exemple 50
7.1.4 Lecture d'un élément de tableau sur l'entrée standard 50
7.1.5 Tableau initialisé 50
7.2 Les tableaux de caractères 50
7.2.1 Chaînes de caractères 50
7.2.2 Fonctions d'entrées de chaînes de caractères 51
7.2.3 Opérations sur les chaînes de caractères 52
7.3 Les tableaux de dimension 2 53
7.4 Adresses et pointeurs 54
7.5 Allocation dynamique de mémoire 55

8 Les structures **57**
8.1 Définition 57
8.2 Déclaration et définition d'une variable structurée 58
8.3 Accès aux membres d'une structure 58
8.4 Structures imbriquées 58
8.5 Tableaux de structures 59
8.6 Références et pointeurs 60
8.6.1 Pointeurs sur structures 61
8.7 Code complet de l'exemple du chapitre 8 62

9 Les énumérations, les unions, les champs **64**
9.1 Les énumérations 64
9.2 Les unions 65
9.3 Les champs de *bits* 67

10 Programmation orientée objet **69**
10.1 Classes et objets 69
10.2 Notions d'invariant et d'interface 72
10.2.1 Spécification et mise en œuvre 72
10.2.2 Exemple : la classe `action` 72
10.3 Création d'un objet avec `new` 73
10.4 Déclaration et définition séparées 73
10.5 Les constructeurs 75
10.6 Surcharge des opérateurs 76
10.7 Redirection des flux standards `cin` et `cout` 77
10.7.1 Fonctions amies (`friend`) 77
10.7.2 Application à la redirection des flux 77
10.8 Objet et allocation dynamique de mémoire 78
10.9 Code complet de la classe action 79
10.10 Le constructeur par recopie 82
10.10.1 Classe sans le constructeur par recopie 82

10.10.2 Classe avec le constructeur par recopie 83

11 L'héritage 85
11.1 Principes 85
11.2 L'héritage public 85
11.3 Exemple : les classes `position` et `point` 87
11.4 Héritage et constructeurs/destructeurs 89
11.5 Les autres formes d'héritages 90
11.6 Le polymorphisme 91

12 Les listes chaînées 93
12.1 Limitation des représentations séquentielles 93
12.2 La liste chaînée 94
12.2.1 Idée de base : 94
12.2.2 Identification d'un objet 95
12.2.3 Chaînage 95
12.2.4 Quelques possibilités des listes chaînées 98
12.3 Exemple complet de mise en œuvre 99
12.3.1 Opérations sur les listes chaînées linéaires 101

13 Mise en œuvre des interfaces 104
13.1 Classe abstraite 104
13.1.1 Méthode virtuelle pure et classe abstraite 104
13.2 Fonctions «patrons» (*template*) 106
13.2.1 Objectifs 106
13.3 Classes «patrons» 107
13.3.1 Objectifs 107
13.3.2 Exemple 107

14 Les fichiers 111
14.1 Introduction aux fichiers 111
14.2 Mise en œuvre des fichiers séquentiels : ouverture et fermeture 111
14.3 Opérations sur les fichiers séquentiels 112
14.3.1 Lecture de fichiers textes 113
14.3.2 Ècriture de fichiers textes 113
14.3.3 Lecture et écriture de fichiers binaires 114
14.4 Les fichiers à accès sélectif 114
14.5 Les flux 115
14.5.1 Le flux `ostream` 116
14.5.2 Le flux `istream` 117
14.6 Application des flux aux fichiers 118
14.6.1 Association d'un fichier et d'un flux 118
14.6.2 Contrôle de l'état d'un flux 119
14.6.3 Formatage des données 119
14.6.4 Les manipulateurs 120

15 Les arguments de la ligne de commande **123**
15.1 Application 124

16 Les exceptions **126**
16.1 Généralités 126
16.2 Création d'exceptions 127

17 La `Standard Template Library` **129**
17.1 Généralités 129
17.2 Les conteneurs 129
17.2.1 Les vecteurs (`vector`) 129
17.2.2 Les listes (`list`) 130
17.2.3 Les *double end queues* (`deque`) 131
17.2.4 Les ensembles (`set`) 131
17.2.5 Les dictionnaires (`map`) 132
17.2.6 `multiset` et `multimap` 133
17.2.7 Autres conteneurs 133
17.3 Les itérateurs 133
17.4 Les algorithmes 134
17.4.1 *Non mutating algorithms* 134
17.4.2 *Sorting algorithms* 135
17.4.3 *Mutating algorithms* 135

A Codes ASCII **137**

Chapitre 1

Introduction

1.1 Généralités

Le langage `C` est un langage impératif, et polyvalent créé à partir de 1970 par Dennis Ritchie. En 1978, Dennis Ritchie et Brian Kerningham publient les spécifications définitives du langage. Le système `UNIX` à servi de banc d'essais pour le langage `C` . La réalisation d'un compilateur facilement «portable» a permis d'implanter le système `UNIX` sur de nombreuses machines.

Le `C` se caractérise par son aptitude à produire un code source concis, et à la diversité de son jeu d'instructions qui permet d'utiliser au mieux les possibilités usuelles des processeurs.

Le `C++` créé par Bjarne Stroustrup est une évolution du `C` . Tout en concervant les caractériqtiques du `C`, il apporte en plus **programmation orientée objet**.

Notons enfin que la syntaxe du `C` , universellement adoptée a également servi de base pour la conception des langages `JavaScript`, `Java` ...

L'utilisation du `C++` touche des domaines très variés :
- abstraction de données ;
- programmation «système» ;
- écriture de logiciels à usage général (bureautique, ...) ;
- infographie ;
- jeux ;
- applications scientifiques ;
- ...

1.2 Caractéristiques

→ C'est un **langage structuré**, conçu pour traiter les tâches d'un programme en les mettant dans des blocs.

→ Il est **déclaratif** : tout objet `C` ou `C++` doit être déclaré avant d'être utilisé.

→ Le **format** est **libre**. La mise en page est laissée à l'initiative du programmeur. Des éditeurs spécialisés tels `Emacs` tendent à rationnaliser et à uniformiser l'écriture du `C` .

→ Il est **modulaire** : une application peut être découpée en modules qui seront compilés séparément. La création de librairies dynamiques ou statiques est possible.

→ Il est **portable**, en ce qui concerne la partie standard.

→ Il est **orienté objet**.

1.3 Structure d'un programme C++

Voici un exemple de programme `C++` , dans deux versions différentes.

Première version qui n'utilise aucune des techniques de programmation avancée.

Remarque : `cout` représente l'écran (la sortie pour le programme) . Ainsi, la ligne

```
cout << "x = ";
```

permet d'afficher le message «`x = `». De la même manière, `cin` représente le clavier (l'entrée pour le programme) et la ligne

```
cin >> x;
```

permet d'initialiser la *variable* `x` avec une valeur numérique saisie au clavier.

```
/* Fichier: intro1.cpp
   créé le: 13 août 2002 - CG
*/

#include <iostream.h>

int main() {
  double x,y;
  int i;

  cout << "x = ";
  cin >> x;

  y =  2*x*x -3*x + 2;

  cout << "f(" << x << ") = " << y <<  '\n';
  return 1;
}
```

Deuxième version qui utilise quelques particularités du langage : les commentaires, la définition de constantes, l'utilisation d'une fonction.

```
/* Fichier: intro2.cpp
   créé le: 13 août 2002 - CG
*/

#include <iostream.h>
#include <math.h>

// constante :
const int A = 2;
const int B = -3;
const int C = 2;

/*
  Fonction : Calcule
  Entrée : X (double)
  Sortie :  A*X*X + B*X + C (double)
*/
double Calcule(double X) {
  double R;
  R =  A*X*X + B*X + C;
  return R;
}


// ----------------- Fonction principale -----------------
int main() {
  double x,y;
  int i;

  cout << "x = ";
  cin >> x;

  y = Calcule(x); // calcul de Ax^2 + Bx + C

  cout << "f(" << x << ") = " << y <<  '\n';
  return 1;
}
```

On y trouve :

→ des **commentaires** : il s'agit de tout ce qui est enfermé entre les séquences de caractères `/*` et `*/`, le nombre de lignes de commentaire est alors quelconque ; ou bien, tout ce qui suit la séquence de caractères `//` : dans ce cas le commentaire s'arrête à la fin de la ligne.

→ des **directives** de compilation : ce sont des lignes de programme commençant par le caractère `#`. Dans l'exemple, on trouve des directives `#include` qui correspondent à des inclusions de fichiers de définitions. On trouve souvent la directive `#define` utilisée pour la définition de

constantes. Cette directive tend à être remplacée par la définition typée de constante : `const <type>`. Ainsi, à la place de

```
const double A = 2
```

Nous aurions pu écrire :

```
#define A 2
```

→ des **déclarations** : la déclaration d'un objet `C` donne simplement ses caractéristiques au compilateur et ne génère aucun code.
`int i ;` est la déclaration d'un entier,
`float Calcule(double X)` est la déclaration d'une fonction qui reçoit un réel et qui renvoie un réel.

→ des **fonctions** : ce sont des sous-programmes qui définissent des instructions opérant sur des variables.

1.4 Compilation

1.4.1 Développement

La fabrication d'un programme se fait en plusieurs étapes.

1. Conception du fichier «source» avec un éditeur de textes.
2. Compilation du programme.
3. À ce niveau, il peut y avoir des erreurs. S'il s'agit d'erreurs de syntaxe, il faut rectifier en reprenant à l'étape 1. S'il s'agit de fonctions non trouvées par le compilateur, il faut sans doute modifier la ligne de commande de compilation afin de le lier à une ou plusieurs librairies supplémentaires.
4. On peut enfin, tester le programme. On peut à ce moment détecter les fautes de logique ...

1.4.2 Les étapes de la compilation

On distingue (voir figure 1.1) :

le préprocesseur : il permet de fusionner en un seul fichier le fichier source et l'ensemble des fichiers *headers* inclus. Les substitutions liées aux constantes (`#define`) sont prises en compte. D'une manière générale, le préprocesseur interprète les directives de compilation.

le compilateur : le fichier `C++` intermédiaire est traduit en assembleur. Les appels aux fonctions non présentes sont laissés en blanc.

l'assembleur : le fichier assembleur créé est traduit en langage binaire. Les appels aux fonctions extérieures ne sont toujours pas pris en compte.

l'éditeur de lien : les fonctions non présentes sont recherchées dans les librairies et associées aux fichiers binaires (ou objets) pour former un seul fichier exécutable.

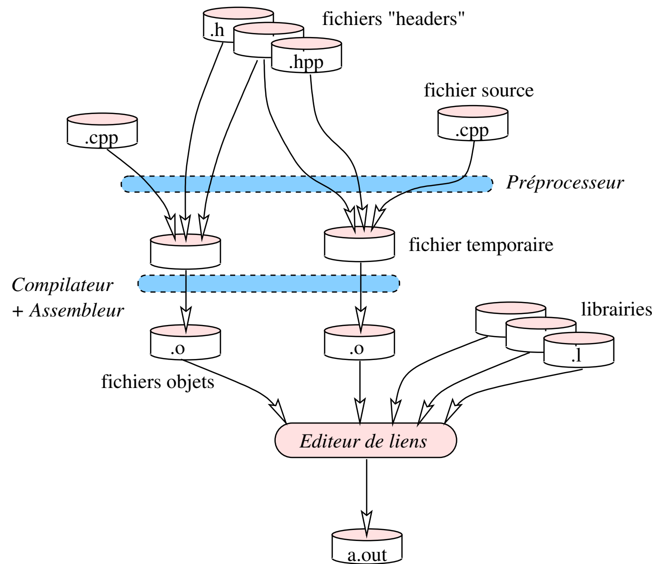


FIG. 1.1 – *Du fichier «source» à l'exécutable.*

1.4.3 Exemple

Si le compilateur utilisé est `g++` (GNU `C++` Compiler),

→ la commande

```
g++ test.cpp
```

produit le fichier exécutable `a.out`

→ la commande

```
g++ -o test test.cpp
```

produit le fichier exécutable `test`

→ la commande

```
g++ -lm -o test test.cpp
```

produit un fichier exécutable `test` et lie le programme avec la librairie mathématique.

Pour plus d'information sur le compilateur `g++`, il faut utiliser la commande `man g++`.

Chapitre 2

Les éléments de base

2.1 Les identificateurs

Ils servent à nommer les variables, les constantes et les fonctions.
- Un identificateur est une suite de lettres ou de chiffres.
- Le premier caractère est obligatoirement une lettre.
- Le caractère *souligné* (`_`) est considéré comme une lettre.
- Les minuscules et les majuscules sont des lettres distinctes. Ainsi,

  `Mesure`, `MESURE` et `mesure`

  correspondent à des objets différents.
- La longueur de l'identificateur dépend de l'implémentation. La norme ANSI prévoit que les 31 premiers caractères sont significatifs.
- Un identificateur ne peut pas être un mot réservé du langage :

```
auto, break, case, char, const, continue, default,
do, double, else, enum, extern, float, for, goto,
if, int, long, register, return, short, signed,
sizeof, static, struct, switch, typedef, union,
unsigned, void, volatile, while, bool, class,
template, try, catch, new, delete, cin, cout,
operator, friend, this, using
```

2.2 Les types de base

Les types de base nous serviront plus tard à composer les types dérivés (tableaux, structures, unions, classes). On les regroupe en quatre catégories :
- booléennes,
- caractères,
- entiers,
- réels.

2.2.1 Les caractères (`char`)

Le type `char` représente un caractère parmi dans le jeu de caractères de la machine. Le langage ne défini pas le code de caractères utilisé. En général,

il s'agit du code ASCII.

Caractéristiques :

Taille : 1 octet

Intervalle : de -128 à +127 ou de 0 à 255 suivant les machines.

Il peut être :

- `signed char` : intervalle de -128 à +127
- `unsigned char` : intervalle de 0 à 255

Les modificateurs `signed` et `unsigned` forcent l'arithmétique en version signée ou non signée. Ils rendent le code plus portable.

2.2.2 Les entiers (`int`)

Le type `int` permet de représenter les nombres entiers. Il correspond à un mot machine[1]. Sa taille dépend donc de la machine hôte. Elle est en général de 16 ou 32 *bits*.

Il peut être qualifié par les modificateurs :

- `short` : la taille de l'entier est alors de 2 octets,
- `long` : la taille d'un `long` est deux fois celle de l'`int`,
- `signed` : l'entier est signé (c'est le cas par défaut)
- `unsigned` : l'entier est non signé.

`int` est facultatif dès qu'il est précédé de `short`, `long` ou `unsigned`.

Exemples de déclarations

```
int i;
signed int i;

unsigned int j;
unsigned j;

unsigned short int k;

long int m;
long m;
```

Dans ces exemples les déclarations groupées sont équivalentes.

2.2.3 Les réels

Ils permettent de représenter les nombres en virgule flottante.

On distingue les types,

- `float` : pour les nombres en virgule flottante codés en simple précision (codage sur 4 octets)

[1] C'est à dire ce qui peut être contenu dans un registre de données du processeur

- `double` : pour les nombres en virgule flottante à double précision (codage sur 8 octets)
- `long double` pour les nombres à précision étendue.

2.3 Les constantes

Les constantes sont définies par une valeur et un type. On peut donner un nom symbolique à une constante en utilisant le mot clé `const`

Exemple : `const double Pi = 3.14 ;`

2.3.1 Les constantes entières

Les constantes entières peuvent être exprimées par l'une des trois formes suivantes :

- **décimale,** exemple : `1234`
- **octale,** le premier chiffre est alors un `0`. Le codage en octal est peu utilisé, mais il faut se méfier des entiers commençant par un `0` ; ils seront interprétés comme des nombres octaux. Exemple : `0777`, `0123`.
- **hexadécimal,** le nombre commence par `0x` ou `0X`. On peut utiliser indifféremment les lettres majuscules ou minuscules. Exemple : `0xff`, `0x12AE`.

Les entiers trop grands pour être codés par un `int` seront considérés comme appartenant au type `long` .

On peut imposer à une constante d'être du type

- `long` en lui ajoutant la lettre `L`[2].
  Exemple : `1234L`
- `unsigned` en lui ajoutant la lettre `U`.

2.3.2 Les constantes réelles

Les constantes réelles sont par défaut du type `double` . Elles peuvent être exprimées sous les formes suivantes :

- **décimale,** exemple : `123.4`, `3.14`
- **scientifique,** exemple : `6.02217e23`, `1.6e-19` correspondant respectivement à 6.02217×10^{23} et 1.6×10^{-19}.

Elles peuvent aussi contenir un suffixe :

- `F` ou `f` impose le type `float` à la constante,
  exemple `1.89f`.
- `L` ou `l` impose le type `double` .
  exemple `3.901L`.

[2] On peut également utiliser le `l` minuscule, mais la confusion est possible avec le chiffre `1`.

Notation	Code ASCII	Signification
'\n'	0x0A	saut de ligne
'\r'	0x09	tabulation horizontale
'\v'	0x0B	tabulation verticale
'\a'	0x07	*beep*
'\b'	0x08	retour arrière
'\r'	0x0D	retour chariot
'\f'	0x0C	saut de page
'\\'	0x5C	*backslash*
'\''	0x2C	simple quote
'\?'	0x3F	point d'interrogation
'\0'	0x00	caractère nul (`NULL`)
'\"'	0x22	double quote

TAB. 2.1 – Représentation symbolique des caractères

2.3.3 Les constantes caractères

Une constante *caractère* est écrite entre apostrophes. Elle peut être exprimée sous l'une des formes suivantes :
- normale : `'a', 'U', '1', '|'`
- octale : `'\033', '\077'`
- hexadécimale :`'\x1b'`
- symbolique (voir tableau 2.3.3

Dans les représentations décimale et hexadécimale, le caractère est représenté par son code ASCII.

La valeur d'une constante caractère est égale à la valeur numérique du caractère dans le code de caractères de la machine (souvent le code ASCII).

2.3.4 Les constantes chaînes de caractères

Elles sont constituées d'une séquence de caractères, et délimitées par le caractère `"` (double quotes). Toutes les notations de caractères (normale, octale, hexadécimale, ou symbolique) sont utilisables dans les chaînes.

Exemples : `"Le langage \'C\'"`
`"Au revoir\n"`
`"\x1b*"`

Remarques :

† Un caractère nul (`'\0'`) est ajoutè automatiquement à la fin de la chaîne. Il permet aux fonctions sur les chaînes d'en détecter la fin.

† Les constantes *chaînes de caractères* sont stockées en zone de mémoire permanente.

† Elles sont considérées comme des tableaux de caractères de dimension 1 et de taille égale à la longueur de la chaîne plus 1 pour tenir compte du caractère `NULL` placé à la fin.

† Une chaîne peut tenir sur plusieurs lignes.

Exemples :

```
"Voici une chaîne"
"sur deux lignes ..."

"En voici\
   une autre"
```

Il n'existe pas d'opérateurs sur les chaînes de caractères, mais il existe une librairie de fonctions de manipulations de chaînes qui permet d'effectuer toutes les opérations courantes (voir les fonctions de `string.h`).

2.3.5 Les constantes booléennes

Ce sont des constantes de type `bool`. Il n'y a que deux valeurs possibles : `true` et `false`. Exemple :

```
const bool VRAI = true;
```

2.4 Les variables

2.4.1 Déclaration d'une variable

En `C` , les variables doivent obligatoirement être déclarées avant d'être utilisées. La syntaxe générale d'une déclaration de variable est de la forme :

```
[classe] [modificateur] type identificateur ;
```

`[classe]` et `[modificateur]` sont facultatifs. Ces différents champs sont inventoriés dans le tableau 2.4.1

classe	modificateur	type
auto extern register static	const volatile	char, short int, long float, double ...

Tab. 2.2 – Déclaration d'une variables

Les classes d'allocation

La classe `auto` est implicite lorsqu'il s'agit d'une variable locale à une fonction ou à un bloc d'instructions. La variable n'a donc d'existence que dans ce bloc. En général, ce mot est omis dans les déclarations.

La classe `extern` est utilisée dans le cas ou le programme est écrit en plusieurs fichiers distincts[3]. Une variable dite `extern` dans un fichier doit avoir été déclarée et définie dans un autre fichier. D'une certaine manière, ceci permet de «partager» des variables entre des fonctions écrites dans des fichiers différents.

La classe `register` indique que la variable doit être contenue dans un registre du microprocesseur, l'objectif étant d'augmenter la vitesse d'exécution du programme. L'affectation d'un registre à la variable n'est cependant pas systématique. Elle dépend de la disponibilité des registres et du nombre de variables candidates.

La classe `static` permet à une variable locale à une fonction de conserver sa valeur courante entre deux appels consécutifs.

Les modificateurs

Le modificateur `const` indique une constante. Il ne sera pas possible de la redéfinir en cours d'exécution du programme.

Le modificateur `volatile` indique au compilateur que la variable pourra être modifiée à des instants indéterminés. C'est le cas par exemple pour une variable exploitée par un programme et pouvant être modifiée à tous moments apar un programme d'interruption.

2.4.2 Définition d'une variable

Pour les variables simples, il suffit d'affecter une valeur.

Exemples :

```
const int Dimension = 3;
static float Rayon = 3.66;
int i=j=k=0;
```

Initialisations implicites. Pour les classes `extern` ou `static`, les variables sont initialisées à 0 en l'absence d'initialisation explicite.

Les variables `auto` ou `register` contiennent une valeur indéfinie.

[3] On parle alors de compilation séparée.

Chapitre 3

Les expressions du C++

3.1 Les opérateurs

Les opérateurs suivants agissent sur tous les types ***réel*** ou ***entier*** , à l'exeption de l'opérateur % qui ne prend que des opérandes ***entier*** .

3.1.1 Les opérateurs arithmétiques

Opérateur	**Opération**
-	moins unaire
+	plus unaire
*	multiplication
/	division
+	addition
-	soustraction
%	*modulo* (reste de la division entière)

TAB. 3.1 – Les opérateurs arithmétiques.

Remarques :

† L'opérateur % ne s'applique qu'aux entiers. Si l'un des opérateurs est négatif, le signe du résultat dépend de l'implémentation du compilateur. En général, il est du même signe que le dividende.

† Il n'existe pas d'opérateurs d'exponentiation.

† La division (/) est une division entière lorsque les deux opérandes sont entiers.

Ainsi, `5/2` donnera `2` pour résultat alors que `5./2.` donnera `2.5`.

3.1.2 Les opérateurs sur les *bits*

Les opérateurs suivants opérent *bit* à *bit* et s'appliquent à des opérandes de type ***entier*** (ou ***caractère***) et de préférence `unsigned` (sinon, ils risquent de modifier le *bit* de signe.

Ils permettent d'accéder à des opérations généralement réservées au langage assembleur. De ce point de vue, le compilateur `C++` utilise plus d'instructions du microprocesseur que la plupart des autres langages.

Opérateur	**Opération**
`~`	négation *bit* à *bit* (unaire)
`<<`	décalage à gauche
`>>`	décalage à droite
`&`	et *bit* à *bit*
`^`	ou exclusif *bit* à *bit*
`|`	ou inclusif *bit* à *bit*

TAB. 3.2 – Les opérateurs sur les *bits*.

négation *bit* **à** *bit* (`~`) c'est un opérateur unaire. Il inverse un à un tous les bits de son opérande.

Exemple : `~0x5f` est une expression qui vaut `0xa0` (soit `160`). En effet,

```
 0x5f  →  0101 1111
~0x5f  →  1010 0000  →  0xa0
```

décalage à gauche (`<<`) `n << i` décale l'entier `n` de `i` *bits* vers la gauche. Les *bits* sortants à gauche sont perdus et des `0` sont introduits à droite. Cet opérateur permet d'effectuer des multiplications d'entiers par des puissances de 2. Si la variable `n` est signée, le bit de signe est conservé.

décalage à droite (`>>`) `n >> i` décale l'entier `n` de `i` *bits* vers la droite. Les *bits* sortants à droite sont perdus et des `0` sont introduits à gauche.Cet opérateur permet d'effectuer des divisions d'entiers par des puissances de 2. Si la variable `n` est signée, le bit de signe est conservé et propagé.

et *bit* **à** *bit* (`&`) un «et» est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 & 0x53` vaut `0x12` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         0001 0010  →  0x12
```

ou exclusif *bit* **à** *bit* (`^`) un «ou exclusif» est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 ^ 0x53` vaut `0xe5` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         1110 0101  →  0xe5
```

ou inclusif *bit* **à** *bit* (`|`) un «ou» inclusif est effectué *bit* à *bit* sur les deux opérandes.

Exemple : `0xb6 | 0x53` vaut `0xf7` :

```
0xb6  →  1011 0110
0x53  →  0101 0011
         ---------
         1111 0111  → 0xf7
```

3.1.3 Les opérateurs d'affectations

Notion de *lvalue*

Une *lvalue* est une expression qui peut apparaître à gauche de l'opérateur d'affectation. Une *lvalue* possède une adresse en mémoire.

Exemple : dans l'expression `i = 7 ;` la *lvalue* est la variable entière `i`.

L'opérateur d'affectation =

En `C++` , l'affectation est une opération comme une autre. L'objet à gauche de l'opérateur d'affectation `=` (la *lvalue*) se voit affecter la caleur retournée par l'expression de droite.

Associativité. On peut écrire :

```
int i, j;
i = j = 0;
```

L'instruction `i = j = 0;` est équivalente à `i = (j = 0);`

L'expression `(j = 0)` vaut 0. Elle peut être regardée de deux points de vue :

1. la variable `j` se voit affecter la valeur 0,
2. c'est une expression de valeur 0

L'affectation combinée

Il s'agit de la traduction en `C` des instructions de «lecture-modification-écriture» souvent présentes dans les jeux d'instructions des micro-processeurs.

La syntaxe générale est

lvalue `op= expression ;`

avec `op` ∈ { `*, /, %, +, -, <<, >>, &, ^, |` }

`X op= Y ;` est équivalent à `X = X op Y ;`

Exemple : `k += 2 ;` est équivalent à `k = k + 2 ;`

3.1.4 Les opérateurs d'incrémentation et de décrémentation

Dans un programme, il est fréquent de devoir incrémenter ou décrémenter une variable. Le `C` propose deux opérateurs unaires pour effectuer ces deux opérations.

`++` incrémentation de 1

`--` décrémentation de 1

Pré-incrémentation et pré-décrémentation

Les opérateurs d'incrémentation [de décrémentation] sont placés devant la variable. L'incrémentation [la décrémentation] est effectuée puis l'utilisation de la variable est faite.

Exemple : `a = ++b ;` $\Leftrightarrow \begin{cases} \texttt{b = b+1 ;} \\ \texttt{a = b ;} \end{cases}$

Post-incrémentation et post-décrémentation

Les opérateurs d'incrémentation [de décrémentation] sont placés après la variable. L'utilisation de la variable est effectuée avant l'incrémentation [la décrémentation]

Exemple : `a = b++ ;` $\Leftrightarrow \begin{cases} \texttt{a = b ;} \\ \texttt{b = b+1 ;} \end{cases}$

La valeur de l'expression `b++ ;` est la valeur de `b` avant l'incrémentation.

3.1.5 L'opérateur de taille

L'opérateur `sizeof` est utilisé pour déterminer la taille (en octets) d'une variable ou d'un type.

Syntaxes : `sizeof(variable) ;` ou `sizeof(type) ;`

- L'opérateur `sizeof` appliqué à une chaîne de caractères retourne sa taille, y compris le caractère `NULL` de fin de chaîne.
- L'opérateur `sizeof` appliqué à un tableau retourne la taille en octets pour stocker celui-ci.

3.1.6 L'opérateur de séquence

L'opérateur `,` permet d'évaluer différentes expressions dans l'ordre de leur écriture. La valeur retournée est la valeur de la dernière expression évaluée.

Exemple : `a = (b=2 ; b+3) ;`$\Leftrightarrow \begin{cases} \texttt{b = 2 ;} \\ \texttt{a = b+3 ;} \end{cases}$

3.2 Les conversions

Le langage `C++` possède ses règles de conversion de type de données à l'intérieur d'une expression. Le but en est d'obtenir une meilleure précision du résultat. Un certain nombre de conversions sont implicitement appliquées :

- les opérandes de type `char` , `unsigned char` et `short` sont convertis en `int` ,
- si un opérande est un `long double` , l'autre est converti en `long double` ,
- si un opérande est un `double` , l'autre est converti en `double` ,
- si un opérande est un `float` , l'autre est converti en `float` .

Des promotions sont effectuées sur les opérandes d'après les règles suivantes (prises dans l'ordre) :

1. si un opérande est un `unsigned long` , l'autre est converti en `unsigned long` ,
2. si un opérande est un `long` , et l'autre est un `unsigned` , les deux opérandes sont convertis en `unsigned long` ,
3. si un opérande est un `long` , l'autre est converti en `long` ,
4. si un opérande est un `unsigned` , l'autre est converti en `unsigned`
5. sinon le résultat est un `int`

Conversions explicites (*casting*)

On peut explicitement demander une conversion dans un type désiré. Cette opération s'appelle le «transtypage» ou *casting*. Une expression précédée d'un nom de type mis entre parenthèses est convertie dans le type désiré.

Exemple :

```
float x = 6.022;
int i;
i = (int)x;
```

L'expression `(int)x` représente la traduction de `x` en entier. La valeur de `x` n'est pas affectée et `i` prend pour valeur 6.

Le *casting* est souvent utilisé pour s'assurer que les paramètres d'une fonction sont de type correct. Il est conseillé de faire appel à cet opérateur plutôt que de spéculer sur les conversions implicites de type.

3.3 Les opérateurs relationnels

Le résultat de la comparaison entre deux expressions est de type `bool` et vaut :

`false` si le résultat de la comparaison est faux,

`true` si le résultat de la comparaison est vrai.

Opérateur	Opération
`==`	test si égalité
`!=`	test si différent
`<`	test si inférieur
`<=`	test si inférieur ou égal
`>`	test si supérieur
`>=`	test si supérieur ou égal

Tab. 3.3 – Les opérateurs relationnels.

Exemple :

`bool estPair = (n % 2 == 0) ;` `estPair` vaut `true` si `n` est pair et `false` sinon.

`if (k != 100)` teste si `k` est différent de 100

3.4 Les expressions booléennes

C'est une expression dont l'évaluation est de type `bool`.

Les opérateurs logiques

Les trois opérateurs logiques ***ou*** ***et*** ***non*** sont prévus en `C` .

Opérateur	Opération
`&&`	réalise le ***et*** logique
`||`	réalise le ***ou*** logique
`!`	réalise le ***non*** logique (opérateur unaire)

Tab. 3.4 – Les opérateurs booléens.

Exemple : l'expression $10 \leq x < 100$ se traduira en `C` par :

```
10<=x && x<100
```

L'évaluation des expresssions booléennes s'effectue de gauche à droite (sauf pour l'opérateur `!`) et elle est stoppée dès que le résultat de l'expression devient définitif (*short evaluation*).

3.5 L'opérateur conditionnel

Il permet d'écrire des expressions dont le résultat est fonction de certaines conditions.

Syntaxe : `(expression) ? expression1 : expression2 ;`
Si `expression` est *Vrai* , le résultat est `expression1`, sinon, le résultat est`expression2`.

Exemple : cet opérateur permet de réaliser simplement la fonction «valeur absolue» :

```
b = (a<0) ? -a : a;
```

3.6 Analyse lexicale

L'analyseur lexical du compilateur `C` extrait les unités syntaxiques de l'instruction de gauche à droite en prenant à chaque fois l'unité syntaxique la plus longue .

Ainsi, l'expression `a+++b` est évaluée en `a++ + b` en non en `a + ++b`.

Cet exemple à lui seul montre l'importance d'écrire des programmes clairs et sans ambiguité pour le lecteur.

3.7 Priorité et associativité des opérateurs

La priorité et l'associativité des opérateurs du langage `C` sont données dans la table 3.5. Les opérateurs de priorités 1 sont les opérateurs de plus forte priorité.

Remarques :

† Les parenthèses permettent d'outrepasser les règles de priorité en forçant l'évaluation des expressions qu'elles contiennent. Elles permettent aussi une meilleure lisibilité des expressions.

† En cas d'opérateurs de même priorité, c'est l'associativité de l'opérateur qui détermine le sens d'évaluation de l'expression .

Exemple : L'opérateur `+` étant associatif de gauche à droite, l'expression `i + j + k` est évaluée comme `(i+j) + k`.

Priorité	Opérateur	Symbole	Associativité
1	fonction	`()`	→
	tableau	`[]`	→
	champ de structure	`->`	→
2	négation booléenne	`!`	←
	négation *bit* à *bit*	`~`	←
	incrémentation	`++`	←
	décrémentation	`--`	←
	`-` unaire	`-`	←
	`+` unaire	`+`	←
	indirection	`*`	←
	adresse	`&`	←
	taille	`sizeof`	←
3	*casting*	`(type)`	←
4	multiplication	`*`	→
	division	`/`	→
	modulo	`%`	→
5	addition	`+`	→
	soustraction	`-`	→
6	décalages	`<<  >>`	→
7	relations logiques	`< <= >= >`	→
8	égalité	`== !=`	→
9	***et*** *bit* à *bit*	`&`	→
10	***ou*** exclusif *bit* à *bit*	`^`	→
11	***ou*** *bit* à *bit*	`|`	→
12	***et*** logique	`&&`	→
13	***ou*** logique	`||`	→
14	opérateur conditionnel	`? : ;`	←
15	affectations	`= *= /= %=`	←
		`+= -= <<= >>=`	←
		`&= ^= |=`	←
16	séquence	`,`	→

TAB. 3.5 – Précédence des opérateurs.

Chapitre 4

Entrée et sortie standards (`stdin`,`stdout` et `cin`,`cout`)

Pour réaliser les opérations élémentaires d'écriture sur l'écran ou de lecture des informations du clavier, le `C` utilise un ensemble de fonctions. Il est utile de maîtriser l'usage d'au moins 6 d'entre elles : `getchar`,`putchar`, `scanf`, `printf`, `gets` et `puts`. Elles sont déclarées dans le fichier `stdout.h`

En `C++` il est possible d'utiliser les flux (ou flots) d'entrées sorties `cin` et `cout`, plus simples à mettre en œuvre. Ils sont présentés en fin de chapitre.

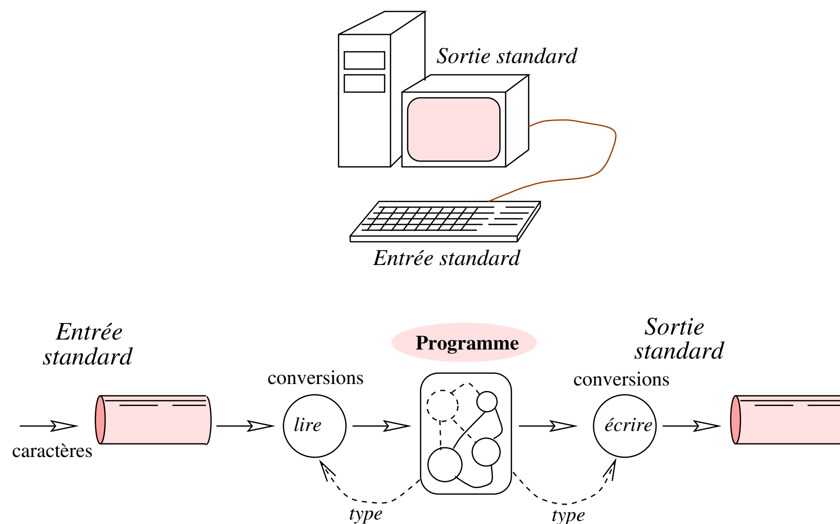


FIG. 4.1 – *Entrée et sortie standards.*

4.1 Écriture et lecture de caractères

4.1.1 Lecture de caractères : `getchar`

La fonction **getchar** a pour objectif la lecture de caractères isolés entrés au clavier. Les caractères entrés au clavier passent par un *buffer*, c'est à dire une zone tampon de stockage provisoire. En général, les caractères ne sont «envoyés» au programme par le terminal que lorsque la touche [Entrée] a été pressée.

Prototype : `int getchar()`
La fonction renvoie un entier qui est en réalité le code du caractère lu (`unsigned char`) converti en `int` .

Exemple :

```
unsigned char c;
c = getchar();
```

4.1.2 Écriture de caractères : `putchar`

Le rôle de la fonction **putchar** est d'écrire sur l'écran les caractères isolés. Á la différence de la fonction **getchar**, les caractères sont pris en compte immédiatement par le périphérique.

Prototype : `int putchar(int c)` La fonction renvoie un entier qui est le code du caractère écrit converti en `int` . Le caractère passé en argument est également converti implicitement en `int` .

Exemple :

```
putchar('H');
putchar('\n');
```

4.1.3 Exemple avec `getchar` et `putchar`

Le programme suivant attend l'appui sur une touche, validé par la touche [Entrée]. Le caractère lu est affiché après avoir été transformé en majuscule. La transformation en majuscule est réalisée par la fonction de la librairie standard **toupper**.

```
#include <stdio.h>

int main() {
  unsigned char c;
  c = getchar();
  getchar(); /* pour prendre le caractère "Entrée" */
```

```
  putchar(toupper(c));
  putchar('\n');
  return 1;
}
```

4.1.4 Écriture de chaînes de caractères (puts)

La fonction putchar n'est pas performante pour écrire un long message à l'écran. Ceci est le rôle de la fonction puts. Elle reçoit en entrée une chaîne de caractères. L'effet produit est l'affichade de cette chaîne.

Prototype : `int puts(const char *s) ;`
En réalité, la fonction prend en entrée l'adresse de la chaîne à afficher. Cette chaîne ne sera pas modifiée par la fonction. Elle renvoie un nombre non nul si la chaîne a été affichée avec succès ou 0 dans le cas contraire.

Exemple :

```
puts("Hello, world !\n");
```

4.1.5 Lecture de chaînes de caractères (gets)

La fonction gets permet de saisir un message au clavier, terminé par la touche Entrée. L'utilisation de gets suppose que dans le programme on dispose d'une variable permettant de stocker l'ensemble du message. La chaîne lue est en général stockée dans un tableau de caractères. La fonction gets ajoute les caractères lus dans le tableau sans vérifier que le tableau est suffisemment grand pour les contenir tous. Son usage met donc en péril la sécurité du système. Nous ne l'utiliserons que dans nos premiers programmes à cause de sa simplicité de mise en œuvre.

Prototype : `char *gets(char *s) ;`
gets reçoit l'adresse du tableau de caractères à initialiser. Elle renvoie cette même adresse.

Exemple :

```
char b[1024]; /* tableau assez grand (peut-être ...) */
puts("Entrez votre message :");
gets(b);
puts("\nVous avez entré :");
puts(b);
```

printf

4.2 Sortie formatée de caractères (printf)

Les fonctions putchar et puts sont faciles d'utilisation, mais elle ne permettent pas d'afficher directement un entier ou un réel ...
La fonction printf permet de réaliser en un seul appel :

1. la transformation des variables numériques en chaînes affichables,
2. le formatage des données (pour la présentation),
3. l'affichage.

4.2.1 Utilisation simple

On peut utiliser `printf` comme la fonction `puts`. Il n'y a pas de formatage à faire.

Exemple :

```
printf("Que dire ?\n");
```

4.2.2 Affichage de variables

Les variables du programme peuvent être affichées par `printf`

Exemple :

```
int i=2, j=3;
float x = 3.14;
char c = 'U';
printf("i = %d ", i);
printf("j=%d  x=%f  c=%c\n",j,x,c);
```

Ce programme affiche : `i = 2 j=3 x=3.140000 c=U`

Un «format» de `printf` est un groupe de caractères de la chaîne à afficher qui commence par le caractère `%`. Il renvoie à une variable passée également comme argument à `printf` . Le premier format correspond à la première variable, le second format à la seconde variable ...

Il doit y avoir cohérence entre le type de la variable à afficher et le format qu'on lui associe.

Dans l'exemple précédent, les formats sont les groupes `%d`, `%f` et `%c`. Ils correspondent aux variables respectives `j`, `x` et `c`

Des caractères supplémentaires sont utilisés pour mieux préciser le format d'affichage :

Exemple :

```
void main() {
  float x=123.4567;
  printf("%f  %1.2f  %1.4f\n",x,x,x);
}
```

Ce programme affiche : `123.456703 123.46 123.4567`

Les caractères insérés entre le `%` et le `f` permettent de définir complètement le format d'affichage en donnant le nombre de décimales.

Dans le cas d'un entier, la spécification `%3d` signifie que l'entier correspondant sera affiché sur 3 digits.

spécificateur	type concerné	affichage
%c	char	caractère
%d	int	entier signé (décimal)
%h	short	entier signé
%x	int , unsigned int	entier en hexadécimal
%u	unsigned int	entier non signé
%ld	long int	entier long
%f	float	réel
%e	float	réel (notation scientifique)
%lf	double	réel en double précision
%le	double	réel en double précision (notation scientifique)
%s	char *	chaîne de caractères

TAB. 4.1 – Les spécificateurs de formats.

4.2.3 Affichage d'une chaîne de caractères

Pour une chaîne constante, le plus simple est d'utiliser la fonction `printf` avec un seul argument :

```
printf("Affichage très simple ...\n");
```

Si la chaîne est définie dans une variable, on utilise alors le format `%s` :

```
char *ch = "Autre affichage !";
...
printf("%s", ch);
```

Ici encore, on peut préciser le format en insérant un nombre pour préciser le nombre de caractères que prendra la chaîne à l'affichage.

scanf

4.3 Entrée formatée de caractères (`scanf`)

La fonction `scanf` permet de lire les caractères du clavier en les interprétant. On peut ainsi initialiser directement des variables entières ou réelles.

L'usage de cette fonction d'entrée est relativement complexe en comparaison des fonctions équivalentes dans d'autres langages (`Pascal`, `Ada`, `Modula2` ...). Son apprentissage nécessite donc une attention particulière.

4.3.1 Lecture d'une variable simple

Le procédé est analogue à celui de la fonction `printf` . Les formats sont les mêmes. Pour lire un entier, on écrira :

```
int n;
```

```
...
scanf("%d", &n);
```

Le premier argument de la fonction `scanf` est une chaîne composée **exclusivement** de formats. Dans cet exemple, le format de la donnée à lire est donné par `%d`. La donnée saisie au clavier ne devra donc contenir que des caractères dans l'ensemble :

{'0','1','2','3','4', '5','6','7','8','9','+','-' }

La variable est spécifiée par l'argument `&n`. À la différence de la fonction `printf` , il faut ici ajouter le caractère `&` avant le nom de la variable. Il indique que l'entier lu au clavier devra être stocké à «l'adresse de `n`». Cette procédure s'appliquera à la lecture de toutes les variables d'un type simple (***caractère*** , ***entier*** , ***réel***).

4.3.2 Lecture de plusieurs variables

Plusieurs variables peuvent être lues en un seul appel à `scanf` .

Exemple :

```
int i;
double X;
...
scanf("%d%lf", &i, &X);
```

Remarques :

→ La chaîne ne contient que des formats.

→ La saisie d'une variable de type ***caractère*** est problématique dans ce contexte : il faut éviter dans ce cas d'insérer des séparateurs (espaces ou tabulations) au moment de la saisie.

Exemple : Soit le programme suivant qui attend l'entrée d'un entier, puis d'un caractère et enfin d'un deuxième entier. :

```
#include <stdio.h>

int main() {
  int i,j;
  char c;;
  scanf("%d%c%d", &i, &c, &j);
  printf("%d %c %d\n",i,c,j);
  return 1;
}
```

$\begin{cases} \text{Si la saisie est } \boxed{\texttt{12e45}}\text{, la sortie est } \boxed{\texttt{12 e 45}} \\ \text{Si la saisie est } \boxed{\texttt{12 e 34}}\text{, la sortie est } \boxed{\texttt{12 0}} \end{cases}$

Cet exemple à lui seul montre les précautions à prendre lorsqu'une variable de type `char` doit être saisie.

Pour la saisie d'un caractère isolé, validé par l'appui sur Entrée on pourra utiliser la solution suivante :

```
char c;
...
scanf("%c%*c", &c);
```

Le format `%*c` indique qu'un caractère doit être lu (en l'occurence, Entrée) mais il ne sera pas pris en compte par `scanf` pour initialiser un variable. Cette technique permet de «vider» le *buffer* de clavier.

4.4 Les flux d'entrées-sorties, `cin` et `cout`

4.4.1 Généralités

Un flux (*stream*) est un type abstrait représentant un flot de données entre une source qui **produit** l'information et une cible qui **consomme** cette information.

Il intègre un *buffer* de stockage des données ainsi que des mécanismes pour les acheminer. Cette section applique les flux aux entrées et sorties standards. Le chapitre 14 les utilisera pour traiter les fichiers.

`cin/cout`

4.4.2 Mise en œuvre

Par défaut, trois flux sont prévus pour faciliter les opérations d'entrées/sorties standards :

- `cout` : sortie standard (l'écran).
- `cin` : entrée standard (le clavier).
- `cerr` : sortie standard d'erreur.

L'opérateur `<<` permet d'envoyer des valeurs dans un flux de sortie. L'opérateur `>>` permet d'extraire des valeurs d'un flux d'entrée.

Les flux prédéfinis sont déclarés dans le fichier `iostream.h`.

- `cout` est un objet de classe `ostream`
- `cin` est un objet de classe `istream`

Les notions de classe, d'objet et de flux sont définies à partir du chapitre 10. Il n'est pas nécessaire de maîtriser ces notions pour les utiliser.

Exemple

```
#include <iostream>
```

```
int main() {
    int i;
    double x=3.14;
    char *b = "les flux d'entrées/sorties\n";
    cout << "i = ";
    cin >> i;
    cout << "i = " << i << "; x = " << x << '\n' << b;
    return 1;
}
```

Remarques

- Pour le flux d'entrée `cin`, les espaces sont considérés comme des séparateurs. Ainsi, pour la lecture d'une chaîne de caractères, seul le premier mot de la chaîne est pris en compte.
- L'opérateur `&` disparaît pour la lecture de variables. La gestion de l'adresse est prise en compte par le compilateur.

4.4.3 Les manipulateurs

Les manipulateurs sont des éléments qui modifient la façon de lire ou d'écrire les variables ou constantes dans un flot. Ils sont déclarés dans le fichier `iomanip.h`. Les principaux sont donnés dans le tableau 4.2.

`oct`	lecture ou écriture d'un entier en octal
`dec`	lecture ou écriture d'un entier en décimal
`hex`	lecture ou écriture d'un entier en hexadécimal
`endl`	insère un saut de ligne
`ends`	insère le caractère nul (`'\0'`)
`setw(n)`	affichage de `n` caractères
`setbase(n)`	affichage dans la base n
`setprecision(n)`	affichage d'une valeur avec `n` chiffres significatifs.
`setfill(c)`	fait du caractère `c` le caractère de remplissage.
`flush`	vide les tampons après écriture.

Tab. 4.2 – manipulateurs de flux.

```
#include <iostream>
#include <iomanip>

int main() {
    double d = 12.345678;
    cout << d << endl
         << setw(10) << setfill('#') << setprecision(4) << d << endl;
    return 1;
}
```

Trace d'exécution :

```
> a.out
12.3457
#####12.35
> _
```

L'exécution montre que le réel `d` est affiché sur 10 caractères avec seulement 4 chiffres significatifs, et en prenant le `'#'` comme caractère de remplissage.

Chapitre 5

Les structures de commande

5.1 Les blocs d'instructions

Un bloc d'instructions est une séquence d'instructions encapsulée par une paire d'accolades. Un bloc peut posséder ses propres variables locales. Dans l'exemple suivant, deux blocs apparaissent :

- le bloc de la fonction principale `main`.
- un bloc isolé dans lequel est réalisée la permutation des valeurs des entiers `i` et `j`. Ce bloc interne possède une variable privée `m` qui n'est pas reconnue en dehors.

```
#include <iostream>
#include <iomanip>

int main() {
  int i,j;
  i=1, j=2;
  cout << i << ' ' << j << endl;
  {
    int m;
    m=i;
    i=j;
    j=m;
  }
  cout << i << ' ' << j << endl;
  return 1;
}
```

Remarque : `i` et `j` sont dites *variables locales* de la fonction `main`.

5.2 Les répétitions

while

5.2.1 L'instruction `while`

C'est la traduction en C++ de la structure ***tant que*** ...***répéter*** ...Elle permet de répéter un bloc d'instructions tant qu'une condition reste vraie.

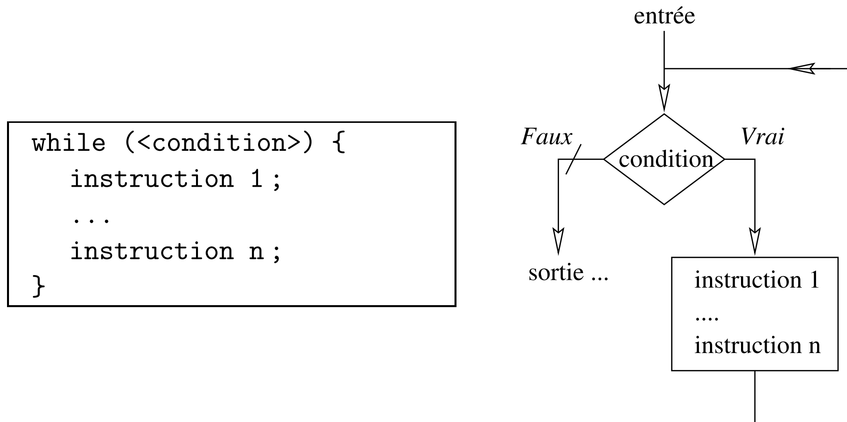


FIG. 5.1 – *Instruction* `while` .

Exemple : On désire concevoir un programme qui lit des entiers strictement positifs entrés au clavier et qui en fait la somme. Le processus s'arrête lorsqu'un entier négatif est entré par l'utilisateur.

Le passage de l'algorithme au C++ est immédiat.

entier n, S ;
écrire "n =" ;
lire n ;
S $\leftarrow$ 0 ;
tant que n $>$ 0 ***répéter***
    début
        S $\leftarrow$ S $+$ n ;
        écrire "n =" ;
        lire n ;
    fin
écrire "S = ", S

```
#include <iostream>

int main() {
  int n, S;
  cout << "n = ";
  cin >> n;
  S = 0;
  while (n > 0) {
    S += n;
    cout << "n = ";
    cin >> n;
  }
  cout << "S = " << S << '\n';
  return 1;
}
```

Remarque : Lorsque l'instruction `while` ne porte que sur une seule instruction, les accolades sont facultatives.

Exemple : le programme suivant calcule la somme des 100 premiers entiers.

```
#include <iostream>
#include <iomanip>

int main() {
    int  n=0, S=0;
    while (n<100) S += ++n;
    cout << "La somme des 100 premiers entiers est " <<  S << endl;
    return 1;
}
```

5.2.2 L'instruction `do ...while`

`do..while`

C'est l'expression en `C++` de la structure de commande ***répéter*** ...***tant que*** .
Cette instruction est similaire à la précédente, mais les instructions du bloc à répéter sont exécutées au moins une fois.

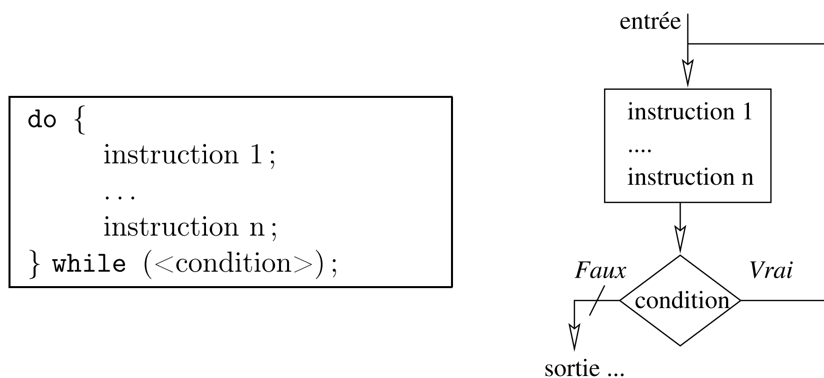


FIG. 5.2 – *Instruction* `do ...while` .

Exemple : On souhaite qu'un utilisateur entre un code à quatre chiffres compris entre 1000 et 9999. Un code erroné est refusé. Il s'agit donc de lire des valeurs jusqu'à ce que l'une d'entre elles répond au critère d'acceptation.

entier code ;
répéter
  début
    écrire "Code =" ;
    lire code ;
  fin
tant que code $<$ 1000
      ou code $>$ 9999
écrire "Code accepté" ;

```
#include <iostream>

int main() {
    int code;
    do {
      cout << " Code = ";
      cin >>code;
    } while (code<1000 ||  code>9999);
    cout << "Code accepté\n";
    return 1;
}
```

Remarque : Comme pour l'instruction `while` , l'usage des accolades est facultatif lorsque le bloc n'est composé que d'une seule instruction.

Voici une autre version du calcul de la somme des 100 premiers entiers :

```
#include <iostream>

int main() {
    int n,S;
    n = S = 0;
    do S += ++n; while (n<100);
    cout << "La somme des 100 premiers entiers est " << S << '\n';
}
```

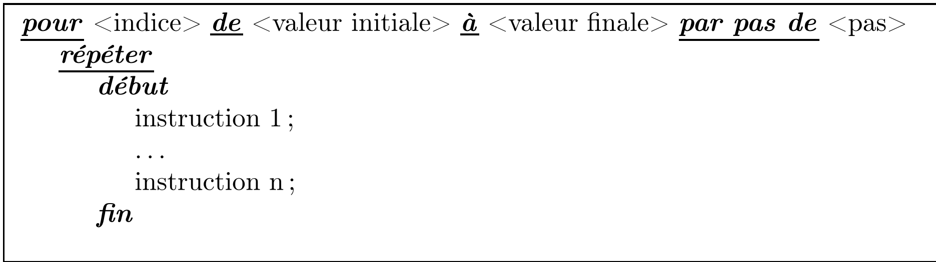
`for`

5.2.3 L'instruction `for`

L'instruction `for` du `C++` permet non seulement de coder la répétition ***pour*** ...vue en algoritmique, mais en plus, elle est une généralisation des répétions.

Utilisation simple

La répétition ***pour*** est utilisée pour exécuter une séquence d'instructions un nombre de fois préalablement connu :



pour <indice> ***de*** <valeur initiale> ***à*** <valeur finale> ***par pas de*** <pas>
répéter
début
instruction 1 ;
...
instruction n ;
fin

Le codage prévu en `C++` est conforme au canevas suivant

```
for (instruction1; <condition>; instruction2 ) {
    instructions3
}
```

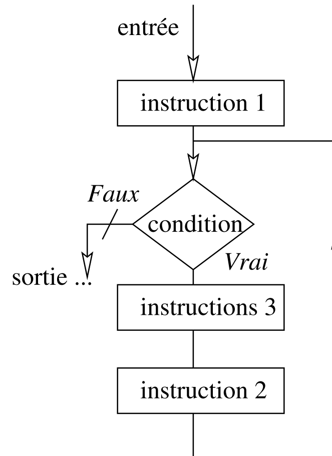


FIG. 5.3 – *Organigramme de l'instruction* `for` *.*

Le comportement de cette instruction est décrit par l'organigramme de la figure 5.3.

La réalisation d'une structure de commande ***pour*** peut donc de faire en appliquant les règles suivantes :

→ la valeur initiale de l'indice sera affectée dans `instruction1`,

→ la valeur finale sera traduite dans la `condition`,

→ l'incrément sera appliqué à l'indice lors de l'exécution de l'`instruction2`.

Exemple : le programme suivant affiche les 100 premiers carrés :

```
#include <iostream>

int  main() {
  int i;
  for (i=1; i<=100; i++) {
    cout << i*i;
  }
  cout << "\n"
  return 1;
}
```

L'instruction `for` traduite par une instruction `while`

L'exemple précédent peut être codé par une instruction `while` :

```
#include <iostream>

int main() {
  int i=1;
  while (i<=100) {
    cout << i*i;
    i++;
  }
  cout << "\n";
  return 1;
}
```

L'instruction `for` est un «raccourci syntaxique» pour l'écriture d'une instruction `while` .

5.3 Les choix

Ils sont très conventionnels et donnent une traduction immédiate des expressions vues en algorithmique.

`if..else`

5.3.1 Le choix simple `if ...else`

```
if (<condition>) {
   instruction1 ;
}
else {
   instruction2 ;
}
```

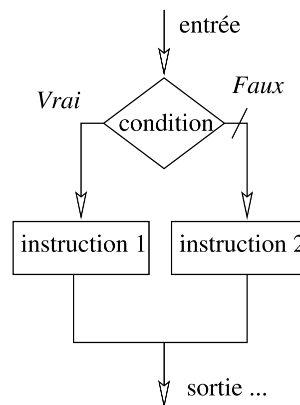


FIG. 5.4 – *Choix simple (alternative).*

Lorsque l'un des blocs d'instructions (ou les deux) est réduit à une seule instruction, les accolades sont facultatives.

Les choix simples répétés

Une instruction quelconque du `if` peut être elle même une instruction `if` . Ceci permet de répéter les structures `if` pour donner le codage d'un choix multiple.

Exemple : le programme suivant calcule une réduction en fonction de l'âge de l'usager :

→ 50% pour les moins de 12 ans,

→ 20% entre 12 et 25 ans,

→ 40% pour les plus de 60 ans.

```
#include <iostream>

int main() {
  unsigned int age, reduction;
  cout << "Quel est votre âge ? ";
  cin >> age;

  if (age < 12) reduction = 50;
  else if (age <25) reduction = 20;
  else if (age <60) reduction = 0;
  else reduction = 40;

  cout << "Votre réduction est de " << reduction << '\n';
  return 1;
}
```

Étudier le comportement de ce programme en l'absence de `else` ...

switch

5.3.2 Le choix multiple `switch ...case`

Le choix multiple du `C++` est limité à la comparaison d'une variable d'un type simple (***caractère*** , ***entier*** ou ***réel***) à plusieurs constantes. Les différents cas traités ne font pas l'objet de la création d'un bloc d'instructions. C'est l'instruction `break` qui détermine la fin du treitement. Elle renvoie systématiquement à l'accolade fermante de l'instruction `switch` . L'étiquette `default` indique le traitement à réaliser lorsqu'il n'y a pas de correspondance entre la variable et les différentes constantes. Ce traitement est facultatif.

Exemple

```
#include <iostream>
#include <iomanip>

int main() {
  cout << "Entrez un No de département :";
  int dep;
  cin >> dep;
  switch(dep) {
  case 22:
    cout << "Cotes d'Armor";
    break;
  case 29:
    cout << "Finistère";
```

```
switch (variable) {
   case valeur1 :
      instructions1 ;
      break ;
   case valeur2 :
      instructions2 ;
      break ;    ...
   case valeurn :
      instructionsn ;
      break ;
   default :
      instructions0 ;
      break ;
}
```

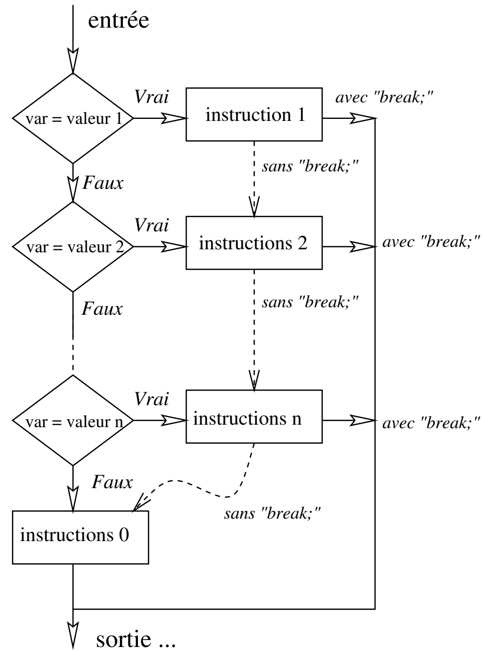


Fig. 5.5 – *Choix multiple.*

```
      break;
    case 35:
      cout << "Ile et Vilaine";
      break;
    case 56:
      cout << "Morbihan";
      break;
    default:
      cout << "Ce n'est pas un département breton";  break;
    }
    cout << endl;
    return 1;
  }
```

Chapitre 6

Les sous-programmes

6.1 Définition d'un sous-programme

Un sous-programme est un élément de programme **nommé** et éventuellement **paramétré**, que l'on définit afin de pouvoir ensuite l'appeler par son nom en affectant, s'il y a lieu, des valeurs aux paramètres. C'est un concept qui existe depuis le début de la programmation. Les intérêts de l'utilisation des sous-programmes sont :

- le gain de place en mémoire pour le code du programme : lorsqu'un sous-programme est appelé plusieurs fois, dans une boucle, par exemple,
- les notions d'abstraction et de modularité : utiliser un appel de sous-programme permet de d'écrire une application en faisant abstraction (en dissimulant) les détails de la fonction que réalise ce sous-programme .

Quelques règles élémentaires doivent être appliquées pour l'écriture de sous-programmes .

- Un sous-programme doit être **homogène** : il doit réaliser une tâche précise, formant un tout.
- Il doit être de **taille «raisonnable»**. La compréhension et la gestion du programme en dépend.

La **définition** du sous-programme est composée d'une **spécification** qui indique son nom, ses paramètres (ou arguments) avec leurs caractéristiques (nom, type) et d'un **corps** comprenant éventuellement des déclarations d'objets locaux au sous-programme et les instructions à exécuter.

Déclaration algorithmique d'un sous-programme

fonction `NomDeFonction`($para_1$: $type_1$, $\cdots$,$para_n$: $type_n$) : $type_{retour}$;

Le sous-programme de identifié sous le nom «`NomDeFonction`» reçoit les arguments $para_1, \cdots, para_n$ de types respectifs $type_1, \cdots, type_n$ et produit en retour une variable de type $type_{retour}$.

Déclaration d'un sous-programme (ou fonction) en C++

```
type_retour NomDeFonction(type_1 para_1, ..., type_n para_n)
```

Lorsque la fonction ne renvoie rien (c'est une procédure), `type_retour` prend le type «vide» `void`.

Exemple :

<u>fonction</u> Max(a,b : ***entier***) : ***entier*** ;
début
   <u>si</u> a>b ***<u>retourner</u>*** a ;
   <u>sinon</u> ***<u>retourner</u>*** b ;
fin Max ;
	`int Max(int a, int b) {`
	`    if (a>b) return a;`
	`    else return b;`
	`}`

Dans cet exemple, la ligne algorithmique :

<u>fonction</u> Max(a,b : ***entier***) : ***entier*** ;

traduite en C++ par

```
int Max(int a, int b)
```

est la **déclaration** (ou **spécification**) d'une fonction dont le nom est Max, qui reçoit deux paramètres ***entier*** a et b, et qui produit en sortie un autre ***entier*** .

a et b sont appelés les **paramètres formels** de la fonction Max. Ils ne servent qu'à l'intérieur du corps du sous-programme . Le corps du sous-programme est défini dans l'algorithme par les instructions contenues entre «***début*** » et «***fin*** Max ;». En C , les instructions du sous-programme (ou **fonction**) font l'objet d'un bloc d'instructions.

6.2 Appel d'un sous-programme

L'appel d'un sous-programme se fait en mentionnant son nom, suivi des paramètres **effectifs** figurant entre parenthèses et séparés par des virgules. L'appel d'une fonction peut être une expression à part entière ou un opérande dans une expression plus complexe. Chaque paramètre effectif doit correspondre à un paramètre formel.

Il existe deux mécanismes importants de substitution entre paramètres effectifs et paramètres formels :
- **le passage par valeurs**,
- **le passage par référence**.

6.3 Passage de paramètres par valeur

La valeur du paramètre efectif est recopîée dans le paramètre formel à l'entrée du sous-programme . Le sous-programme travaille sur une copie du paramètre effectif. Celui-ci n'est pas modifié à l'issue du sous-programme .

Exemples Reprenons la fonction Max du paragraphe précédent.

```
#include <iostream>

int Max(int x, int y) {
    int max;
    if (x<y) max=y; else max=x;
    x=0; // utile juste pour montrer le passage par valeur
    y=0;
    return max;
}

int main() {
    int a,b;
    cout << "a = ";  cin >> a;
    cout << "b = ";  cin >> b;
    int m = Max(a,b);
    cout << "Le max. de " << a << " et " << b
         << " est " << m << '\n';
    return 1;
}
```

L'exécution du programme donne :

```
% a.out
a = 12
b = 23
Le max. de 12 et 23 est 23
% _
```

→ L'appel `m = Max(a,b)` est un appel par valeur,

→ les paramètres formels `x` et `y` sont remplacés par `a=12` et `b=23` qui sont les données entrées par l'utilisateur ;

→ les paramètres formels `x` et `y` sont mis à 0 à la fin du sous–programme, ceci n'a aucune incidence sur le programme principal. Ceci prouve bien que la fonction `Max` reçoit des copies des paramètres `a` et `b`.

6.4 Passage de paramètres par référence

6.4.1 Notion de référence

Une variable de type *référence* correspond dans la réalité à un accès par adresse, mais l'utilisation d'une telle variable se fait comme pour une variable normale.

Exemple :

```
#include <iostream>

int main(){
    int i=7;
    cout << "i= " <<  i << '\n';
    int& j = i;
    j++;
    cout << "i= " <<  i << '\n';
    return 1;
}
```

1. On déclare puis on affiche un entier `i`.
2. `j` est une *référence* sur un entier. Écrire `j=i ;` revient à dire que `j` et `i` représente maintenant la même donnée physique en memoire.
3. `j++ ;` modifie donc à la fois `i` et `j`.

6.4.2 Les références utilisées comme paramètres

L'adresse du paramètre effectif est communiquée au sous-programme qui travaille alors directement sur l'«original» et non sur une copie locale. Le sous-programme peut alors modifier la valeur d'une variable du programme qui l'a appelé.

En `C++` , les paramètres sont passés par valeur. Pour réaliser un passage par référence, il faut explicitement passer l'adresse de la variable au sous-programme .

Insuffisance du problème par valeur. Considérons une fonction `permute` qui échange les valeurs des deux paramètres reçus.

```
#include <iostream>

void permute(int x, int y) {
    int m=x; x=y; y=m;
}

int main(){
    int a=1, b=2;
    cout << "a=" << a << " b=" << b << '\n';
    permute(a,b);
    cout << "a=" << a << " b=" << b << '\n';
    return 1;
}
```

Exécution

```
% a.out
a=1 b=2
a=1 b=2
% _
```

On constate que la permutation opérée sur les paramètres formels est sans conséquence sur les paramètres effectifs.

Solution. Il faut explicitement indiquer au sous programme de travailler sur les paramètres originaux. Paramètres formels et paramètres effectifs sont alors confondus. Pour que le paramètre formel `x` soit confondu avec le paramètre effectif, il faut le précéder du signe de référence `&`. L'exemple modifié devient :

```
#include <iostream>

void permute(int &x, int &y) {
    int m=x; x=y; y=m;
}

int main(){
    int a=1, b=2;
    cout << "a=" << a << " b=" << b << '\n';
    permute(a,b);
    cout << "a=" << a << " b=" << b << '\n';
    return 1;
}
```

L'exécution donne alors :

```
% a.out
a=1 b=2
a=2 b=1
% _
```

6.5 Paramètres déclarés `const`

Lorsqu'une fonction ne modifie pas le paramètre réel reçu, le paramètre formel correspondant peut être déclaré `const`. C'est une précaution car le compilateur vérifie qu'aucun accès en écriture n'est fait sur le paramètre. C'est aussi un habitude de clarté car la simple lecture de la déclaration de la fonction indique que le paramètre réel ne sera pas modifié.

La présence du mot clé `const` prend tout son sens lorsqu'il s'agit d'un passage de paramètre par référence. Nous y reviendrons plus en détail dans le chapitre 8.

6.6 Les pointeurs comme paramètres

L'exemple de la fonction de permutation peut être traduit en utilisant les notions de pointeur et d'adresse :

```
#include <iostream>

void permute(int *px, int *py) {
    int m=*px; *px=*py; *py=m;
}

int main(){
    int a=1, b=2;
    cout << "a=" << a << " b=" << b << '\n';
    permute(&a,&b);
    cout << "a=" << a << " b=" << b << '\n';
    return 1;
}
```

C'est la seule possibilité en `C` . En `C++` on préférera bien sûr la solution précédente qui utilise les références.

6.7 Les fonctions `inline`

Le qualificatif `inline` peut-être appliqué à une fonction. Dans ce cas, le compilateur recopie le code de cette fonction à chaque fois qu'elle est appelée. Il n'y a donc pas de branchement, mais une substitution. Le nombre d'instructions du programme se trouve par conséquent augmenté. Cette technique doit être réservée aux fonctions courtes (quelques lignes seulement) dans des situations ou le temps d'exécution devient un paramètre critique.

Exemple :

```
inline int max(int a, int b) {return a>b ? a : b;}
```

6.8 Surcharge de fonctions

Une fonction est reconnue, non seulement par son nom, mais aussi par sa signature. Ainsi, deux fonctions de même nom mais recevant des arguments de type et/ou en nombre différents sont parfaitement différenciées par le compilateur.

Exemple avec des nombre de paramètres différents

```
#include <iostream>
#include <iomanip>

// Fonction avec 2 arguments
int max(int a, int b){
    if (a<b) return b;
    return a;
}
```

```
// Fonction avec 3 arguments
int max(int a, int b, int c){
    return max(a, max(b,c));
}

int main(){
    cout <<  max(1,2) <<  max(1,2,3) << endl;
    return 1;
}
```

Exemple avec des types différents

```
int abs(int x) { if (x>0) return x; else return -x; }
double abs(double x) { if (x>0) return x; else return -x; }
```

6.9 Les paramètres «par défaut»

Lors d'un appel de fonction, les derniers arguments peuvent être omis et initialisés à une valeur par défaut. Le nombre de paramètres peut apparaître variable à l'appel de la fonction, mais en réalité, la fonction utilise toujours le même nombre de paramètres.

Exemple

```
#include <iostream>
#include <iomanip>

const long MAX = 2147483647; // +oo

long Min(long x, long y, long z=MAX) {
    long m;
    if (x<y) m=x; else m=y;
    if (m<z) return m; else return z;
}

int main() {
    cout << Min(2,1,3) << endl;
    cout << Min(-3, -7) << endl;
    return 1;
}
```

Dans le cas ou seuls deux paramètres sont utilisés, le troisième est remplacé par la valeur par défaut.

Chapitre 7

Les tableaux

De nombreuses applications font appel à la manipulation de données multiples ayant des caractéristiques communes. En général, elles apparaissent alors sous la forme de données indexées $(x_1, x_2, \cdots, x_n)$. Il est utilie, dans de telles situations de disposer d'une structuation de données en tableau qui permet d'associer sous un nom unique l'ensemble des données.

Ceci sera possible à chaque fois que

$$\begin{cases} \text{le nombre d'éléments de données est connu au départ,} \\ \text{tous les éléments de données sont de même type.} \end{cases}$$

7.1 Déclaration et définition d'un tableau

Un tableau est une représentation séquentielle de données. C'est à dire que les éléments de données (tous de même type) occupent des emplacements contigus en mémoire.

7.1.1 Déclaration

Un tableau sera défini par :

→ un nom,

→ un nombre d'éléments (c'est la *taille* du tableau),

→ le type des éléments.

Exemple : la déclaration

```
int Tab[32];
```

annonce un tableau de nom `Tab` constitué de 32 entiers

7.1.2 Accès aux membres

Si le tableau est de taille n, alors ses éléments sont indexés de 0 à $n-1$. Pour l'exemple précédent, `Tab[0]` représente le premier élément du tableau et `Tab[31]` le dernier élément.

7.1.3 Exemple

Le programme suivant initialise un tableau de 100 réels avec les racines carrées des 100 premiers entiers.

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

int main() {
  float X[100];
  int i;

  for (i=0; i<100; i++) X[i] = sqrt(i);
  for (i=0; i<100; i++) cout << X[i];

  return 1;
}
```

7.1.4 Lecture d'un élément de tableau sur l'entrée standard

Un élément du tableau est considéré commenune donnée élémentaire. Si son type est simple, on procède à la lecture comme d'habitude :

```
int Tab[10], i;
cout << "Initialisation des éléments du tableau.";
for (i=0; i<10; i++) {
   cout << "Tab[" << i << "] = ";
   cin >> Tab[i];
}
```

7.1.5 Tableau initialisé

Pour initialiser un tableau dans la déclaration, il suffit d'énumérer les valeurs des éléments du tableau entre les caractères `{` et `}`.

Exemple :

```
float T[4] = {12.0, 14.8, 11.7}
```

Cette déclaration permet de fixer les valeurs des éléments `T[0]` à `T[2]`. La valeur de `T[3]` reste indéterminée.

7.2 Les tableaux de caractères

7.2.1 Chaînes de caractères

Une chaîne de caractères est une liste de caractères dont le dernier est le caractère nul. Ainsi, une chaîne de n caractères occupe en réalité $n+1$ octets de la mémoire. Très souvent, une chaîne de caractères est stockée dans un

tableau de caractères.
Exemple :

```
...
char T[16];
...
cin >> T;
...
```

Dans cet exemple, une partie du tableau T est initialisée avec une chaîne de caractères[1]. L'étiquette T représente l'adresse du premier caractère du tableau. On a l'équivalence :

$$T \Leftrightarrow \&T[0]$$

Si la chaîne de caractères *"accord"* est entrée lorsque les lignes de l'exemple précédent sont exécutées, le tableau T sera initialisé de la manière suivante :

$T =$	'a'	'c'	'c'	'o'	'r'	'd'	0	?	?	?	?	?	?	?	?	?

Les caractères '?' représentent des octets de la mémoire non initialisés.

7.2.2 Fonctions d'entrées de chaînes de caractères

En dehors du flux d'entrée `cin`, deux fonctions `C` de `stdio.h` permettent de lire une chaîne de caractères, ce sont :

`scanf` l'instruction `scanf("%s", T);` recopie à partir de l'adresse T tous les caractères entrés au clavier, jusqu'à ce qu'un caractère séparateur[2] soit rencontré. On ne peut donc pas utiliser cette fonction pour saisir un nom composé.

`gets` l'instruction `gets(T)` recopie à partir de l'adresse T tous les caractères entrés au clavier, jusqu'à ce que le caractère *retour-chariot* soit rencontré. À la différence de la fonction précédente, elle permet de saisir un nom composé. Pour plus de sécurité dans le programme, on peut remplacer `gets(T)` par `fgets(T,n,stdin)`, `n` étant le nombre maximum de caractères pouvant être lus.

Exemple

Le programme suivant :

```
#include <stdio.h>
int main() {
    char str[16];
    printf("?");
    fgets(str,15,stdin);    // prend toute une ligne (< 16 car.)
```

[1]On remarque ici qu'aucune précaution n'est prise quant au nombre de caractères entrés. On *suppose* que l'entrée ne contiendra pas plus de 15 caractères...

[2]Espace, retour chariot ou tabulation.

```
        printf("%s\n",str);
        printf("?");
        scanf("%s",str); // ou : cin >> str;
        printf("%s\n",str);
        return 1;
    }
```

À l'exécution de ce programme, si on entre à chaque fois le message : *Au revoir*!!!, l'affichage est alors :

```
 > a.out
?Au revoir
Au revoir
?Au revoir
Au
 > _
```

Ceci montre bien que la fonction `fgets` lit tous les caractères jusqu'à rencontrer un «retour chariot» alors que la fonction `scanf` s'arrête au premier séparateur rencontré.

Notons qu'à l'issue de cet exemple, le *buffer* de clavier n'est pas vide et contient toujours la chaîne de caractères `" revoir\n"`.

Remarque : la solution la plus simple en `C++` reste, bien sûr, l'initialisation avec un flux :

```
cin >> str;
```

7.2.3 Opérations sur les chaînes de caractères

Afin de manipuler plus aisément les chaînes de caractères, la librairie standard du langage `C` propose quelques fonctions prédéfinies reconnaissables par le préfixe *str* (pour *string*).

Les notions d'adressse et de pointeurs présentées au paragraphe 8.3 (page 61) sont utiles pour la bonne compréhension des fonctions qui suivent.

Parmi ces fonctions on trouve :

`int strcmp(char *s1, char *s2)` Reçoit deux adresses de chaînes de caractères $s1$ et $s2$ et renvoie :

- **1** si $s1 > s2$ pour l'ordre alphabétique ;
- **0** si les deux chaînes sont rigoureusement identiques ;
- **-1** si $s1 < s2$ pour l'ordre alphabétique.

`char *strcpy(char *s1, char *s2)` La chaîne pointée par $s2$ est recopiée à partir de l'adresse $s1$. Ceci suppose que $s1$ pointe sur une zone de mémoire «convenable» , c'est à dire un tableau suffisemment grand, ou une zone de mémoire allouée par le système. Pour répondre à certaines commodités de programmations, le pointeur $s1$ est renvoyé.

`int strlen(char *s1)` Renvoie la longueur de la chaîne pointée par $s1$.

`char *strcat(char *s1, char *s2)` Les chaînes de caractères pointées respectivement par $s1$ et $s2$ sont juxtaposées. Le résultat est une nouvelle chaîne de caractères stockée à partir de l'adresse $s1$. Ceci suppose que $s1$ pointe sur un tableau dont la taille initiale est assez grande pour recevoir la concaténation des 2 chaînes de caractères[3].

7.3 Les tableaux de dimension 2

Ils permettent de contenir des données représentées physiquement en 2D :
- carte de relief (les éléments du tableau sont des altitudes),
- image (les éléments sont des niveaux de gris ou des couleurs),
- ...

Exemple : Le programme suivant affiche tous les points d'une carte qui se trouvent au dessus d'une altitude donnée par l'utilisateur. La fonction `initCarte` a pour rôle d'initialiser le tableau avec des valeurs cohérentes.

```
#include <iostream>

float Carte[100][200];

void initCarte() {
  int i,j;

  for (i=0; i<100; i++)
    for (j=0; j<200; j++)
      Carte[i][j] = (float)(i+j);
}


int main() {
  int i;
  register int j;
  float A;

  initCarte();
  cout << "Altitude de test : ";
  cin >> A;
  for (i=0; i<100; i++)
    for (j=0; j<200; j++)
      if (Carte[i][j] > A) cout << "X=" << j << "  Y=" << i << '\n';
  return 1;
}
```

[3] Aucun contrôle n'est effectué par le système.

Remarques :

- La déclaration `float Carte[100][200] ;` annonce un tableau de réels de dimension 2 : 100 lignes de 200 éléments chacune. On dit aussi, 100 lignes et 200 colonnes.
- Comme pour les tableaux à une dimension, l'indexation commence à partir de 0.
- Les programmes traitant des tableaux 2D contiennent souvent des boucles `for` imbriquées. Il est souhaitable de déclarer l'index de la boucle intérieure comme `register`.

Pointeur

7.4 Adresses et pointeurs

Un pointeur sur une variable, c'est l'adresse de cette variable. La déclaration

```
int k;
```

indique que `k` est un entier. Pour obtenir l'adresse de `k`, il suffit d'écrire : `&k`.

Pour déclarer une variable de type pointeur (c'est à dire pouvant contenir une adresse, il faut utiliser l'opérateur unaire `*`. Par exemple,

```
int *pk;
```

indique que `pk` est l'adresse d'un entier. Une autre façon (équivalente) de voir les choses, est de dire que `*pk` est un entier. `*pk` indique en effet le **contenu** de la variable `pk`.

Remarque : Il est obligatoire de *typer* les pointeurs. En toute rigueur, une variable de type pointeur aura toujours le même encombrement en mémoire, indépendemment du type de la variable pointée. Cependant, le `C++` utilise le type du pointeur pour l'arithmétique des pointeurs, notion essentielle pour la manipulation des structures séquentielles de données.

Exemple :

```
int *pk; // pointeur non initialisé
int k;
pk = &k; // initialisation du pointeur
*pk = 4;
cout << "k = " <<  k << endl;
```

1. `*pk` est un entier.
2. `pk` est une adresse d'entier ou un «`int *`».
3. `k` est un entier.
4. `&k` est une adresse (celle de `k`).
5. Un pointeur doit être initialisé avant d'écrire une valeur à l'adresse pointée. Ici, `pk = &k;` réalise cette initialisation[4].

[4]pointeur non initialisé =: «segmentation fault»

Adresse d'un tableau

Avec la déclaration :

```
int T[128];
```

`T` est un symbole qui représente l'adresse du tableau en mémoire. On a l'équivalence : `T` ⇔ `&T[0]`

Arithmétique des pointeurs

On peut ajouter un déplacement à un pointeur, de façon à obtenir une nouvelle adresse. Ainsi, pour afficher le tableau d'entiers déclaré précédemment, on peut écrire :

```
int *p;
for (p=T; p<(T+128); p++) cout << *p << " ";
```

La notion d'**arithmétique des pointeurs** est illustrée par l'instruction `p++` : l'incrémentation du pointeur est conforme au type de la variable pointée. On a déclaré

```
int *p;
```

et donc :

`p++;` (pour le langage) ⇔ *p = p + sizeof(int);* (en mémoire)

7.5 Allocation dynamique de mémoire

Principe Il est possible d'obtenir de la mémoire pour stocker une variable simple ou un tableau de valeurs en cours d'exécution du programme. L'allocation dynamique de mémoire pour une variable d'un type natif n'a pas beaucoup de sens, mais le principe deviendra plus intéressant pour l'allocation de structures ou d'objets dans les chapitres suivnts.

Allocation d'une variable simple (opérateur `new`) Considérons un pointeur non initialisé :

```
int *pi;
```

Pour que ce pointeur contienne l'adresse d'un entier, on écrira :

```
pi = new int;
```

sizeof(int) octets supplémentaires ont été alloués aux données du programme.

`new []`

Allocation d'un tableau (opérateur `new []`) Si `p` doit pointer sur un tableau de 128 entiers on écrit :

```
int *p;
p = new int[128];
```

128×*sizeof(int)* octets supplémentaires ont été alloués aux données du programme.

Restitution de la mémoire (`delete` et `delete[]`) Lorsque la mémoire est allouée dynamiquement, il est essentiel de la restituer après usage. Supposons en effet que l'allocation dynamique figure dans une boucle du programme, au bout d'un certain temps, les ressources en mémoire du système seront vite épuisées si chaque allocation n'est pas compensée par une restitution. L'opérateur `delete` est prévu à cet effet, pour libérer la mémoire allouée dans les deux exemples précédents, on écrira :

`delete[]`

```
delete pi;
delete[] p;
```

Chapitre 8

Les structures

8.1 Définition

Une structure est un ensemble comprenant une ou plusieurs variables, en général de type différents, qu'on regroupe sous un seul nom pour mieux organiser le traitement. Ceci devient incontournable dès l'instant ou les données manipulées sont complexes. Définir une sructure revient à imaginer un nouveau type pour le langage. On dit que l'on a défini un *type abstrait.*

`struct`

Déclaration d'une structure : la forme générale d'une déclaration de structure est :

```
struct <nom>{
      <membre 1> ;
      <membre 2> ;
     ...
      <membre n> ;
} ;
```

Dans cette déclaration, `struct` est un mot clé réservé du langage, *<nom>* est une étiquette qui permet d'identifier la nouvelle structure et *<membre i>* sont des déclarations de variables de type de base ou de type abstrait (donc du type d'une autre structure préalablement définie).

Exemple : soit à modéliser un type pour représenter les points du plan. On choisit de décrire un point du plan par un nom (tableau de caractères) et ses deux coordonnées réelles.

On défini dans le plan un type abstrait `struct point` de la manière suivante :

```
struct point {
     double X, Y;
     char Nom[8];
};
```

8.2 Déclaration et définition d'une variable structurée

Lorsque la structure est déclarée, nous obtenons simplement un nouveau concept abstrait (un nouveau type utilisable). On ne dispose d'aucune variable. Il faut maintenant en déclarer. La déclaration se fera de la manière suivante :

```
struct point M1; // déclaration de M1
struct point M2 = {2, 3.5, "M2"}; // déclaration et définition de M2
point M3;
```

La variable $M1$ est non initialisée alors que $M2$ l'est. Le mot clé `struct` est facultatif pour déclarer une variable.

8.3 Accès aux membres d'une structure

L'accès se fait par l'opérateur `'.'`. À gauche du `'.'` on place la variable de type abstrait, et à droite, on place le membre de la structure auquel on veut accéder. Par exemple, les lignes suivantes initialisent la variable $M1$ précédente, puis affichent $M1$ et $M2$.

```
cout << "Entrer le nom, puis les deux coordonnées :";
cin >> M1.Nom >> M1.X >> M1.Y;
cout <<  M1.Nom << "(" << M1.X << "," << M1.Y << ")" << endl;
cout <<  M2.Nom << "(" << M2.X << "," << M2.Y << ")" << endl;
```

remarques :

`M1.X` est de type `double`

`M1` est de type `struct point`

`M1.Nom` est de type `char[]`

8.4 Structures imbriquées

Un type abstrait peut être utilisé pour en définir un autre. Ainsi, on peut définir un segment à partir de deux points :

```
 ...
struct segment {
  point A, B;
  double Longueur;
};
 ...
  segment S = { {1, 2.5, "R"}, {-1, 3, "T"}, 0.0};
  segment U;
 ...
```

La figure 8.1 représente les données avec différents points de vue.

→ `S` et `U` sont des variables composées de type `segment`

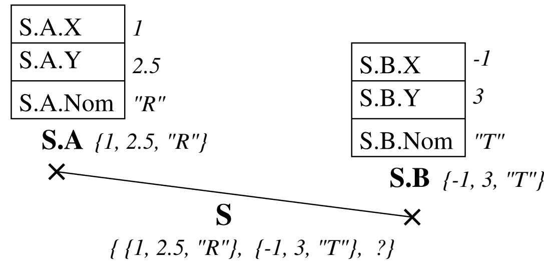


FIG. 8.1 – *Structures imbriquées : un segment composé de deux points.*

→ `S.A` est de type `point`

→ `S.B.Y` est de type `double`

→ ...

8.5 Tableaux de structures

La création de tableaux de structures se fait comme pour la création d'un tableau pour un type natif.

Exemple : déclaration un tableau de 5 structures initialisées, puis affichage.

```
point Tab[5] = {
  {0, 1, "A"},
  {2, 3, "B"},
  {4, 5, "C"},
  {6, 7, "D"},
  {8, 9, "E"}
};

for (int i=0; i<5; i++)
  cout << Tab[i].Nom << "(" << Tab[i].X << "," << Tab[i].Y << "); ";
cout << endl;
```

Un tableau de structures peut lui même être membre d'une structure.

Exemple : une «polyligne» est une ligne brisée (figure 8.2), elle peut être représentée par un tableau de points reliés entre eux. Pour notre exemple, on limite ce nombre de points à 5, et une donnée `N` précise le nombre effectif de points pour la polyligne ($2 \leq N \leq 5$)

```
struct polyligne {
  int N; // Nombre de points (de 2 à 5)
  point T[5];
};

 ...
```

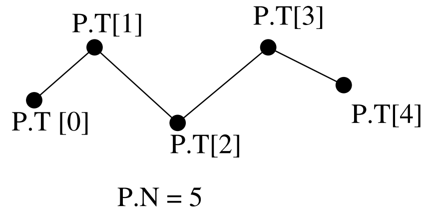


FIG. 8.2 – *Polyligne avec 5 points.*

```
polyligne P = {3,
              {{1, 2, "A"},
               {4, 3, "B"},
               {9, 5, "C"}}
};

for (int i=0; i<P.N; i++) cout << P.T[i].Nom << ", ";
cout << endl;
```

8.6 Références et pointeurs

Les **références aux structures** deviennent incontournables dès l'instant ou on doit utiliser des fonctions :

- **pour une modification :** si la fonction doit modifier son paramètre, le passage par référence est obligatoire.
- **pour un simple accès :** les structures contiennent souvent un volume important de données. Aussi, lors de l'utilisation de structures comme paramètres pour des fonctions, il convient d'utiliser dans la fonction une référence à la variable originale.

Exemple : fonctions `affichePoint` et `initPoint` :

```
 ...
void affichePoint(const point &M){
  cout << M.Nom << "(" << M.X << "," << M.Y << ")";
}

void initPoint(point &M){
  cout << "Nom: "; cin >> M.Nom;
  cout << " X = "; cin >> M.X;
  cout << " Y = "; cin >> M.Y;
}

int main() {
  point M3;
  initPoint(M3);
  affichePoint(M3);
  return 1;
}
```

La fonction `affichePoint` n'est pas supposée modifier le paramètre reçu. Le paramètre est donc précédé du mot `const`.

8.6.1 Pointeurs sur structures

Les pointeurs permettent de manipuler les adresses et de réaliser des créations dynamiques de structures.

Exemple : `point *pM ;` pM est un **pointeur sur une structure**. Il doit être initialisé avant toute utilisation. Il y a deux façons de l'initialiser : par une allocation avec l'opérateur `new` , ou bien en lui affectant l'adresse d'une structure existante :

```
point *pM = new point; // allocation dynamique
initPoint(*pM);        // *pM est un point
affichePoint(*pM);
delete pM;             // restitution

point *pM2;
pM2 = &M2;   // pM2 pointe vers M2
pM2->X = 123;
affichePoint(M2); // M2 est modifié (M2.X = 123)
```

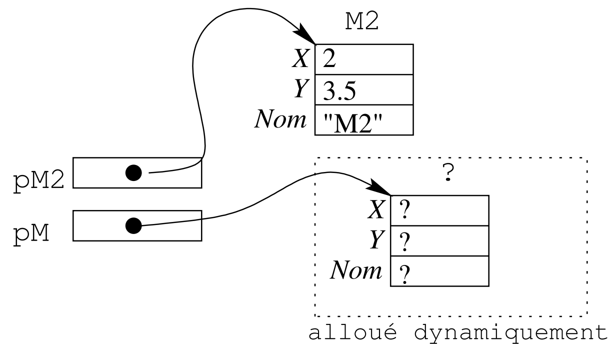


FIG. 8.3 – *Initialisation de pointeurs*

Remarques

→ il ne faut pas oublier d'appliquer l'opérateur `delete` à un pointeur alloué par `new`.

→ l'accès aux membres d'une structure pointée se fait avec l'opérateur `'->'`. Dans l'exemple, `pM2->X` et `M2.X` représentent la même variable en mémoire. On pourrait également écrire `(*pM2).X`.

→ `*pM` est la variable pointée par `pM`.

8.7 Code complet de l'exemple du chapitre 8

```
#include <iostream>
#include <iomanip>

/* Définition des types */
struct point {
  double X, Y;
  char Nom[8];
};

struct segment {
  point A, B;
  double Longueur;
};

struct polyligne {
  int N; // Nombre de points (de 2 à 5)
  point T[5];
};

/* Fonctions */
void affichePoint(const point &M){
  cout << M.Nom << "(" << M.X << "," << M.Y << ")";
}

void initPoint(point &M){
  cout << "Nom: "; cin >> M.Nom;
  cout << " X = "; cin >> M.X;
  cout << " Y = "; cin >> M.Y;
}

/* Fonction principale (test) */
int main() {
  struct point M1; // variable non initialisée
  struct point M2 = {2, 3.5, "M2"}; // variable initialisée

  // test des fonctions
  point M3;
  initPoint(M3);
  affichePoint(M3);

  // accès aux champs
  cout << "Entrer le nom, puis les deux coordonnées :";
  cin >> M1.Nom >> M1.X >> M1.Y;
  cout <<  M1.Nom << "(" << M1.X << "," << M1.Y << ")" << endl;
  cout <<  M2.Nom << "(" << M2.X << "," << M2.Y << ")" << endl;

  // structures imbriquées
  segment S = { {1, 2.5, "R"}, {-1, 3, "T"}, 0.0};
  segment U;
```

```
  // tableau de structures
  point Tab[5] = {
    {0, 1, "A"},
    {2, 3, "B"},
    {4, 5, "C"},
    {6, 7, "D"},
    {8, 9, "E"}
  };

  for (int i=0; i<5; i++)
    cout << Tab[i].Nom << "(" << Tab[i].X << "," << Tab[i].Y << "); ";
  cout << endl;

  // initialisation d'une structure contenant un tableau de structures
  polyligne P = {3,
                {{1, 2, "A"},
                 {4, 3, "B"},
                 {9, 5, "C"}}
  };

  for (int i=0; i<P.N; i++) cout << P.T[i].Nom << ", ";
  cout << endl;

  // pointeurs
  point *pM = new point; // allocation dynamique
  initPoint(*pM);        // *pM est un 'point'
  affichePoint(*pM);
  delete pM;             // restitution

  point *pM2;
  pM2 = &M2;   // pM2 pointe vers M2
  pM2->X = 123;
  affichePoint(M2); // M2 est modifié (M2.X = 123)


  cout << endl;
  return 1;
}
```

Chapitre 9

Les énumérations, les unions, les champs

9.1 Les énumérations

Une énumération est un type de données permettant de définir un ensemble de constantes entières non signées.

La syntaxe générale pour la déclaration d'une énumération est la suivante :

enum

`enum` *étiquette* `{` $membre_1, membre_2, \cdots, membre_n$ `}` ;

Dans cette déclaration, `enum` est un mot clé obligatoire, *étiquette* est le nom donné à l'énumération, ce qui équivaut à un type, et les termes $membre_k$ sont les valeurs que pourront prendre des variables de ce nouveau type de donnée.

Exemple :

```
enum forme {triangle, cercle, rectangle};
forme F;
...
switch(F) {
  case triangle: ...
  case cercle: ...
  case rectangle: ...
}
```

Les constantes figurant dans les énumérations ont une valeur entière affectée de façon automatique et séquentielle, la première valant 0. Autrement dit, $membre_1$ prend pour valeur 0, $membre_1$ prend pour valeur 1, et ainsi de suite.

Dans l'exemple précédent, les constantes d'énumérations prennent pour valeur :

triangle	0
cercle	1
rectangle	2

Il est possible de forcer ces valeurs initiales par défaut lors de la définition d'une énumération, en explicitant des valeurs différentes. Pour cela, chaque membre recevant une nouvelle valeur doit figurer dans une expression d'affectation ordinaire du type :

$$membre_i = \texttt{N}$$

où N représente une valeur entière signée.

Les constantes restantes qui ne sont pas initialisées explicitement reçoivent la valeur affectée à la constante précédente incrémentée de 1.

Exemple : la déclaration

```
enum forme {triangle=-2, cercle, rectangle=3};
```

provoquera les affectations :

triangle	-2
cercle	-1
rectangle	3

Remarque : dans de nombreux programmes, certaines longues listes de «#define ...» pourraient être avantageusement remplacée par des énumérations. L'utilisation d'énumérations permet un contrôle plus pointu sur le type des variables et constantes utilisées dans le programme.

union

9.2 Les unions

Une union permet à des variables de types différents de se superposer (de partager) une zone commune de mémoire.

La syntaxe générale pour la déclaration d'une énumération est la suivante :

```
union étiquette {
    membre1 ;
    membre2 ;
    ...
    membren
};
```

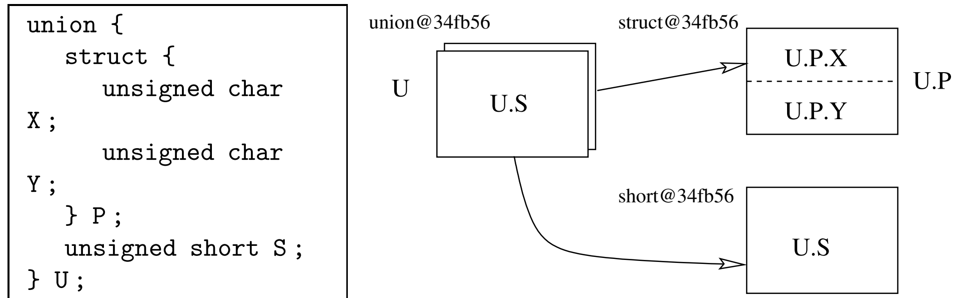


FIG. 9.1 – *Exemple d'union : les deux membres (une structure et un entier court) se recouvrent en mémoire.*

Exemple : considérons l'union définie par

Le programme suivant met en évidence les possibilités d'accès aux différents éléments de mémoire et montre la superposition de données.

small

```
#include <iostream>
#include <iomanip>

union {
  struct {
    unsigned char X;
    unsigned char Y;
  } P;
  unsigned short S;
} U;

int main() {
  unsigned char m;

  cout << "Taille de l'union : " << sizeof(U) << endl;

  U.S = 0x1234;
  cout << "U.S = " << hex << U.S
       << " - U.P.X = " <<  (unsigned)U.P.X
       << " - U.P.Y = " <<  (unsigned)U.P.Y << endl;

  m = U.P.X;
  U.P.X = U.P.Y;
  U.P.Y = m;

  cout << "U.S = " << hex << U.S
       << " - U.P.X = " <<  (unsigned)U.P.X
       << " - U.P.Y = " <<  (unsigned)U.P.Y << endl;
```

```
    return 1;
  }
```

L'exécution donne :

```
 > a.out
Taille de l'union : 2
U.S = 1234  - U.P.X = 34  - U.P.Y = 12
U.S = 3412  - U.P.X = 12  - U.P.Y = 34
 > _
```

Remarque : on note que l'ordre entre les poids forts et les poids faibles de l'entier court ne sont pas respecté par l'ordre induit par la structure. On aurait pu s'attendre à ce que X contienne les poids forts et Y les poids faibles. Cet état est uniquement lié à l'architecture du système sur lequel le programme s'exécute. Dans le cas présent, le programme a été exécuté sur une architecture $Intel$. Or, pour cette architecture, le stockage des entiers se fait en inversant systématiquement les poids forts et les poids faibles. Si on exécute le même programme sur une archirecture $Motorola$, on constate une inversion entre les contenus de $U.P.X$ et $U.P.Y$.

9.3 Les champs de *bits*

Les champs de *bits* permettent d'accéder au niveau du *bit*, dans une variable entière (donc, l'implémentation dépend de l'architecture).

La déclaration se fait comme pour une structure qui ne contient que des membres entiers dont on précise pour chacun le nombre de *bits* qui le caractérise.

La syntaxe générale pour la déclaration d'un champ de *bits* est la suivante :

```
struct étiquette {
   int membre1 :n1 ;
   int membre2 :n2 ;
   ...
   int membrek :nk
} ;
```

Condition à respecter :

$$(\sum_{i=1}^{i=k} n_i) \leq (8 \times \text{sizeof}(\textit{étiquette}))$$

L'utilisation des champs de *bits* est souvent associée aux unions, comme dans l'exemple suivant :

```
#include <iostream>
#include <iomanip>

union {
  struct {
    int X:2;
    int Y:6;
    int c1:8;
    int c2:8;
    int c3:8;
  } B;
  unsigned int I;
}U;

int main(){
  cout << "Taille du champ : " << sizeof(U.B) << endl;

  U.B.c1=0x34;
  U.B.c2=0x56;
  U.B.c3=0x78;
  U.B.Y = 4;
  U.B.X = 2;
  cout << "U.I = " << hex <<  U.I << endl;

  U.I = 0x12345678;
  cout << "U.B.X= " << hex << U.B.X
       << " - U.B.Y= " << U.B.Y
       << " - U.B.c1= " << U.B.c1
       << " - U.B.c2= " << U.B.c2
       << " - U.B.c3= " << U.B.c3 << endl;
  return 1;
}
```

L'exécution donne :

```
 > a.out
Taille du champ : 4
U.I = 78563412
U.B.X= 0 - U.B.Y= 1e - U.B.c1= 56 - U.B.c2= 34 - U.B.c3= 12                    > _
```

Chapitre 10

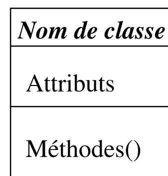
Programmation orientée objet

Classe

10.1 Classes et objets

On associe dans une même unité les données et les fonctions. Les **classes** sont des types composés créés par l'informaticien. Elles contiennent aussi bien des données membres décrivant le type en question que des fonctions implémentant des opérations sur ce type.

Les données membres s'appellent les **attributs** ; les fonctions membres s'appellent des **méthodes.**



Objet

FIG. 10.1 – *Représentation graphique d'une classe*

Les **objets** sont créés ou **instanciés** à partir d'une classe. Pour simplifier, on peut comparer la classe à un type et les objets à des variables.

Exemple : Considérons les déclarations suivantes, en supposant que la classe `fraction` a préalablement été programmée :

```
int i = 9;
fraction w(1,2);
fraction u(5,3);
```

1. `int` est un type, `fraction` est une classe ;
2. `i` est une variable de type `int`, `w` est un objet de type (ou de classe) `fraction` ;

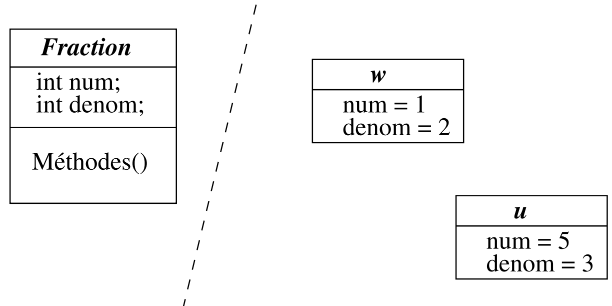


FIG. 10.2 – *Classe et objets.*

Mise en œuvre d'une classe `fraction`

```
#include <iostream>

class fraction {
protected:

    int pgcd(int a, int b) {
        if (b==0) return a;
        return pgcd(b, a%b);
    }

    void simplifie(){
        int commun = pgcd(num,denom);
        num /= commun;
        denom /= commun;
    }

public:
    int num, denom;
    fraction(int n, int d){
        num = n;
        denom = d;
        simplifie();
    }

    void affiche() {
        cout << num << '/' << denom;
    }
};

int main() {
    fraction w(2,4);
    cout << " w = ";
    w.affiche();
    cout << endl;
    return 1;
}
```

fraction
#num: int #denom: int
+fraction(n:int.d:int) +affiche(): void #simplifie(): void #pgcd(a:int.b:int): int

Définitions

→ Une **classe** est déclarée en `C++` avec le mot clé `class` :

```
class nom_de_la_classe {
    ...
    ...
};
```

Dans l'exemple, les méthodes sont également définies entre les accolades de déclaration.

→ Dans la déclaration de la classe, on trouve aussi bien des données (les **attributs**) que des fonctions (les **méthodes**).

→ Une méthode au moins porte le nom de la classe. On l'appelle le **constructeur**. Ainsi, lors de la déclaration

```
fraction w(2,4);
```

le constructeur est exécuté. Le constructeur est en général utilisé pour initialiser les données internes de l'objet.

→ Certaines caractéristiques de l'objet sont déclarées sous le label `public`. Elles peuvent être accessibles en dehors de l'objet. Le constructeur est forcément `public`. Dans l'exemple, la méthode `affiche` est publique, ce qui autorise l'instruction `w.affiche() ;` dans le programme principal.

→ Par contre, la méthode `simplifie` est protégée (`protected`), par conséquent l'instruction `w.simplifie()` placée dans le programme principal conduirait à une erreur de compilation.

→ Le choix `public` ou `protected` pour un attribut ou une méthode découle d'une méthodologie de programmation. Nous y reviendrons avec la notion d'interface.

→ Le type `fraction` est un **type abstrait** : en effet, certains aspects sont dissimulés (tout ce qui est protégé) et donc inaccessibles depuis le progamme principal.

→ `fraction` est une classe. Dans le programme principal, `w` est un **objet** de type `fraction`, ou encore, une **instance** de `fraction`.

Liste d'initialisations

Le constructeur :

```
fraction(int n, int d){
    num = n;
    denom = d;
    simplifie();
}
```

peut également s'écrire :

```
fraction(int n, int d) : num(n), denom(d) {
    simplifie();
}
```

et l'expression «`num(n), denom(d)`» s'appelle une liste d'initialisations.

10.2 Notions d'invariant et d'interface

10.2.1 Spécification et mise en œuvre

Avant de créer un objet, il y a deux étapes essentielles :

- il faut d'abord **spécifier** une classe pouvant servir de «moule» à l'objet. À ce niveau, on se contente de décrire les attributs elt les méthodes.
- Vient ensuite l'implémentation ou **mise en œuvre** : il s'agit d'écrire les algorithmes des différentes méthodes, éventuellement de déterminer des attributs.

interface

Lors de l'étape de **spécification** la question est de savoir ce qu'on va faire de l'objet. On précise les attributs ou méthodes qui seront visibles de l'extérieur. On peut faire l'analogie avec un véhicule : on se contente à ce niveau de dire qu'il faut un volant, un actionneur pour accélérer et un autre pour freiner. Les attributs et les méthodes permettant d'agir sur un objet constituent l'**interface**.

La **mise en œuvre** est à l'opposé une étape de codage. On choisit les algorithmes pour coder les méthodes. Éventuellement, on décide de compléter la spécification avec des méthodes qui resteront cachées à l'intérieur de la classe. Il faut également déterminer les données internes ...

10.2.2 Exemple : la classe `action`

Interface. On décide de créer une classe `action` pour représenter une action monétaire accessible en euros ou en dollars. On doit être capable de modifier ou de lire la valeur de l'action dans l'une quelconque des deux monnaies.

Ceci conduit à l'**interface** suivante :

`action()`	constructeur (valeur = 0.0)
`setEuro(double e) : void`	modificateur de la valeur en €
`setDollar(double d) : void`	modificateur de la valeur en $
`getEuro() : double`	accesseur de la valeur en €
`getDollar() : double`	accesseur de la valeur en $

Mise en œuvre (ou **réalisation**). On choisit de stocker simultanément deux variables :

- `double euro` : valeur € de l'action
- `double dollar` : valeur $ de l'action

Ce qui donne la réalisation suivante :

```
#include <iostream>

class action {
protected:
```

```
    double euro;
    double dollar;
    static double rate = 0.965;
public:
    action() { euro=dollar=0.0; }
    void setEuro(double e) { euro=e; dollar=e*rate; }
    void setDollar(double d) { euro=d/rate; dollar=d; }
    double getEuro() { return euro; }
    double getDollar() { return dollar; }
    void print() {
        cout << "action = " << euro << "E (" << dollar << "$)";
    }
};
```

new

10.3 Création d'un objet avec `new`

Il est possible d'allouer dynamiquement de la mémoire pour un objet avec l'allocateur `new`. Dans ce cas, il y a deux effets consécutifs :

$$\begin{cases} \text{Allocation de mémoire pour l'objet} \\ \text{Exécution du constructeur} \end{cases}$$

Un objet alloué avec `new` doit être détruit avec `delete`.

Exemple :

```
fraction *pF;
  ...
pF = new fraction(22, 7);
  ...
delete pF;
```

10.4 Déclaration et définition séparées

La programmation en langage `C++` impose (dès l'instant ou les programmes nécessitent une compilation séparée avec utilisation d'une même classe dans plusieurs fichiers) de séparer la déclaration de la classe de la définition des méthodes et de l'initialisation des variables. Il est habituel de **déclarer** la classe dans un fichier «en-tête» (*.hpp*) Par exemple, la classe `action` sera déclarée dans le fichier *action.hpp*. Il ne reste plus qu'à définir les méthodes dans un fichier *.cpp*.

Fichier de déclaration `action.hpp`

```
// fichier : action.hpp
// classe : action
// date :  Fri Sep 13 2002
// auteur : C.G.
```

```
#ifndef _action
#define _action

class action {
protected:
    double euro;
    double dollar;
    static double rate;
public:
    action();
    void setEuro(double e);
    void setDollar(double d);
    double getEuro();
    double getDollar();
    void print();
};

#endif
```

→ On y trouve les déclarations des variables et les prototypes des méthodes.

→ Les directives `#ifndef ...#define ...#endif` permettent un compilation conditionnelle de la déclaration de la classe. Ceci est utile si la classe est utilisée dans plusieurs fichiers. À la première lecture, le compilateur défini une variable `_action` et prend acte de la déclaration de la classe. Les fois suivantes, la directive `#ifndef` (pour *if not defined*) est évaluée à *faux* et aucune ligne n'est preise en compte jusqu'au `#endif`.

→ Les variables ne peuvent pas être définies dans la déclaration.

Fichier de définition `action.cpp`

```
// fichier : action.cpp
// classe : action
// date :  Fri Sep 13 2002
// auteur : C.G.

#include "action.hpp"
#include <iostream>

double action::rate = 0.965;

action::action() { euro=dollar=0.0; }
void action::setEuro(double e) { euro=e; dollar=e*rate; }
void action::setDollar(double d) { euro=d/rate; dollar=d; }
double action::getEuro() { return euro; }
double action::getDollar() { return dollar; }
void action::print() {
    cout << << euro << "E (" << dollar << "$)";
}
```

→ La déclaration de la classe doit figurer en tête, d'où la directive :
`#include "action.hpp"`

→ Chaque définition de méthode ou initialisation de variable est repérée par le préfixe «`nom_de_la_classe : :`», exemple :
`void action : :print() { ...}`

Fichier de test `actionT.cpp`

```
#include <iostream>
#include "action.hpp"

int main() {
    action a;
    a.setEuro(100);
    a.print();
    cout << endl;
    return 1;
}
```

La compilation de ce programme se fait par la commande :

```
g++ actionT.cpp action.cpp
```

On remarque que le fichier *action.hpp* est donc ouvert deux fois. Grâce aux directives de compilation, la classe ne sera déclarée qu'une seule fois.

10.5 Les constructeurs

Pour une classe donnée, il peut y avoir plusieurs constructeurs. Il est courant d'en trouver au moins trois :

le constructeur vide : il ne reçoit aucun paramètre. Il peut éventuellement initialiser les attributs de l'objet à des valeurs «par défaut». **Il est obligatoire pour pouvoir créer des tableaux d'objets.**

un constructeur paramétré qui reçoit en argument des valeurs pour initialiser les attibuts.

le constructeur par recopie d'objet : il reçoit un objet de la même classe qui est dupliqué.

Exemple : on complète la classe `action` : Dans *action.hpp* :

```
class action {
    ...
public:
    action();
    action(double e);
    action(const action& a);
    ...
};
```

Dans *action.cpp* :

```
  ...
action::action() { euro=dollar=0.0; }
action::action(double e) { setEuro(e); }
action::action(const action& a) { euro=a.euro; dollar=a.dollar; }
  ...
```

10.6 Surcharge des opérateurs

La surcharge des opérateurs est une particularité du langage `C++`. Ceci signifie que l'on peut redéfinir (ou réécrire) un opérateur (addition, soustraction, ...) pour l'adapter à des objets. L'idée est de rendre plus aisée et plus intuitive la lecture des programmes utilisateurs.

Quelques règles d'utilisation :

→ Il faut garder l'esprit de l'opérateur, `+` pour additionner ...

→ Les opérateurs `' : :'`, `'.'`, `'.*'`, `' :? '`, `sizeof` ne sont pas «surchargeables»,

→ les opérateurs `'()'`, `[]`, `'->'`, `new`, `delete` ne peuvent être surchargés que comme des fonctions membres.

→ Il n'est pas possible
- de changer la priorité d'un opérateur,
- de changer son associativité,
- de changer son arité (unaire, binaire, ternaire),
- de créer de nouveaux opérateurs.

Lorsqu'un opérateur est appelé, l'opérateur `'+'` par exemple, le compilateur génère un appel à la fonction `operator+`.

Ainsi, `Z = X+Y` ⇔ `Z = X.operator+(Y)`

Surcharger un opérateur *<op>* revient donc à surcharger la fonction correspondante `operator`*<op>*.

Attention : si X et Y ne sont pas du même type, il n'y a pas commutativité de l'opérateur.

Exemple. Ajoutons à la classe `action` les opérateurs `'*'` de multiplication par un scalaire et `'='` d'affectation.

Dans *action.hpp* :

```
class action {
    ...
public:
    ...
  action operator*(double t);
  action operator=(const action &a);
};
```

Dans *action.cpp* :

```
  ...

action action::operator*(double t) {
  action retour(euro*t);
  return retour;
}

action action::operator=(const action &a) {
  action retour(a);
  return retour;
}
```

L'utilisation des opérateurs ainsi redéfinis est illustré dans l'exemple suivant :

```
action a(100);
action b = a*1.05; // et non 1.05*a !!
cout << "b = "; b.print();
cout << endl;
```

10.7 Redirection des flux standards `cin` et `cout`

friend

10.7.1 Fonctions amies (`friend`)

Une fonction amie d'une classe X est une fonction qui, bien que n'étant pas fonction membre de X, dispose des droits d'accès complets à tous les membres de X, qu'ils soient publics ou non.

10.7.2 Application à la redirection des flux

La fonction `operator<<` surcharge l'opérateur `'<<'` pour envoyer des données vers un flux de type `ostream`. Elle renvoie une référence sur la variable de type `ostream` qu'elle a reçue, de façon à pouvoir le cascader dans une même instruction.

Exemple :

	`cout << a << b;`	
⇔	`operator<<(cout, a)  <<  b;`	Affichage de `a`
⇔	`cout <<  b;`	La fonction renvoie `cout`
⇔	`operator<<(cout, b);`	Affichage de `b`
⇔	`cout;`	Valeur finale de l'expression.

Pour appliquer le flux de sortie à la classe `action`, il nous suffit donc de surcharger la fonction `operator<<` et d'en faire une fonction amie de la classe `action`. Le code se modifie ainsi :

1. dans *action.hpp*, on ajoute la ligne

```
class action {
  ...
 friend ostream &operator<<(ostream &os, action &a);
};
```

2. et dans *action.cpp* :

```
ostream &operator<<(ostream &os, action &a) {
  a.print();
}
```

L'affichage d'un objet de classe `action` peut se faire désormais directement selon le modèle :

```
action a(100);
cout << "a = " << a;
```

La redirection du flux d'entrée se fait selon le même principe. On trouvera le code complet de la classe `action` dans la section 10.9

10.8 Objet et allocation dynamique de mémoire

Position du problème

Il est fréquent que certains attributs d'une classe doivent être alloués dynamiquement. Par exemple, si on souhaite mémoriser le nom de l'action dans la classe précédente, il est plus intéressant d'ajouter un attribut de type `char*` que l'on initialise en fonction de la longueur de la chaîne de caractères plutôt que de réserver un tableau de taille constante.

Cependant, lorsque l'objet «disparait» (devient inaccessible au programme) il faut restituer la mémoire allouée.

Destructeur

Le **destructeur** est le mécanisme qui permet de résoudre le problème précédent. C'est une méthode spéciale qui est exécutée systématiquement dès qu'un objet disparait. En `C++`, cette méthode porte le nom de la classe précédé du caractère `'~'`. Elle ne reçoit rien et ne renvoie rien. Pour la classe `action`, le prototype du destructeur de la classe est donc

```
~action();
```

Nous ne l'avons pas écrit pour le moment car il n'était pas nécessaire. En fait, il existe un destructeur «par défaut» qui ne fait rien. Ne rien mettre n'est donc pas une faute.

Exemple

On décide de mémoriser dans un objet `action` le nom de l'action. Afin d'optimiser les ressources, on choisit d'allouer dynamiquemement la zone de stockage pour la chaîne de caractères. Un nouvel attribut apparaît dans la partie protégée :

```
char *nom;
```

On créée un nouveau constructeur :

```
action(double e, char *str);
```

chargé de l'allocation mémoire. Son code est :

```
action::action(double e, char* str) {
  setEuro(e);
  nom = new char[strlen(str) + 1];
  strcpy(nom,str);
}
```

Il est important d'initialiser le pointeur `nom` également dans les autres constructeurs. En effet, le destructeur étant unique, il faut pouvoir distinguer un pointeur intialisé d'un pointeur non initialisé. Il est recommandé de mettre à 0 le pointeur `nom` dans les autres constructeurs. (`nom = NULL ;`). (Voir le code complet de l'exemple au paragraphe 10.9). Cette précaution étant prise, on peut écrire le code du destructeur :

```
action::~action() {
  if (nom != NULL) delete[] nom;
}
```

Exécution du destructeur

Deux cas seulement existent :

1. **L'objet est déclaré de manière statique** dans un blocd'instructions : le destructeur est appelé à l'accolade fermande du bloc.
2. **L'objet est alloué dynamiquement avec `new`** : le destructeur est appelé avec l'opérateur `delete`.

```
action *p ;
{
     action a(100, "eurotunnel") ;   ← constructeur de a
     p = new action(50, "bic") ;     ← constructeur de (*p)
     ...
}                                    ← le destructeur de a est appelé
delete p ;                           ← le destructeur de (*p) est appelé
```

10.9 Code complet de la classe action

Fichier *action.hpp*

```
// fichier : action.hpp
// classe : action

#ifndef _action
#define _action

#include <iostream>

class action {
protected:
  double euro;
  double dollar;
  static double rate;
  char *nom;
```

```
  void print();
public:
  action();
  action(double e);
  action(double e, char *str);
  action(const action& a);
  ~action();
  void setAction(double e, char *str);
  void setEuro(double e);
  void setDollar(double d);
  double getEuro();
  double getDollar();
  action operator*(double t);
  action operator=(const action &a);
  friend ostream &operator<<(ostream &os, action &a);
  friend istream &operator>>(istream &is, action &a);
};

#endif
```

Fichier *action.cpp*

```
// fichier : action.cpp
// classe : action

#include "action.hpp"
#include <iostream>
#include <string.h>

double action::rate = 0.965;

// constructeurs
action::action() { euro=dollar=0.0; nom=NULL;}
action::action(double e) {
  setEuro(e);
  nom = NULL;
}
action::action(double e, char* str) {
  setEuro(e);
  if (str != NULL) {
    nom = new char[strlen(str) + 1];
    strcpy(nom,str);
  }
}

action::action(const action& a) {
  euro=a.euro;
  dollar=a.dollar;
  if (a.nom==NULL) nom=NULL;
  else {
    nom = new char[strlen(a.nom)+1];
```

```
    strcpy(nom,a.nom);
  }
}

// destructeur
action::~action() {
  if (nom != NULL) delete[] nom;
}

// modificateurs
void action::setAction(double e, char* str) {
  setEuro(e);
  if (nom!=NULL) delete[] nom;
  nom = new char[strlen(str)+1];
  strcpy(nom, str);
}
void action::setEuro(double e) { euro=e; dollar=e*rate; }
void action::setDollar(double d) { euro=d/rate; dollar=d; }

// accesseurs
double action::getEuro() { return euro; }
double action::getDollar() { return dollar; }
void action::print() {
  if (nom != NULL) cout << nom << ": ";
  cout << euro << "E (" << dollar << "$)";
}



// opérateurs
action action::operator*(double t) {
  action retour(euro*t, nom);
  return retour;
}

action action::operator=(const action &a) {
  action retour(a);
  return retour;
}

ostream &operator<<(ostream &os, action &a) {
  a.print();
  return os;
}

// fonctions amies
istream &operator>>(istream &is, action &a) {
  double e;
  char b[128];
  cout << "  -> Valeur en euro: ";
  is >> e;
  cout << "  -> Nom : ";
```

```
  cin >> b;
  a.setAction(e,b);
  return is;
}
```

Programme de test *actionT.cpp*

```
#include <iostream>
#include "action.hpp"

int main() {
    action a(100, "eurotunnel");
    cout << "a = " << a  << endl;
    action b(a);
    cout << "b = " <<  b << endl;

    action d = a*1.05;
    cout << "d = " << d << endl;
    cout << "Nouvelle action :\n";
    action x;
    cin >> x;
    cout << "x = " << x << endl;

    return 1;
}
```

10.10 Le constructeur par recopie

La présence de ce constructeur n'est pas toujours obligatoire, mais son absence peut occasionner des erreurs difficiles à détecter dans bien des cas.

10.10.1 Classe sans le constructeur par recopie

Considérons l'exemple suivant dans lequel une class *truc* réalise dans son constructeur une allocation de mémoire, et par conséquent dans son destructeur une restitution de cette même mémoire.

```
class truc {
public:
  int *Tab;
  int f;
  truc(){ Tab = new int[1024]; f=1;}
  ~truc() { delete Tab; f=0;}
};

void fonction(truc t){}

int main() {
  truc tr;
```

```
    cout << "tr.f = " << tr.f << endl;
    fonction(tr);
    cout << "tr.f = " << tr.f << endl;
  }
```

L'exécution du programme donne :

```
tr.f = 1
Erreur de segmentation
```

ce qui prouve bien que lors du deuxième affichage, on utilise un objet *tr* qui n'existe plus !

Explication la variable *tr* est passée en argument à la fonction *fonction*, et puisqu'il n'y a pas de constructeur par recopie, c'est l'original qui est utilisé. À la fin de la fonction (qui paraît ne rien faire), il y a destruction du paramètre *t* qui est un alias de la variable originale.

Autre exemple

```
truc t1;        // construction de t1
if (...) {
  truc t2 = t1; // t2 est un autre nom pour t1
  ...
} // ---> !!! destruction de t2 donc de t1 !!!
cout << t1.f; // faute, cat t1 n'existe plus
```

10.10.2 Classe avec le constructeur par recopie

L'exemple devient :

```
class truc {
public:
  int *Tab;
  int f;
  truc(){ Tab = new int[1024]; f=1;}
  truc(truc& tt) {
     Tab = new int[1024]; f=1;
     for (int i=0; i<1024; i++) Tab[i] = tt.Tab[i];
  }
  ~truc() { delete Tab; f=0;}
};
```

Ici, chaque fois qu'un objet *truc* est affecté dans une nouvelle variable, ou passé en argument à une fonction, il conserve son intégrité. Il y a une duplication des données, chaque objet dispose de son espace propre. Avec cette version, l'exécution donne :

```
tr.f = 1
tr.f = 1
```

Il est donc primordial de coder ce constructeur systématiquement. Notons que l'exemple donné ici conduit à un problème explicite, du fait de l'allocation de mémoire et du codage du destructeur.

Autre exemple

```
truc t1;        // construction de t1
if (...) {
  truc t2 = t1; // t2 est un nouvel objet
  ...
} // ---> destruction de t2 mais pas  de t1
cout << t1.f; // OK
```

Chapitre 11

L'héritage

11.1 Principes

L'**héritage** est un mécanisme qui permet de définir une classe B *à partir* d'une classe A.

- A est la classe de base.
- B est la classe dérivée.
- La classe B est une *spécialisation* de la classe A.

Intérêts

- La classe dérivée «récupère» les propriétés et opérations publiques (et éventuellement protégées) de la classe de base.
- On peut lui ajouter de nouveaux membres.
- On peut redéfinir les méthodes.
- C'est un principe structurant pour les applications.
- Économie de temps dans le développement.

11.2 L'héritage public

C'est le type d'héritage le plus répandu. Il existe également les héritages protégés et privés, plus marginaux, qui seront présentés en fin de chapitre pour information.

Caractéristiques de l'héritage public :

→ il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.

→ c'est la forme la plus courante d'héritage, car il permet de modéliser les relations «`Y est une sorte de X`» ou «`Y est une spécialisation de la classe de base X`».

Accès aux attributs et méthodes

L'accès aux attributs de la classe dérivée se fait conformément au tableau suivant. On introduit pour la notion d'héritage le label `private` qui permet

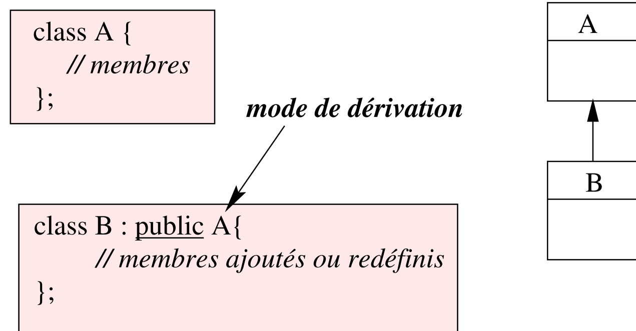


FIG. 11.1 – *Héritage public*

d'interdire l'héritage d'une partie de la classe de base.

`A`	`B`	**Autre classe**
`public`	`public`	`public`
`private`	*non accessible*	*non accessible*
`protected`	`protected`	*non accessible*

Codage `C++`

```
class A {
  // Attributs
private:

protected:

public:

  // Constructeurs - méthodes

private:

protected:

public:
  A();

};
```

```
class B : public A {
  // Attributs
private:

protected:

public:

  // Constructeurs - méthodes

private:

protected:

public:
  B();

};
```

Redéfinition des membres

On redéfini un membre dans une classe dérivée si on lui donne **le même nom** que dans la classe de base.

Dans l'exemple ci-contre, il y a deux fonctions `f2()`, mais il sera possible de les différencier avec l'opérateur `::` de résolution de portée.

```
class X {
public:
  void f1();
  void f2();
};

class Y : public X {
public:
  void f2();
  void f3();
};

void Y::f3() {
  X::f2(); // f2 de la classe X
  f1(); // appel de f1 de la classe X
  f2(); // appel de f2 de la classe Y
}
```

11.3 Exemple : les classes `position` et `point`

On considère les deux classes définies par le diagramme de la figure 11.2. La classe `point` spécialise la classe `position` par l'ajout d'un nom.

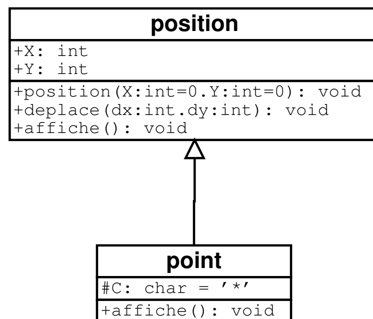


FIG. 11.2 – *Classes `position` et `point`*

La réalisation donne 4 fichiers : deux fichiers d'en-tête (*.hpp*) et deux fichiers de mise en œuvre (*.cpp*)

1. ***position.hpp***

```
#ifndef _position
#define _position

class position {
public:
  int X;
  int Y;
  position(int x=0, int y=0);
```

```
  void deplace(int dx, int dy);
  void affiche();
};
#endif
```

2. ***position.cpp***

```
#include <iostream>
#include <iomanip>
#include "position.hpp"

position::position(int x, int y) : X(x),Y(y) {}
void position::deplace(int dx, int dy) {
  X+=dx;
  Y+=dy;
}
void position::affiche() {
  cout << '(' << X << ',' << Y << ')';
}
```

3. ***point.hpp***

```
#ifndef _point
#define _point
#include "position.hpp"

class point : public position {
private:
  char C;   // caractère de traçage
public:
  point(int x, int y, char c);
  void affiche(); // affiche le point
};
#endif
```

4. ***point.cpp***

```
#include <iostream>
#include "point.hpp"

point::point(int x, int y, char c) : position(x,y), C(c) {}
void point::affiche() {
  cout << C;
  position::affiche();
}
```

   Le constructeur de la classe `point` fait ici explicitement appel au constructeur de la classe de base. L'appel se fait dans la liste d'initialisation.

Les classes peuvent être utilisées dans un programme de test :

```
#include <iostream>
#include <iomanip>
#include "point.hpp"
#include "position.hpp"

int main() {
  point P(1,2,'P');
  P.affiche();
  cout << endl;
  P.deplace(-1, -2);
  P.affiche(); cout << endl;
  P.position::affiche(); cout << endl;
  return 1;
}
```

Exécution

```
P(1,2)
P(0,0)
(0,0)
```

Remarques

- Un objet de la classe `point` accède bien à la méthode `deplace` de la classe `position`, ce qui montre bien que si une m"thode n'est pas redéfinie, alors, c'est la méthode de la classe de base qui est prise en compte ;
- La méthode `affiche` est correctement appelée et un objet de la classe `point` peut appeler soit `affiche()` ou soit `position : :affiche()`.

11.4 Héritage et constructeurs/destructeurs

Les règles suivantes s'appliquent :

- Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.
- Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.
- Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, il faut employer une liste d'initialisation (comme dans l'exemple précédent).
- L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

L'exemple suivant illustre l'ordre d'exécution des constructeurs et des destructeurs d'objets :

```
class Vehicule {
public:
   Vehicule() { cout<< "Vehicule" << endl; }
   ~Vehicule() { cout<< "~Vehicule" << endl; }
};

class Voiture : public Vehicule {
public:
   Voiture() { cout<< "Voiture" << endl; }
   ~Voiture() { cout<< "~Voiture" << endl; }
};
```

```
int main() {
  Voiture x;
  // ...
}
```

Exécution

```
% a.out
Vehicule
Voiture
~Voiture
~Vehicule
% _
```

11.5 Les autres formes d'héritages

Le `C++` prévoit deux autres formes d'héritage en plus de l'héritage public. Il s'agit de l'**héritage privé** et de l'**héritage protégé**. Les règles qui s'appliquent alors sont représentés par les schémas des figures 11.4 et 11.3.

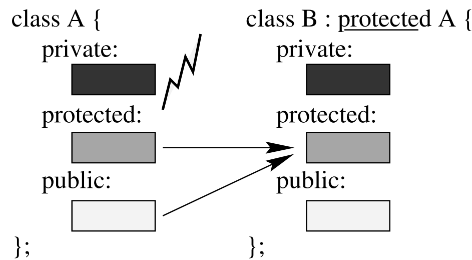


Fig. 11.3 – *Héritage protégé*

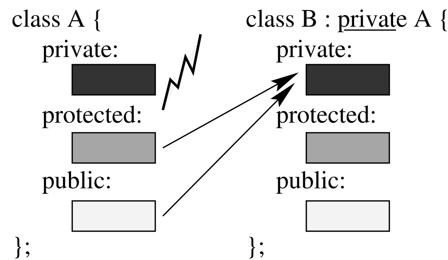


Fig. 11.4 – *Héritage privé*

Dans l'héritage privé, la classe dérivée n'accède qu'aux champs publics et protégés de la classe de base. Ces champs deviennent privés pour la classe dérivée.

Ces deux formes d'héritage ne sont que peu utilisées. Elles n'ont pas d'équivalent dans des langages comme `java`. Lorsqu'on parlera d'**héritage**, il s'agira – sauf avis contraire – d' **héritage public**.

11.6 Le polymorphisme

Nous avons vu que l'**héritage** permet de réutiliser du code d'une classe de base dans les classes dérivées. Le **polymorphisme** rend possible l'utilisation d'une même instruction pour appeler des méthodes différentes dans la hiérarchie des classes. En `C++` le polymorphisme est mis en œuvre par l'utilisation des **fonctions virtuelles**. Prenons un exemple autour des classes `position` et `point`. On considère le nouveau programme de test :

```
#include <iostream>
#include <iomanip>
#include "point.hpp"
#include "position.hpp"


void f(position& p) {
  p.affiche();
  cout << endl;
}

int main() {
  point M(1,2,'P');
  position p(3,4);

  f(M);
  f(p);

  return 1;
}
```

Exécution

```
 > a.out
(1,2)
(3,4)
 >
```

Lors de l'appel `f(M)`, il y conversion du type `point` en type `position`, et l'affichage ne s'adapte pas au véritable type du paramètre. C'est toujours la méthode `affiche` de la classe de base qui est appelée.

Solution : si on souhaite que la fonction `f` «s'adapte», il suffit de modifier les fichiers *position.hpp* et *point.hpp* en remplaçant dans chacun de ces fichiers la ligne

```
void affiche(); // affiche le point
```

par :

```
virtual void affiche(); // affiche le point
```

Le mot clé `virtual` indique qu'une fonction est «virtuelle» Ce qui donnera l'affichage attendu :

```
 > a.out
P(1,2)
(3,4)
 >
```

Chapitre 12

Les listes chaînées

12.1 Limitation des représentations séquentielles

Nous avons vu précédemment un seul type pour représenter les collections de données : le tableau. L'utilisation des tableaux induit une gestion aisée des données en mémoire.

Rappel : un tableau est une représentation **séquentielle** des données, ce qui signifie que les données (toutes de même type) sont rangées dans des emplacements mémoire **contigues** et directement accessible par indexage.

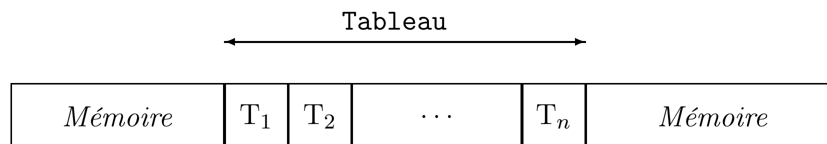


FIG. 12.1 – *Encombrement d'un tableau en mémoire.*

Exemple

```
#include <iostream>
#include <iomanip>

struct point {
  double X,Y;
  char Nom[8];
};

point T[100] = { {0., 0., "O"},
                 {1., 0., "A"},
                 {0., 1., "B"}
};

int main() {
  cout << "Taille du tableau = " <<  100*sizeof(point) << endl;
```

```
    return 1;
  }
```

Dans le programme précédent, on peut imaginer que le tableau `T` a été prévu de taille 100 au cas où ...Ceci conduit à une encombrement de la mémoire de 2400 octets.

Avantages de la représentation séquentielle

→ Simplicité du rangement.

→ Allocation automatique de la mémoire.

→ Accès aléatoire aux données avec la syntaxe `T[i]` (le temps d'accès à une donnée ne dépend pas de son emplacement.

Inconvénient

Il faut connaître à l'avance le nombre d'éléments. Lorsqu'on ne connaît pas à l'avance le nombre d'éléments à mémoriser, l'utilisation d'un tableau trop petit conduit à un «plantage» du programme. La solution consistant à prendre un tableau surdimensionné est acceptable, mais n'optimise pas la gestion de la mémoire. Cette solution est pourtant très souvent retenue lors de la lecture de chaînes de caractères.

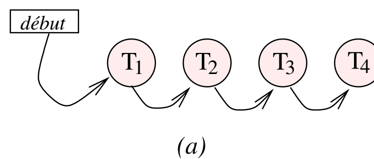
Si cela ne pose pas de problème lorsque la taille de l'élément n'est que d'un octet, elle peut être mise en défaut lorsque l'élément est une structure de donnée plus encombrante.

Il convient donc de rechercher une solution plus souple pour organiser les collections de données en nombre variable et imprévisible.

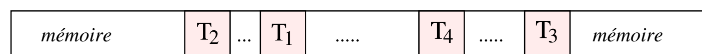
12.2 La liste chaînée

12.2.1 Idée de base :

- Chaque élément de donnée est alloué individuellement.
- Les éléments de données sont reliés entre eux pour former l'ensemble.



(a)



(b)

FIG. 12.2 – *Représentation symbolique (a) et physique d'une liste chaînée.*

Les éléments de la collection de données occupent des emplacements en général non contigus en mémoire. Il faut pouvoir identifier les éléments. Pour cela, il suffit de connaître l'identification du premier élément, et que chaque élément connaisse l'identité de son successeur (méthode récurrente).

Définition : On dit qu'une liste chaînée est *linéaire* lorsque chaque élément de la chaîne connaît l'identificateur d'un suivant et d'un seul. On appelle souvent cette organisation une **liste linéaire simplement chaînée** (il n'y a qu'un seul sens de parcours).

12.2.2 Identification d'un objet

L'identificateur est matérialisé en langage `C++` par un *pointeur*. C'est une **donnée** qui contient l'adresse physique de l'élément. L'opérateur `new` permet d'allouer un espace physique en mémoire et renvoie l'identificateur de l'élément créé (son adresse).

Exemple : création dynamique d'un objet de type `struct point`

```
point *P;  /* P est l'identificateur */
/* Initialisation de P :*/
P = new point;
```

L'exécution de l'opérateur `new` produit deux résultats :

1. Allocation de mémoire pour un nouvel élément.
2. Retour d'un identificateur pour l'objet créé.

Opérations avec un identificateur : on peut

- l'affecter dans une variable ;
- s'en servir pour accéder aux champs de l'élément (opérateur `->`) ;
- le passer en paramètre à une fonction ;
- l'utiliser comme retour dans une fonction.

12.2.3 Chaînage

Chaque élément doit donner un accès à son successeur dans la liste. Il convient donc de compléter la définition du type de base pour qu'il contienne cette information «administrative» : l'identificateur du suivant.

Exemple : Modification du type `listePoint` en vue de réaliser un chaînage :

```
struct listePoint {
  point P;
  listePoint *suivant;
};
```

`suivant` est une variable de type `point *`, c'est donc bien un identificateur d'élément.

On peut alors utiliser les identificateurs pour «attacher» les objets les uns aux autres et produire des chaînes d'objets. Le champ `suivant` de chaque élément contiendra l'identificateur de son successeur dans la chaîne. Le dernier élément n'a pas de successeur. Le champ `suivant` contiendra alors l'identificateur nul `NULL`.

Remarque : il faut une variable de type *identificateur d'élément* pour marquer le début de la chaîne (voir figure 12.3).

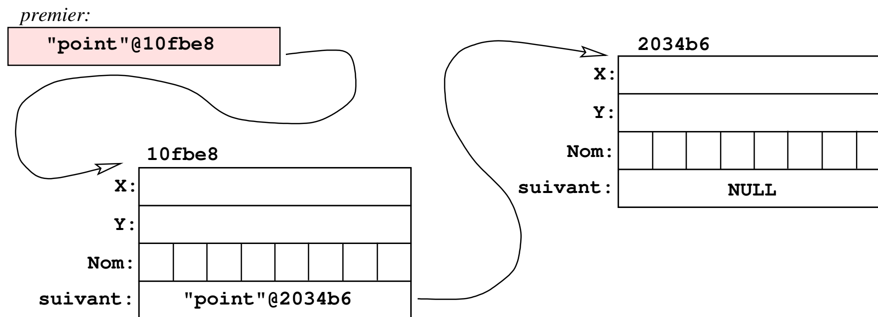


FIG. 12.3 – *Exemple de chaînage avec deux éléments. La variable* `premier` *marque le début de la chaîne.*

Parcours de la liste chaînée : il suffit de faire prendre à une variable de type identificateur les valeurs successives des identificateurs des éléments de la liste chaînée.

Dans le programme suivant, une liste est créée à partir d'éléments figurant dans un tableau, puis la liste est affichée, et enfin, la mémoire est restituée.

```
#include <iostream>
#include <iomanip>


// ----------- type 'point' ---------------
struct point {
  double X,Y;
  char Nom[8];
};

point T[3] = { {0., 0., "O"},
               {1., 0., "A"},
               {0., 1., "B"}
};
```

```
// point chaîné:
// ----------- type 'pointChaine' ----------
struct pointChaine {
    point P;
    pointChaine *suivant;
    pointChaine(point M) { P = M; }
};

// déclaration d'une liste vide:
pointChaine *premier=NULL;


int main() {

  // transformation tableau --> liste chaînée
  for (int i=0; i<3; i++) {
      pointChaine *lp = new pointChaine(T[i]);
      lp->suivant = premier;
      premier = lp;
  }

  // affichage de la liste
  for (pointChaine *lp=premier; lp != NULL; lp=lp->suivant)
      cout << lp->P.Nom << '(' << lp->P.X << ',' << lp->P.Y << ')' << endl;

  // restitution de la mémoire
  pointChaine *lp1=premier;
  while (lp1 != NULL) {
      pointChaine *lp2=lp1->suivant;
      delete lp1;
      lp1 = lp2;
  }

  return 1;
}
```

Trace d'exécution :

```
B(0,1)
A(1,0)
O(0,0)
```

Parcours récursif : dans notre exemple, le parcours de la liste est réalisé par une boucle `for`. Il existe une autre méthode : c'est le **parcours récursif**. S'il n'est pas indispensable pour une liste linéaire (un seul suivant), il sera obligatoire pour listes non linéaires (arbres, graphes ...).

1. **Fonctions récursives d'affichages :** il y a deux possibilités, afficher l'élément de tête, puis le reste de la liste (`afficheListe1` , ou bien appeler d'abord la fonction d'affichage, puis afficher l'élément de tête `afficheListe2` :

```
void afficheListe1(pointChaine *pc) {
```

```
    if (pc == NULL) return;
    // 1 - affiche l'élément en tête:
    cout << pc->P.Nom << '(' << pc->P.X << ',' << pc->P.Y << "); ";
    // 2 - affiche la liste privée de son premier élément
    afficheListe1(pc->suivant);
}

void afficheListe2(pointChaine *pc) {
    if (pc == NULL) return;
    // 1 - affiche la liste privée de son premier élément
    afficheListe2(pc->suivant);
    // 2 - affiche l'élément en tête:
    cout << pc->P.Nom << '(' << pc->P.X << ',' << pc->P.Y << "); ";
}
```

2. **Appel des fonctions** : elles reçoivent comme argument un pointeur sur l'élément de tête.

```
// affichage récursif
cout << endl << "Affichage récursif (1): " << endl;
afficheListe1(premier);
cout << endl << "Affichage récursif (2): " << endl;
afficheListe2(premier);
cout << endl;
```

3. **Exécution** : on obtient bien un affichage dans chaque sens.

```
Affichage récursif (1):
B(0,1); A(1,0); O(0,0);
Affichage récursif (2):
O(0,0); A(1,0); B(0,1);
```

12.2.4 Quelques possibilités des listes chaînées

On peut ajouter des liens pour parcourir plus efficacement la collection d'éléments :

→ le **chaînage double** permet de parcourir la liste dans les deux sens. On garde en mémoire un pointeur `premier` qui indique le début de la chaîne et un pointeur `dernier` qui marque la fin (figure 12.4) . L'exemple de déclaration précédent devient :

```
struct pointChaine {
    point P;
    pointChaine *suivant;
    pointChaine *precedant;
    pointChaine(point M) { P = M; }
};

pointChaine *premier=NULL, *dernier=NULL;
```

→ Le **chaînage double** peut également être utilisé pour parcourir une liste avec deux vitesses de parcours différentes : un pointeur `suivant` permet de visiter les éléments un à un, et un pointeur `groupeSuivant` permet de passer d'un élément à un autre qui est beaucoup plus éloigné

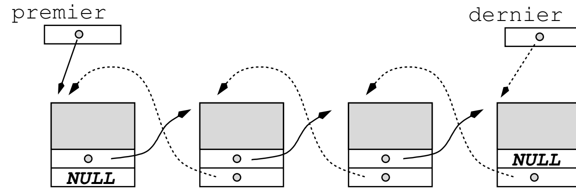


FIG. 12.4 – *Utilisation du chaînage double pour parcourir une liste dans les deux sens.*

dans la liste. Par exemple, le pointeur `suivant` réalise le parcours de tout un dictionnaire de mots, alors que le pointeur `groupeSuivant` fait passer de tous les mots commençant par la lettre *'a'* au premier mot commençant par la lettre *'b'* et ainsi de suite ...(figure 12.5). Pour trouver un mot, il suffit de parcourir «rapidement» la liste pour trouver le premier mot commençant par le premier caractère du mot recherché, puis de procéder à un parcours sur chaque élément.

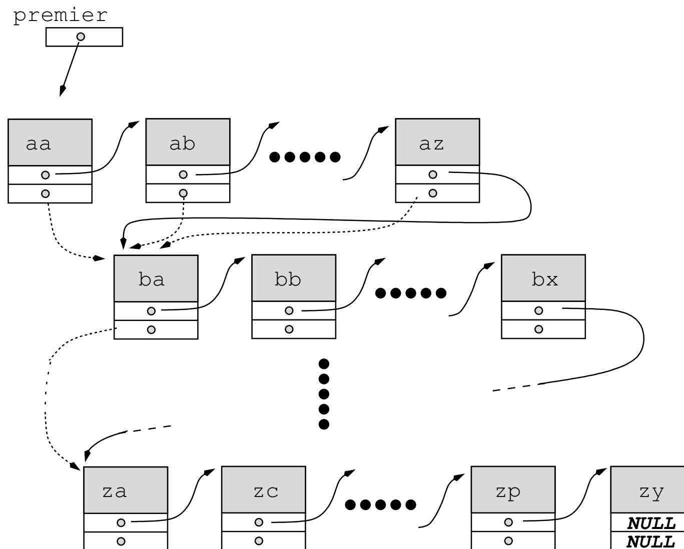


FIG. 12.5 – *Utilisation du chaînage double pour parcourir une liste avec deux «vitesses» de parcours.*

→ On peut construire des **structures de données complexes** :arbres, graphes (voir figure 12.6). Les algorithmes récursifs sont recommandés pour parcourir ce type de listes.

12.3 Exemple complet de mise en œuvre

On encapsule dans une classe `listePoint` une liste d'éléments de type `point` définis précédemment. L'interface minimum pour une gestion d'une

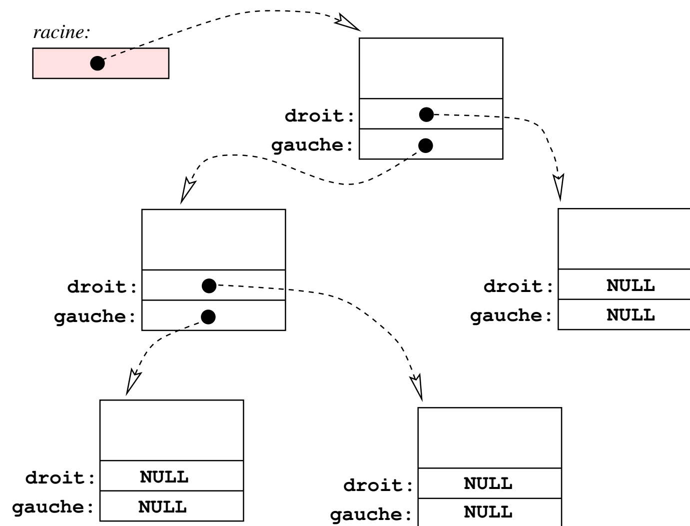


FIG. 12.6 – *Utilisation du chaînage pour une structure non linéaire en arbre binaire.*

liste doit permettre :

- de créer un liste vide,
- d'ajouter un élément à un rang `n`,
- de supprimer l'élément de rang `n`,
- de compter le nombre d'éléments de la liste,
- de savoir si la liste est vide.

La mise en œvre conduit au diagramme des classes de la figure 12.7 (page 100). Les structures `point` et `pointChaine` sont les mêmes que précédemment.

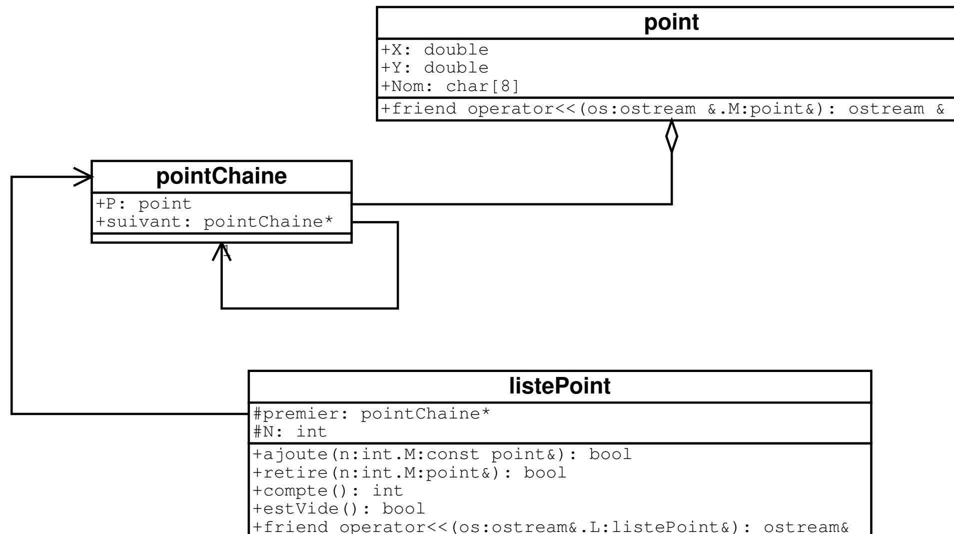


FIG. 12.7 – *Diagramme des classes utilisées pour réaliser la liste chaînée.*

12.3.1 Opérations sur les listes chaînées linéaires

Liste vide

La liste repérée par l'identificateur `premier` est vide ⇔ `premier == NULL`

```
int EstVide(point *p) {  return p == NULL; }
```

Création de la liste et ajout d'un élément

La liste repérée par l'identificateur `premier` est créée lorsqu'on y insère le premier élément. La fonction `Ajouter` reçoit les informations `(x, y, nom)`, créée un élément de type `struct point` et l'ajoute à la liste.

Dans la réalisation suivante, l'élément est ajouté en tête de liste. L'identificateur `premier` prend donc une nouvelle valeur.

```
void Ajouter(double x, double y, char *nom) {
  struct point *M;
  M = new point;
  M->X = x; M->Y = y; strcpy(M->Nom, nom);
  M->suivant = premier;
  premier = M;
}
```

Lorsque l'ajout de l'élément ne se fait pas en tête, les opérations sont plus complexes. À titre d'exemple, voici une fonction qui ajoute un élément à la seconde place. Ceci n'est possible que pour une liste non vide.

```
int  AjouterEnSecond(double x, double y, char *nom) {
  struct point *M,*S;
  if (EstVide(premier)) {
    Ajouter(x,y,nom); /* Ajout en tête */
    return 0; /* Ajout impossible en seconde position */
  }
  else {
    M = new point;
    M->X = x; M->Y = y; strcpy(M->Nom, nom);
    S = premier->suivant; /* S identifie l'ancien second élément */
    M->suivant = S;
    premier->suivant = M;
    return 1; /* Ajout en second effectué */
  }
}
```

Parcours récursif d'une liste

Les fonctions récursives s'adaptent bien au collections de données en listes chaînées. Le principe est d'appliquer la fonction à l'élément de tête, puis la fonction s'appelle récursivement pour la liste privée de son premier élément. Le processus s'arrête lorsque la liste est vide.

L'affichage des éléments de la liste peut se régler de cette manière.

```
void Afficher(struct point *p) {
  if EstVide(p) return;
  else {
    cout << p->Nom << " : " << p->X ',' << p->Y;
    Afficher(p->suivant);
  }
}

....
 Afficher(premier); /* Affichage de toute la liste */
....
```

Recherche d'un élément dans une liste

Nous devons tout d'abord disposer d'une fonction qui détermine si un élément de la liste repéré par son identificateur est conforme à ce que l'on cherche. C'est l'objet de la fonction `Compare`.

La recherche peut ensuite s'effectuer de manière itérative (`Recherche`) ou récursive (`RechercheR`). Lea appels diffèrent légèrement pour ces deux fonctions.

```
int Compare(double x, double y, char *nom, struct point *M){
  return M->X==x && M->Y==y  && strcmp(nom, M->Nom)==0;
}

int RechercheR(double x, double y, char *nom, struct point *p) {
  if (EstVide(p)) return 0;
  else if (Compare(x,y,nom,p)) return 1;
  else return RechercheR(x,y,nom,p->suivant);
}

int Recherche(double x, double y, char *nom) {
  struct point *M;
  for (M=premier; M!=NULL; M=M->suivant) {
    if (Compare(x,y,nom,M)) return 1; /* arrêt de la boucle */
  }
  return 0;
}
```

Destruction d'un élément dans la chaîne

Pour supprimer un élément, il faut :

→ S'assurer que l'opération est possible (la liste est non vide, l'élément à supprimer existe).

→ Éventuellement, libérer la mémoire occupée par l'élément à supprimer, si celle-ci a élé allouée dynamiquement par un appel à `new`.

→ Réaliser un nouveau chaînage.

remarque : la «récupération» de mémoire (obligatoire en `C++` et, à la charge du programmeur), est l'un des points délicats de la programmation. Le langage `Java` apporte une solution au problème en prenant à sa charge la gestion de la mémoire.

En `C++`, la fonction qui permet de restituer un bloc de mémoire est `delete`. Le bloc de mémoire doit avoir été alloué par `new`.

Afin de préciser les choses, étudions une fonction qui supprime le deuxième élément d'une chaîne.

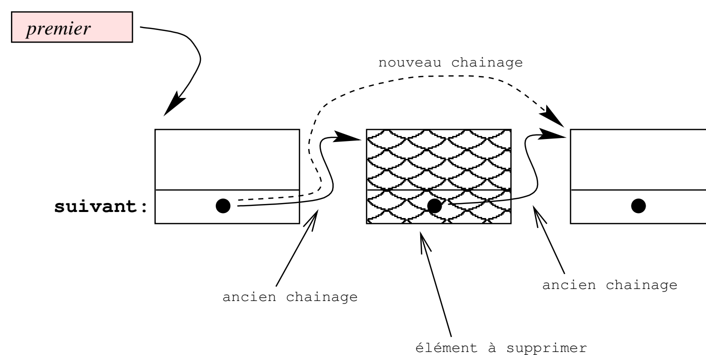


FIG. 12.8 – *Suppression du deuxième élément.*

```
int  SupprimerEnSecond() {
  struct point *M;
  /* suppression impossible ? */
  if (premier==NULL || premier->suivant==NULL) return 0;
  else { /* suppression possible */
    M = premier->suivant;
    premier->suivant = M->suivant; /* nouveau chainage */
    delete M; /* restitution de la mémoire */
    return 1;
  }
}
```

Chapitre 13

Mise en œuvre des interfaces

Une mise en œuvre complète des interfaces nécessite l'utilisation conjointe des classes abstraites et des patrons de fonctions (*template*).

13.1 Classe abstraite

13.1.1 Méthode virtuelle pure et classe abstraite

Une méthode est dite **virtuelle pure** si elle n'est pas définie dans la classe ou elle est déclarée. En `C++` on ajoute «`= 0`» à la déclaration de la méthode pour indiquer qu'elle est virtuelle pure.

Exemple :

```
virtual void int push(int) = 0;
```

Une **classe** est dite **abstraite** si elle contient au moins une méthode virtuelle pure.

Remarques :

- On ne peut pas créer d'instance d'une classe abstraite.
- Un classe abstraite ne peut pas être utilisée comme argument ou retour d'une fonction.
- Les pointeurs et les références sur une classe abstraites sont possibles.

Utilité : l'intérêt essentiel est de pouvoir matérialiser au niveau du code la notion d'interface.

Par exemple, on peut restreindre l'interface d'une structuration de donnée gérée en pile à trois fonctions :

`void push(int e)`	l'élément `e` est empilé
`int pop()`	un élément est retiré de la pile
`bool isEmpty()`	renvoie `true` si la pile est vide

Le codage en `C++` de cette interface donne :

```
#ifndef _pileEntier
#define _pileEntier
class pileEntier {
```

```
public:
    virtual void push(int e) = 0;
    virtual int pop() = 0;
    virtual bool isEmpty() = 0;
};
#endif
```

L'interface `pileEntier` est déclarée dans un fichier d'en-tête *pileEntier.hpp*. Dans le cas présent, il n'y a pas de fichier de code (définition des méthodes) à écrire, puisque toute les méthodes ont été déclarées virtuelles pures. Ce sont les classes qui vont hériter de `pileEntier` qui devront implémenter le code.

Au moment de la définition de l'interface, on ne se pose pas la question de la mise en œuvre. On ne sait pas si la pile sera gérée avec un tableau, une liste chaînée …

Mise en œuvre en tableau. On créée une classe `pileTableauEntier` qui hérite de `pileEntier`.

Codage :

1. Déclaration de la classe `pileTableauEntier` (*pileTableauEntier.hpp*)

```
#ifndef _pileTableauEntier
#define _pileTableauEntier

#include "pileEntier.hpp"
class pileTableauEntier : public pileEntier {
protected:
    int N;
    int* tab;
public:
    pileTableauEntier(int n);
    virtual ~pileTableauEntier();
    virtual void push(int e);
    virtual int pop();
    virtual bool isEmpty();
};

#endif
```

2. Définition des méthodes de la classe `pileTableauEntier` (*pileTableauEntier.cpp*)

```
#include "pileTableauEntier.hpp"

pileTableauEntier::pileTableauEntier(int n) { N=0; tab = new int[n]; }
pileTableauEntier::~pileTableauEntier() {delete[] tab;}
void  pileTableauEntier::push(int e) { tab[N++]=e; }
int  pileTableauEntier::pop() { return tab[--N]; }
bool  pileTableauEntier::isEmpty()  { return N==0; }
```

3. Fichier de test (*pileEntierT.cpp*)

```
#include <iostream>
#include <iomanip>

#include "pileTableauEntier.hpp"
```

```
int main() {

    /* Pile d'entier */
    pileTableauEntier P(12);
    P.push(1);  P.push(2);  P.push(3);  P.push(4);

    while (!P.isEmpty()) cout <<' '<<P.pop();

    cout<<endl;
    return 1;
}
```

13.2 Fonctions «patrons» (*template*)

13.2.1 Objectifs

- Lorsqu'un algorithme est le même pour plusieurs types de données, il est plus intéressant de créer une fonction générique pouvant s'adapter.
- C'est un modèle à partir duquel le compilateur peut générer les fonctions nécessaires à l'exécution.
- Exemple : un algorithme de tri peut être appliqué à tous les types de base, ainsi qu'à toutes les classes qui surchargent l'opérateur `<`.

Mise en œuvre sur un exemple

- Fonction «template» réalisant un tri a bulle sur un tableau d'objets ou de valeurs d'un type natif du langage.
- On utilise un symbole pour représenter la classe ou le type des objets à ranger :
  `template <class T>`
- À l'utilisation, tout se passe comme si on avait surchargé la fonction de tri autant de fois qu'il existe de types déclarés dans le programme.
- Contrainte pour l'exemple : l'opérateur `<` doit être surchargé.

Le code suivant définit une fonction patron (ou *paramétrée* ou *template*) :

```
template<class T>
void tri(T* tab, int n) {
    bool f;
    int N=n-1;
    do {
        f=false;
        for (int i=0; i<N; i++)
            if (tab[i+1]<tab[i]) {
               T m=tab[i]; tab[i]=tab[i+1];
               tab[i+1]=m; f=true;
            }
        N--;
    } while (f);
}
```

Le programme suivant utilise cette fonction pour le type natif `int` et pour le type `point` (créé par l'utilisateur).

```
#include <iostream.h>

struct point {
    double X,Y;
    point(){}
    point(double x, double y):X(x),Y(y) {}
    bool operator<(point& P) {
        return X*X+Y*Y < P.X*P.X+P.Y*P.Y;
    }
};

point P[4]={point(1,1), point(0,0.5),
            point(3,8), point(2,1.5)};

int I[5]={1,5,3,7,3};

int main() {
    tri(P,4);
    for (int i=0;i<4;i++)
       cout <<'('<<P[i].X<<','
            <<P[i].Y<<"); ";
    cout << endl;
    tri(I,5);
    for (int i=0;i<4;i++)
      cout <<I[i]<<"; ";
    cout << endl;

    return 1;
}
```

13.3 Classes «patrons»

13.3.1 Objectifs

- Elles permettent de créer des classes génériques
- Pour les mêmes raisons que les patrons de fonctions, il est intéressant d'écrire une seule fois un modèle pour plusieurs classes.
- Le paramétrage d'une classe abstraite donne des perspectives intéressante pour la mise en œvre de types abstraits.
- Elles matérialisent la notion théorique d'**interface**.

13.3.2 Exemple

1. Déclaration d'un type *pile* paramétré (➔ on pourra utiliser la pile pour tous les types d'objets,
2. Mise en œuvre d'une pile en tableau et en liste chaînée

Classe abstraite :

```
// fichier : pileAbstraite.hpp
// classe : pileAbstraite
```

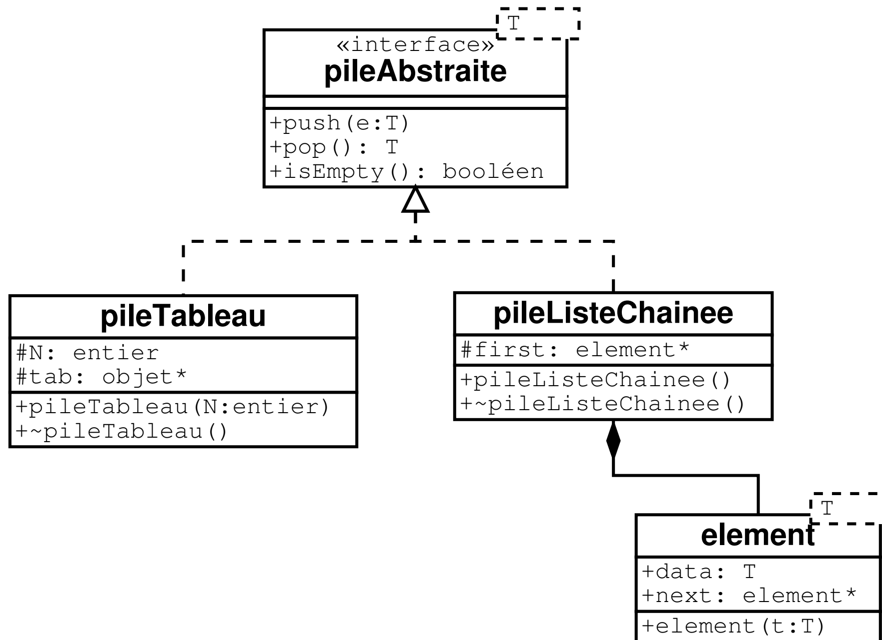


FIG. 13.1 – *Mise en œuvre d'un type abstrait* `pileAbstraite`

```
#ifndef _pileAbstraite
#define _pileAbstraite

template<class T> class pileAbstraite {
public:
    virtual void push(T e) = 0;
    virtual T pop() = 0;
    virtual bool isEmpty() = 0;
};
#endif
```

Mise en œuvre avec un tableau

```
#ifndef _pileTableau
#define _pileTableau
#include "pileAbstraite.hpp"
template<class T>  class pileTableau : public pileAbstraite<T> {
protected:
    int N;
    T* tab;
public:
    pileTableau(int n) { N=0;   tab = new T[n]; }
    virtual ~pileTableau() {delete[] tab;}
    void push(T e) { tab[N++]=e; }
    T pop() { return tab[--N]; }
    bool isEmpty()  { return N==0; }
};
#endif
```

Mise en œuvre avec une liste chaînée La classe `pileListeChainee` utilise ici une classe interne `element` pour réaliser le chaînage :

```
#ifndef _pileListeChainee
#define _pileListeChainee

template<class T> class pileListeChainee : public pileAbstraite<T> {
protected:
    class element {
    public:
        T data;
        element  *next;
        element(T t) { data = t; }
    };
    element *first;

public:
   pileListeChainee() {first=NULL;}
   virtual ~pileListeChainee() {
        while (first!=NULL) { delete first; first=first->next; }
   }
   void push(T e) {
        element* E = new element(e);
        E->next = first;
        first = E;
   }

   T pop() {
        T retour = first->data;
        element *X = first;
        first = first->next;
        delete X;
        return retour;
    }

    bool isEmpty() {return first==NULL;}

};
#endif
```

Programme de test dans lequel les variables instanciées à partir du type paramétré sont passées comme argument à une fonction f. On utilise le type abstrait pour l'argument.

```
#include <iostream.h>

#include "pileTableau.hpp"
#include "pileListeChainee.hpp"

struct date {
    int j, m, a;
};
```

```
template <class T>
void f(pileAbstraite<T>& p, char* m) {
    cout<<"Pile "<<m << (p.isEmpty()?" ":" non ") << "vide\n";
}
int main() {
    /* Piles d'entiers */
    pileTableau<int> P(12);
    pileListeChainee<int> PL;
    P.push(1);     P.push(2);     P.push(3);     P.push(4);
    PL.push(5);    PL.push(6);    PL.push(7);    PL.push(8);
    while (!P.isEmpty()) cout <<' '<<P.pop(); cout<<endl;
    while (!PL.isEmpty()) cout <<' '<<PL.pop(); cout<<endl;

    /* Piles de dates */
    pileTableau<date> D(14);
    pileListeChainee<date> DL;
    date d1={31, 12, 1987},d2={11,9,1989},d3={31,3,1991},d4={23,5,2000};
    D.push(d1); D.push(d2); D.push(d3); D.push(d4);
    while (!D.isEmpty()) DL.push(D.pop());
    while (!DL.isEmpty()) {
        date d = DL.pop();
        cout <<'['<<d.j<<'/'<<d.m<<'/'<<d.a<<"]\n";
    }
    f(P,"P"); f(D,"D"); f(DL,"DL");
    return 1;
}
```

Exécution

```
 % a.out
 4 3 2 1
 8 7 6 5
[31/12/1987]
[11/9/1989]
[31/3/1991]
[23/5/2000]
Pile P vide
Pile D vide
Pile DL vide
%
```

Chapitre 14

Les fichiers

14.1 Introduction aux fichiers

La notion de fichier est associée au stockage des informations sur la mémoire de masse.

Selon les informations contenues, on distingue deux types de fichiers :

les fichiers textes : ils ne contiennent que des caractères imprimables (en général les caractères affichables du code ASCII).

les fichiers binaires : ils peuvent contenir n'importe quelle valeur de caractères (de 0 à 255). Ce sont en général des fichiers exécutables ou des fichiers de données «lisibles» à travers un logiciel spécifique.

Selon la manière d'accéder aux données, on les divise en deux catégories (du point de vue de du programme qui y accède) :

les fichiers séquentiels : dans ce mode d'accès, on utilise des fonctions qui permettent des opérations sur les fichiers ou les caractères sont lus ou écrits dans l'ordre où ils apparaissent dans le programme. L'écran et le clavier sont des cas particuliers de fichiers séquentiels appelés respectivement `stdout` et `stdin`.

les fichiers à accès sélectifs : les fonctions prévues pour ce mode d'accès permettent de positionner un pointeur de lecture ou d'écriture dans un endroit quelconque du fichier.

Quelque soit le mode d'accès, pour utiliser un fichier dans un programme, il convient :

1. de l'ouvrir (en lecture ou en écriture) ;
2. d'effectuer les opération de lecture ou d'écriture ;
3. de fermer le fichier une fois que toutes les opérations sont terminées.

14.2 Mise en œuvre des fichiers séquentiels : ouverture et fermeture

Considérons le programme suivant :

```
#include <stdio.h>
```

```
void main() {
  FILE *fp;
  char c;
  fp = fopen("exemple.txt", "r");
  while (!feof(fp)) {
    c = fgetc(fp);
    if (isascii(c)) printf("%c", toupper(c));
  }
  printf("\n");
  fclose(fp);
}
```

le fichier `exemple.txt` contient les informations suivantes :

```
Ceci est un exemple de fichier !
Il comporte trois lignes et
plusieurs caractères !!!
```

Il doit se trouver dans le même répertoire que le fichier exécutable.

Commentaires :

† `fp` est un pointeur sur un descripteur de fichier (c'est à dire une variable de type `FILE*`. Le descripteur est nécessaire pour toutes les opérations de lecture et/ou d'écriture sur les fichiers.

† L'instruction `fp = fopen(...);` réalise la liaison entre le descripteur et un nom de fichier. De plus le mode d'accès au fichier est précisé. Il peut être[1] :
- `"w"` pour écrire dans le fichier (: on modifie le fichier,
- `"r"` pour lire le fichier (: on consulte simplement les informations du fichier).
- `r+` pour lire et écrire dans le fichier.
- `a` pour écrire des information à la fin d'un fichier déjà existant.

† `feof(fp)` renvoie une valeur booléenne vraie lorsque la fin de fichier est atteinte.

† `fgetc` permet de lire un fichier caractère par caractère. Cette fonction illustre très bien le principe de séquentialité. Á chaque nouvel appel, c'est un nouveau caractère qui est lu. Il n'y a pas de variable à incrémenter ou de pointeur à déplacer. Tout se passe comme si le fichier (ou son descripteur) fournissait les caractères dans l'ordre, jusqu'à ce qu'il n'y en ait plus.

† `fclose(...)` est utilisée pour «fermer» le fichier. Si le programme est accidentellement interrompu avant l'appel à `fclose`, le contenu du fichier sera irrémédiablement perdu. Fermer un fichier dés que l'on a fini de l'utiliser (même en cours de programme) est une manière de garantir la sécurité de l'information.

14.3 Opérations sur les fichiers séquentiels

Les fonctions à utiliser sont présentées à travers des exemples pour lesquels on conviendra des déclarations suivantes :

[1] Pour un information plus complète, consulter l'aide en ligne (`man fopen`)

```
#include <stdio.h>

char c;
int n;
char b[128];
FILE *fp;
```

14.3.1 Lecture de fichiers textes

Le fichier doit être préalablement ouvert dans un mode ou il est permis de le lire :

```
fp = fopen("exemple.txt", "r");
```

Il est alors possible de le lire **par caractères** :

```
c = fgetc(fp);
```

ou **par lignes** :

```
fgets(b, 127, fp);
```

l'entier donné comme deuxième paramètre permet de limiter le nombre de caractères à lire, dans le cas ou une ligne contiendrait plus de caractères que ce qu'il est possible de stocker dans le *buffer* prévu. Il est important que cet entier soit strictement inférieur à la taille du *buffer* à cause du caractère `'\0'` ajouté par `fgets` à la fin de la chaîne lue.

On peut également lire un fichier texte préalablement **formaté** :

```
fscanf(fp,"%c%d%s", &c, &n, b);
```

Le fonctionnement est similaire au `scanf`, si ce n'est que les informations sont lues dans le fichier de descripteur `fp` au lieu du clavier.

Remarque : la fonction `sscanf` travaille selon le même principe, en prenant pour source d'informations une chaîne de caractères : Considérons les déclarations suivantes,

```
int i;
float x;
char *str = "103 3.14";
```

l'instruction

```
sscanf(str,"%d%f",&i, &x);
```

a pour effet d'intialiser $i \leftarrow 103$ et $x \leftarrow 3.14$.

14.3.2 Ècriture de fichiers textes

Le fichier doit être préalablement ouvert dans un mode ou il est permis de l'écrire ou de le modifier (`w`, `a`, `w+` ...) :

```
fp = fopen("exemple.txt", "w");
```

Par analogie aux opérations de lecture, on trouve :

l'écriture par caractères :

```
fputc(c, fp);
fputc('A', fp);
```

l'écriture par lignes :

```
fputs(b, fp);
fputs("Au revoir !", fp);
```

l'écriture formatée :

```
fprintf(fp,"%d => %c -- %s\n", n, c, b);
```

14.3.3 Lecture et écriture de fichiers binaires

Certains compilateurs répondant à la norme ANSI C3.159-1989 exigent qu'un fichier binaire dans un mode contenant le caractère `b`. Par exemple : `"rb"`, `"wb"`, `wb+`, ...

```
fp = fopen("exemple.bin", "wb");
```

Pour écrire n octets contenus dans le *buffer* `b[]`, ($n \leq 128$) on écrira :

```
fwrite(b, 1, n, fp);
```

Le deuxième argument (mis à 1 dans l'exemple) donne la taille des données à écrire. Ainsi, les appels `fwrite(b, 1, 128, fp)` ; et `fwrite(b, 4, 32, fp)` ; donneront le même résultat final.

Pour lire n octets contenus dans le fichier décrit par `fp` en les recopiant dans *buffer* `b[]` :

```
fread(b, 1, n, fp);
```

14.4 Les fichiers à accès sélectif

Les opérations d'ouverture, de fermeture, de lecture et d'écritures sont identiques. Ce qui change, c'est la possibilité d'intervenir sur la position en cours dans le fichier. La fonction `fseek` offre un moyen de déplacer le pointeur de caractère dans le fichier :

```
fseek(fp, offset, mode);
```

permet de définir la position courante à partir
- du début si `mode=0` (ou `SEEK_SET`)
- de la position courante si `mode=1` (ou `SEEK_CUR`)
- de la fin si `mode=2` (ou `SEEK_END`)

À partir de cette origine définie, le pointeur est positionné par déplacement de la valeur `offset`.

Exemple :

```
#include <stdio.h>

int main() {
  FILE *fp = fopen("exemple.txt", "r+");
  char c;
  while (!feof(fp)) {
    c = fgetc(fp);
    if (c=='e') {
      fseek(fp, -1L, SEEK_CUR);
      fputc('E', fp);
    }
  }
  fclose(fp);
  return 1;
}
```

Remarques

- Ce programme transforme tous les caractères `'e'` du fichier `exemple.txt` en `'E'`.
- L'instruction

  ```
  c = fgetc(fp);
  ```

  a pour effet de lire un caractère dans le fichier, mais le pointeur de caractères est avancé.
- Dans le cas ou le caractère devait être réécrit, l'instruction

  ```
  fseek(fp, -1L, SEEK_CUR);
  ```

  permet de reculer le pointeur d'un caractère de façon a pouvoir écrire un `'E'` au bon endroit.

14.5 Les flux

Nous les avons utilisés intuitivement lors du chapitre 4. Les classes de bases sont données par le diagrammme de la figure 14.1.

Ces classes sont dans *iostream.h*.

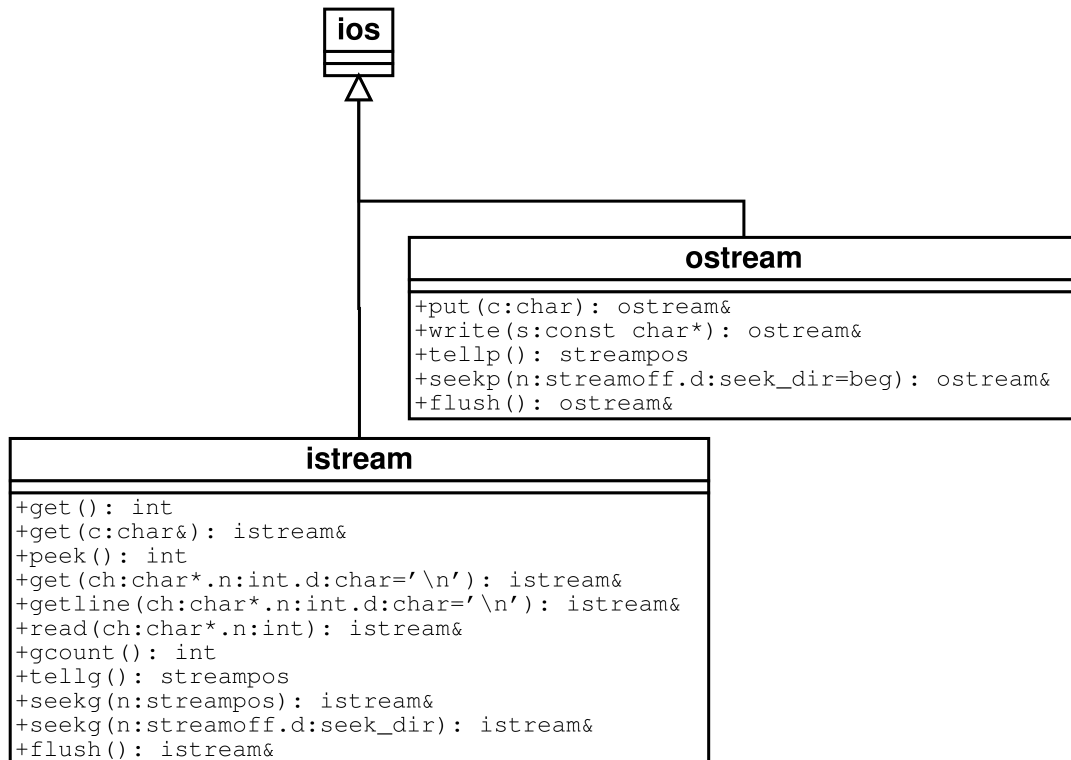


FIG. 14.1 – *Flux : classes de base.*

14.5.1 Le flux `ostream`

`ostream& putc(char c)`	envoie le caractère c dans le flux. `cout.put('U') ;`
`ostream& write(char* S,int n)`	envoie n caractères de la chaîne S dans le flux. `cout.write("Hello world", 5);`
`streampos tellp()`	retourne la position courante dans le flux.
`ostream& seekp(streampos n)`	Positionne le pointeur à n octets par rapport au début du flux. La première position vaut 0. Le type `streampos` est homogène à la position d'un caractère dans un fichier, c'est un type entier.
`ostream& seekp(streamoff n, seek_dir d);`	Positionne le pointeur à n octets par rapport : { au début si `d = seek_dir : :deb` à la position courante si `d = seek_dir : :cur` à la fin `d = seek_dir : :end` ($n < 0$) `seek_dir` est une énumération de la classe `ios`.
`ostream& flush()`	Vide les tampons du flux.

Exemple, calcul de la taille d'un fichier pendant sa lecture (on suppose que `fo` est un flux déjà ouvert) :

```
streampos P = fo.tellp(); // mémorise la position courante
fo.seekp(0, end); // pointeur à la fin du flux
```

```
cout << "Taille = " << fo.tellp() << "octets\n";
fo.seekp(P, deb); // retrouve la position d'origine
```

14.5.2 Le flux `istream`

Nous avons déjà utilisé l'opérateur d'extraction `>>`. Les principales autres méthodes sont :

Lecture d'un caractère

`int get() ;`	retourne la valeur du caractère lu (`EOF` si la fin de fichier est atteinte)
`istream& get(char&c) ;`	extrait le premier caractère du flux (même si c'est un espace) et le place dans *c*.
`int peek() ;`	lecture sans consommation du caractère suivant dans le flux (le caractère <u>reste</u> dans le flux. Retourne la valeur du caractère lu (`EOF` si la fin de fichier est atteinte)

Lecture d'une chaîne de caractères

`istream& get(char* S, int n, char d='\n');`	extrait $n-1$ caractères du flux et les place à l'adresse S. La lecture s'arrête au délimiteur qui est par défaut `'\n'` ou la fin de fichier. Le délimiteur n'est pas extrait du flux.
`istream& getline(char* S, int n, char d='\n');`	Même chose que la méthode précédente, sauf que le délimiteur est extrait du flux.

Autres méthodes

`istream& read(char&S, int n) ;`	extrait un bloc d'au plus n octets et les place à l'adresse S.
`int gcount() ;`	retourne le nombre de caractères extraits lors de la dernière lecture.
`streampos tellg()`	retourne la position courante dans le flux.
`istream& seekg(streampos n)`	Positionne le pointeur à n octets par rapport au début du flux. La première position vaut 0. Le type `streampos` est homogène à la position d'un caractère dans un fichier, c'est un type entier.
`istream& seekg(streamoff n, seek_dir d);`	Positionne le pointeur à n octets par rapport : { au début si `d = seek_dir : :deb` à la position courante si `d = seek_dir : :cur` à la fin `d = seek_dir : :end` ($n < 0$) `seek_dir` est une énumération de la classe `ios`.
`istream& flush()`	Vide les tampons du flux.

14.6 Application des flux aux fichiers

Alors que toutes les fonctions vues précédemment proviennent du langage C, le C++ autorise l'utilisation des flux pour l'accès aux fichiers. Les classes principales sont données par le diagramme de la figure 14.2. Ce diagramme est à compléter avec celui de la figure 14.1.

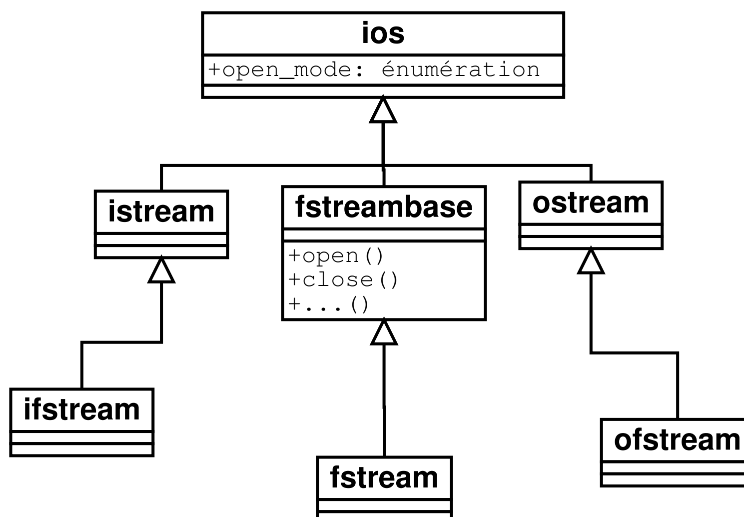


Fig. 14.2 – *Flux et fichiers.*

14.6.1 Association d'un fichier et d'un flux

Elle se fait grâce à la méthode open() :

```
void open(const char* nom, ios::open_mode mode, filebuf::prot droits);
```

ou,

- nom est le nom du fichier à ouvrir.
- mode précise le mode d'ouverture du fichier.
- droits (option pour UNIX) droits d'accès.

Le paramètre mode est à choisir dans l'énumération définie dans la classe ios :

```
enum open_mode {
  app,       // ajout des données en fin de fichier
  ate,       // positionnement à la fin du fichier
  in,        // ouverture en lecture
  out,       // ouverture en écriture
  binary,    // ouverture en mode binaire
  trunc,     // détruit le fichier s'il existe et le remplace
  nocreate,  // pas d'ouverture si le fichier n'existe pas déjà
  noreplace  // pas d'ouverture si le fichier existe
};
```

Exemples :

```
#include <fstream>

ifstream fi;
fi.open("test1.txt"); // ouverture en lecture de 'test1.txt'

ofstream fo;
fo.open("test2.txt"); // ouverture en écriture de 'test2.txt'
fo >> 'U';            // écriture d'un caractère dans 'test2.txt'
fo.close();           // fermeture de 'test2.txt'

fstream fp;
fp.open("test3.txt", ios::in | ios::out | ios::binary);
// 'test3.txt' est ouvert en lecture/écriture et en mode binaire
```

14.6.2 Contrôle de l'état d'un flux

Quelques méthodes de la classe virtuelle `ios` sont prévues à cet effet.

`int good();`	retourne une valeur $\neq 0$ si la dernière opération d'entrée/sortie s'est effectuée avec succès, et 0 en cas d'échec.
`int fail();`	le contraire de la précédente
`int eof();`	retourne une valeur $\neq 0$ si la fin de fichier est atteinte et 0 sinon
`int bad();`	retourne une valeur $\neq 0$ si une opération interdite a été tentée et 0 sinon
`int rdstate();`	retourne la valeur de la variable d'état du flux (0 si tout va bien)
`void clear();`	remet à 0 l'indicateur d'erreur du flux. À faire systématiquement après qu'une erreur se soit produite

14.6.3 Formatage des données

Chaque flux conserve en permanence un ensemble d'indicateurs spécifiant l'état du formatage des données. Ceci permet d'initialiser le flux avec un certain comportement pour toute une application[2].

L'indicateur de format du flux est contenu dans la classe `ios`. Les indindicateurs sont accessibles séparément grâce à une énumération de `ios` :

```
enum {
  skipws,     // ignore les espaces en entrée
  left,       // justifie les sorties à gauche
  right,      // justifie les sorties à droite
  internal,   // remplissage après le signe ou la base
  dec,        // conversion en décimal
  oct,        //            en octal
  hex,        //            en hexadécimal
  showbase,   // affiche l'indicateur de la base
  showpoint,  // affiche le point décimal pour les réels
  uppercase,  // nombre hexadécimaux en majuscules
  showpos,    // affiche le signe '+' devant les nombres positifs
  scientific, // notation 1.234000E2 (réels)
```

[2] Avec `scanf` et `printf` il faut préciser le formatage à chaque fois.

```
    fixed,      // notation 123.4 (réels)
    unitbuf,    // vide les flux après une insertion
    stdio       // permet d'utiliser cout et stdout
  }
```

Ces indicateurs sont accessibles par groupes, identifiés par des constantes de la classe `ios` :

- `basefield` : pour le choix de la base,
- `adjustfield` : pour l'alignement,
- `floatfiels` : pour le choix de la notation des réels.

Les méthodes suivantes permettent d'agir sur les indicateurs :

`long flags()`	retourne la valeur de l'indicateur de formats
`long flags(long n)`	modifie l'ensemble des indicateurs (mis à n) et retourne l'ancienne valeur
`long setf(long n, long f)`	permet de sélectionner un indicateur n d'un groupe f `cout << setf(ios::hex, ios::basefield);`
`long setf(long i)`	positionne un ou plusieurs indicateurs de format donné et retourne l'ancienne valeur de ces indicateurs. `cout.setf(ios::oct | ios::right)`
`long unsetf(long i)`	efface les indicateurs précisés.

On peut aussi agir sur le format des informations avec les méthodes :

`int width(int n)`	fixe la largeur du champ de sortie
`int width()`	retourne la largeur du champ de sortie
`cgar fill char c)`	fixe le caractère de remplissage
`int precision(int n)`	fixe le nombre de chiffres significatifs pour les réels
`int precision()`	retourne ce même nombre.

Exemple

```
cout << "Total";
cout.width(12);
cout.fill('.');
cout.precision(4);
cout<<  12.34E1 << ";" << endl;
```

affiche :

```
Total.......123.4;
```

14.6.4 Les manipulateurs

Les manipulateurs servent aussi à formater les données, mais rendent le programme plus lisible. Ainsi, l'exemple précédent peut s'écrire :

```
cout << "Total" << setw(12) << setfill('.')
     << setprecision(4) <<  12.34E1 << ";" << endl;
```

Le manipulateurs usuels sont présentés dans le tableau 4.2 (page 32).

Création d'un manipulateur sans paramètre

Supposons que l'on souhaite écrire un manipulateur qui regroupe la présentation de l'exemple précédent. Il suffit d'écrire :

```
ostream& mon_manip(ostream& os){
  return os <<  "Total" << setw(12)
            << setfill('.')  << setprecision(4);
}
```

(`mon_manip` est le nom choisi pour appeler le nouveau manipulateur). Le code se réduit alors à

```
cout << mon_manip << 12.34E1 << ";" << endl;
```

Dans le codage du manipulateur, le fait de recevoir un flux en argument et d'en retourner un autre est le mécanisme qui permet d'utiliser le manipulateur en cascade.

Création d'un manipulateur avec paramètre

C'est un peu plus compliqué, l'implémentation se fait en 2 parties :

1. le **manipulateur** dont la forme générale est :

   ```
   ostream& nom_du_manip(ostream& os, <type> n)
   ```

   `<type>` étant le type du paramètre.
2. l'**applicateur**, qui appelle le manipulateur. C'est une fonction globale dont la forme est :

   ```
   xxxmanip(<type>) nom_du_manip(type n ) {
       return xxxmanip(<type>)(nom_du_manip, n);
   }
   ```

   avec `xxx` =
   - `O` pour un `ostream`
   - `I` pour un `istream`
   - `IO` pour un `iostream`

Exemple : Le programme suivant construit 2 manipulateurs pour afficher des nombres en binaire : un pour les entiers longs, et l'autre pour les caractères.

```
#include <iostream>
#include <iomanip>
#include <limits.h> // pour ULONG_MAX

// ------ manipulateur base 2 ---------------------------
ostream& bin(ostream& os, long n) {
  for (unsigned long mask = ~(ULONG_MAX  >> 1); mask; mask>>=1)
    os << ((n & mask)?'1':'0');
  return os;
}
omanip<long> bin(long n){ return omanip<long>(bin,n); }

ostream& bin(ostream& os, char n) {
  for (unsigned char mask = 0x80; mask; mask>>=1)
    os << ((n & mask)?'1':'0');
  return os;
}
```

```
omanip<char> bin(char n){ return omanip<char>(bin,n); }
// -----------------------------------------------------

int main() {
  cout << "2003(2) --> " << bin(2003L) << endl;
  cout << "0x55(2) --> " << bin((char)0x55) << endl;
  return 1;
}
```

L'exécution donne :

```
2003(2) --> 00000000000000000000011111010011
0x55(2) --> 01010101
```

Notez que dans le programme principal, les données à afficher doivent être bien typées.

Chapitre 15

Les arguments de la ligne de commande

Lorsqu'un programme est lancé à partir d'un interpréteur de commande, il est dans certain cas commode de lui passer des arguments. Par exemple, dans la commande

```
--> copy f1.txt f2.txt
```

le nom du programme est `copy` et les arguments sont `f1.txt` et `f2.txt`. Lorsque la commande `copy` a été écrite (dans un fichier *copy.c*, un moyen technique a du être mis en œuvre pour «récupérer» les arguments donnés sur la ligne de commande, donc au moment de l'exécution.

En `C` , les arguments de la ligne de commandes sont associés aux paramètres optionnels de la fonction principale `main`.

La fonction `main` peut recevoir des paramètres depuis le système d'exploitation. Le compilateur fait le lien avec le système d'exploitation par deux paramètres en général appelés `argc` et `argv`, respectivement de type entier et tableau de chaînes de caractères.

Le prototype de la fonction `main` est donc :

```
int main(int argc, char *argv[])
```

ou

- `argc` est un entier qui est égal au nombre d'arguments passés sur la ligne de commande. Le nom de la commande est comptabilisé dans `argc`.
- `argv` est un tableau de chaînes de caractères correspondant aux différent arguments passés. `argv[0]` est le nom de la commande. `argv[argc-1]` est le dernier argument.

Remarque : la fonction `main` renvoie un entier en `C` normalisé. Il peut être utilisé pour rendre compte du fonctionnement normal ou non du programme.

Mise en œuvre

Considérons le programme suivant :

```
/* fichier : argument.c */
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
```

```
  printf("argc = %d\n", argc);
  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
  return 1;
}
```

La compilation par la ligne

```
gcc -o argument argument.c
```

donne le programme exécutable `argument`. On teste ensuite ce programme sous l'interpréteur de commande du système d'exploitation :

```
 > argument
argc = 1
argv[0] = argument
 > argument un
argc = 2
argv[0] = argument
argv[1] = un
 > argument un deux
argc = 3
argv[0] = argument
argv[1] = un
argv[2] = deux
 > argument un deux "et trois"
argc = 4
argv[0] = argument
argv[1] = un
argv[2] = deux
argv[3] = et trois
 >
```

Lorsque plusieurs mots sont encapsulés par des double-quotes ("), ils sont considérés comme un seul argument.

15.1 Application

Écrire un programme `calcul` qui reçoit au moins 3 paramètres sur la ligne de commande : un opérateur et au moins deux opérandes.

1. si le nombre de paramètre est insuffisant, un message *intelligent* est affiché ;
2. si l'opérateur est "-a", le programme affiche la somme des opérandes ;
3. si l'opérateur est "-s", le programme affiche la différence entre le premier opérande et tous les suivants ;
4. si l'opérateur est "-m", le programme affiche le produit de tous les opérandes ;
5. une mauvaise syntaxe pour l'opérateur conduit à l'affichage d'un message d'erreur pertinent.

Exemples d'utilisation du programme

```
> calcul -m 1 2 3 4
24.000000
> calcul -s 1 2 3 4
-8.000000
```

```
 > calcul -a 1 2 3 4
 10.000000
 > calcul
calcul -[asm] op1 op2 [op3 ...]
 > calcul -r 1 2
L'opérateur doit être dans [asm]
 >
```

Indication : pour la transformation $char* \rightarrow double$, voir la fonction `atof`.

Chapitre 16

Les exceptions

16.1 Généralités

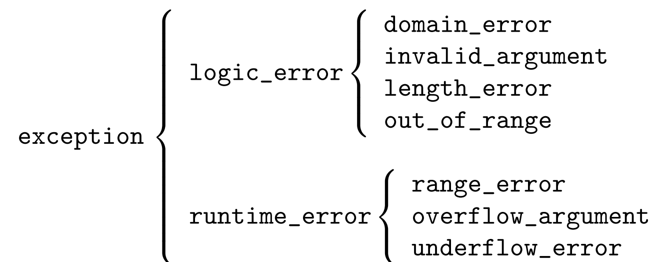
Lors de l'exécution d'un programme, des erreurs peuvent survenir :
- erreurs **logiques** (divisions par zéro, ...)
- erreurs **matérielles** (allocation mémoire, faute de segmentation)

Le `C++` permet au programme de traiter ces situations délicates grâce à la structure de contrôle : `try ...catch`. Son utilisation est la suivante :

```
try {
  // instructions à risque
} catch (exception e) {
  // code à exécuter en cas d'erreur
}
```

Le paramètre du `catch` est un objet de classe `exception` ou bien dérivé de exception.

La hiérarchie des classes d'exceptions est la suivante :



Elles sont définies dans le fichier *stdexcept*.

Exemple dans lequel on cherche à déclarer un tableau le plus grand possible.

```
#include <stdexcept>
#include <iostream>

int main() {
  char *p;
  bool f = true;
  unsigned long N = 1000000000L;
```

```
    do {
      try {
        p = new char[N];
        f = false;
      } catch (exception e) {
        N -= 1000;
      }
    } while (f);
    cout << "Valeur finale, N = " << N << endl;
    delete[] p;
    return 1;
  }
```

Exécution :

```
Valeur finale, N = 206319000
```

Remarques :

1. Lorsqu'une exception est détectée dans un bloc `try`, le contrôle de l'exécution est donné au bloc `catch` correspondant.
2. Un bloc `try` doit être suivi d'un bloc `catch` au moins.
3. Quand une exception est détectée, les destructeurs des objets inclus dans le bloc `try` sont appelés avant d'appeler un bloc `catch`.
4. À la fin d'un bloc `catch` le programme continue son exécution sur l'instruction qui suit le dernier bloc `catch`.
5. Capturer une exception de type `exception` permet de capturer tout ce qu'il est possible de capturer.

16.2 Création d'exceptions

Un objet donné peut posséder sa ou ses propre(s) exceptions. Lors de l'appel des méthodes de cet objet, les exceptions peuvent être provoquées. Ceci se fait avec l'instruction `throw`. On dit qu'une exception est levée.

Exemple :

```
#include <stdexcept>
#include <iostream>

class truc {
public:
  /* ----- définition de l'exception */
  class ex_truc : public exception {
    char* message;
  public:
    ex_truc() { message = "exception de 'truc'";}
    virtual const char* what () const { return message; }
  };
  /* -------------------------------- */

  truc() {}
  void fonc(int n) {
```

```
    if (n==13) throw ex_truc();
    else cout << " n = " << n << endl;
  }
};

int main() {
  truc T;
  try { T.fonc(4); } catch (truc::ex_truc e) { cout << e.what() << endl;}
  try { T.fonc(13); } catch (truc::ex_truc e) { cout << e.what() << endl;}

  return 1;
}
```

La classe qui se protège par un système d'exception s'appelle ici `truc`. L'exception utilisée est une classe dérivée de `exception` : c'est la classe `ex_truc` interne à `truc`.

Exécution

```
 n = 4
exception de 'truc'
```

En plus de l'aspect «sécurité», L'utilisation des exceptions est un atout pour la structuration du logiciel.

Chapitre 17

La Standard Template Library

17.1 Généralités

La S.T.L. propose une solution «clef en main» pour les problèmes classiques de la programmation : structure de données et algorithmes. Elle est développée selon les axes :

$$\begin{cases} \text{conteneurs} \\ \text{algorithmes} \\ \text{itérateurs} \end{cases}$$

Elle permet de mettre en œuvre d'autres structures de données plus complexes (graphes, automates ...) et de faciliter l'écriture d'algorithmes avancés. Par exemple, fusionner deux listes et placer le résultat dans un fichier se fait en une seule ligne.

Étant donné la complexité du langage C++ pour la mise en œuvre de structure de données abstraites (en comparaison de java par exemple), l'usage systématique des structures de données de la S.T.L. est fortement recommandé.

17.2 Les conteneurs

Les conteneurs regroupent les structures de données linéaires usuelles : vecteurs, listes, dictionnaires .... L'utilisation des templates dans la S.T.L. permet l'adaptation à tous les types de données, type natifs du langage ou classes.

Chaque conteneur étant caractérisé par

$$\begin{cases} \text{un coût des insertions/suppressions d'éléments} \\ \text{un coût des accès aux éléments.} \end{cases}$$

, le choix de l'un ou l'autre devra être guidé par l'application.

17.2.1 Les vecteurs (vector)

Méthodes importantes

- size(), empty(), clear(), resize()
- front(), back()
- push_front(val), push_back(val)
- pop_front(), pop_back()

Opérateurs : =, ==, <, []

Exemple

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

using namespace std;

int main(){
    vector<int> v1;

    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);

    vector<int> v2(3);
    v2[0] = 1;
    v2[1] = 2;
    v2[2] = 3;

    cout << "Vecteurs " << (v1==v2 ? "identiques" : "différents") << endl;

    return 1;
}
```

Avantages :
- accès aléatoire aux éléments ($O(1)$) ;
- ajout/suppression d'éléments à la fin ($O(1)$) ;
- gestion automatique de la mémoire.

Inconvénient : ajout/suppression au début ou au milieu en $O(n)$.

Spécialisations : `sort()`, `unique()`, `reverse()` ;

17.2.2 Les listes (`list`)

Méthodes importantes
- `size()`, `empty()`, `clear()`, `resize()`
- `front()`, `back()`
- `push_front(val)`, `push_back(val)`
- `pop_front()`, `pop_back()`, `remove(val)`

Opérateurs : `=`, `==`, `<`, `[]`

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

using namespace std;

int main(){
    list<int> l;
```

```
    l.push_back(2);
    l.push_back(1);
    l.push_back(3);

    l.sort();
    l.reverse();

    cout << "front=" << l.front() << "; back=" << l.back() << endl;

    return 1;
}
```

Avantage : il s'agit en fait d'une liste doublement chaînée. Les ajouts/suppressions sont donc en $O(1)$.

Inconvénient : accès aux éléments en $O(n)$.

17.2.3 Les *double end queues* (deque)

Une *deque* se comporte comme un vecteur, à la différence que l'ajout et la suppression d'un élément au début se fait en temps constant ($O(1)$).

Cette structure est très avantageuse dans les cas ou les ajouts/suppressions se font aux extrémités et ou on a besoin d'un accès efficace aux éléments.

Méthodes importantes : voir `vector`.

```
#include <deque>
#include <iostream>

using namespace std;

int main(){
    deque<int> d1;

    d1.push_back(1);
    d1.push_back(2);
    d1.push_back(3);

    deque<int> d2(3);
    d2[0]=1; d2[1]=2; d2[2]=3;

    cout << "Deque " << (d1==d2 ? "identiques" : "différents") << endl;

    return 1;
}
```

17.2.4 Les ensembles (set)

Chaque valeur d'un ensemble est unique.

Méthodes importantes

- `insert(val)`, `erase(val)`,
- `find(val)` (renvoie un itérateur),
- `size()`, `empty()`.

Opérateurs : `=`, `==`, `<`.

```
#include <set>
#include <iostream>

using namespace std;

int main(){
    set<int> S;

    S.insert(1); S.insert(2); S.insert(3); S.insert(4);
    S.insert(1); S.insert(2); S.insert(3); S.insert(4);

    S.erase(2);

    cout << "size = " << S.size() << endl;

    return 1;
}
```

Remarques :

- les opérations sont toutes en $O(n)$ (ajout, suppression, accès) ;
- en interne les éléments sont toujours rangés par ordre croissant.

17.2.5 Les dictionnaires (`map`)

C'est une implémentation de table : ensemble de doublets $(clef, valeur)$ avec unicité de la clef.

Méthodes importantes

- `insert(val)`, `erase(val)`
- `find(val)` (renvoie un itérateur)
- `size()`, `empty()`, `clear()`.

Opérateurs : `=`, `==`, `<`, `[]`.

L'opérateur `[]` sert pour accéder à un élément par sa clef, et aussi pour en ajouter un nouveau.

```
#include <map>
#include <iostream>

using namespace std;
```

```
int main(){
    map<string, string> tel;

    tel["zaza"] = "029-456-4587";
    tel["max"] = "+33603202377";

    cout << "tel(max) = " << tel["max"] << endl;

    return 1;
}
```

17.2.6 `multiset` et `multimap`

Ce sont des variantes de `set` et `map`, définis dans les mêmes en-têtes :
- `multiset` autorise des valeurs répétées
- `multimap` autorise des clef multiples

```
multimap M;
M[make_pair("clavier", "azerty")] = "ok";
```

17.2.7 Autres conteneurs

`slist` : liste simplement chaînée ;

`stack` : structure de pile, on y trouve en plus les méthodes `push(val)`, `pop()` et `top()`.

`queue` : dérivée de `deque`, avec en plus les méthodes `push(val)` et `pop()`.

17.3 Les itérateurs

Il s'agit d'une généralisation de la notion de pointeurs. Ce sont des objets qui pointent d'autres objets. Ils servent de lien entre les conteneurs et les algorithmes, car ils permettent d'accéder aux informations, toujours de la même manière.

Chaque conteneur fourni un type d'itérateur. Par exemple, le type `list<int>` donne un itérateur de type `list<int> : :iterator`. Deux valeurs particulières sont renvoyées par les méthodes `begin()` et `end()` de ces itérateurs.

- `begin()` : itérateur sur le premier élément
- `end()` : itérateur sur le premier élément en dehors du conteneur.

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

using namespace std;

int main(){
    list<int> V;

    for (int i=1; i<=20; i++) V.push_back(i*i);
```

```
    list<int>::iterator I;
    for (I=V.begin(); I!=V.end(); I++) cout << *I << " ";
    cout << endl;

    list<int>::reverse_iterator RI;
    for (RI=V.rbegin(); RI!=V.rend(); RI++) cout << *RI << " ";
    cout << endl;

    return 1;
}
```

Dans cet exemple, on utilise également un itérateur inversé qui permet le parcours en sens inverse.

Opérateurs : les principaux sont `*` qui donne l'acccès à la valeur, et les opérateurs classiques `++` et `--` d'incrémentation et de décrémentation.

17.4 Les algorithmes

D'une manière générale, ils utilisent les itérateurs. On les sépare en deux catégories :

1. ceux qui ne modifient pas le conteneur (*non mutating algorithms*)
2. ceux qui le modifient (*mutating algorithms*).

17.4.1 *Non mutating algorithms*

recherche `binary_search()`

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

using namespace std;

int main(){
    list<int> V;

    for (int i=1; i<=20; i++) V.push_back(i*i);


    list<int>::iterator I;
    for (I=V.begin(); I!=V.end(); I++) cout << *I << " ";
    cout << endl;

    bool res = binary_search(V.begin(), V.end(), 13);
    cout << "13 -> " << (res?"Trouvé":"Pas trouvé") << endl;
    res = binary_search(V.begin(), V.end(), 121);
    cout << "121 -> " << (res?"Trouvé":"Pas trouvé") << endl;
    return 1;
}
```

comptage `count` compte le nombre d'"éléments égaux à une valeur donnée.

```
count(V.begin(), V.end(), 10);
```

comparaison de deux conteneurs `equal()` : compare élément à élément le contenu de deux conteneurs. Les conteneurs peuvent être de types différents.

recherche `find()` : renvoie un itérateur sur la première occurence d'un élément recherché.

```
list:<int>::iterator K = L.find(121);
```

17.4.2 *Sorting algorithms*

minimum et maximum `min` et `max` renvoient le minimum et le maximum compris entre deux valeurs. `min_element` et `max_element` renvoient le plus petit et le plus grand d'un conteneur.

tri `sort` effectue un tri ; `is_sorted()` renvoie vrai si le conteneur est ordonné entre les deux itérateurs donnés en argument. Il faut que l'opérateur< soit implémenté pour le type concerné.

inclusion `includes`

```
set<int> A, B;
...
if ( includes(A.begin(), A.end(), B.begin(), B.end()) {...}
```

Attention, les conteneurs doivent être préalablement ordonnés (ce qui est automatiquement le cas dans l'exemple précédent.

17.4.3 *Mutating algorithms*

copie `copy` effectue une copie d'un conteneur vers un autre.

```
list<int> V1;
list<int> V2;
...
V2.resize(V1.size());
copy(V1.begin(), V1.end(), V2.begin());
```

`fusion` `merge()` fusionne deux conteneurs de valeurs triées en un seul ensemble trié de valeurs.

```
int tab1[] = {1, 3, 5, 7};
int tab2[] = {2, 4, 6, 8};

list<int> L;
L.resize(8);

merge(tab1, tab1+4, tab2, tab2+4, L.begin());
```

remplacement `replace()` remplace les occurences d'une valeur par une autre.

```
replace( L.begin(), L.end(), "cerise", "pomme");
```

calcul de sommes `accumulate`

opérations ensemblistes `set_union`, `set_intersection`, `set_difference` ;

inversion `reverse()`

remplissage `fill()`

suppression des doublons `unique()`

Annexe A

Codes ASCII

Déc	Hex	Clav	Car
0	00	^@	NUL
1	01	^A	SOH
2	02	^B	STX
3	03	^C	ETX
4	04	^D	EOT
5	05	^E	ENQ
6	06	^F	ACK
7	07	^G	BEL
8	08	^H	BS
9	09	^I	HT
10	0a	^J	LF
11	0b	^K	VT
12	0c	^L	FF
13	0d	^M	CR
14	0e	^N	SO
15	0f	^O	SI
16	10	^P	DLE
17	11	^Q	DC1
18	12	^R	DC2
19	13	^S	DC3
20	14	^T	DC4
21	15	^U	NAK
22	16	^V	SYN
23	17	^W	ETB
24	18	^X	CAN
25	19	^Y	EM
26	1a	^Z	SUB
27	1b	^[	ESC
28	1c	^\	FS
29	1d	^]	GS
30	1e	^^	RS
31	1f	^_	US
32	20		SP
33	21		!
34	22		"
35	23		#
36	24		$
37	25		%
38	26		&
39	27		'
40	28		(
41	29		)
42	2a		*

Déc	Hex	Car
43	2b	+
44	2c	'
45	2d	-
46	2e	.
47	2f	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3a	:
59	3b	;
60	3c	<
61	3d	=
62	3e	>
63	3f	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4a	J
75	4b	K
76	4c	L
77	4d	M
78	4e	N
79	4f	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U

Déc	Hex	Car
86	56	V
87	57	W
88	58	X
89	59	Y
90	5a	Z
91	5b	[
92	5c	\
93	5d	]
94	5e	^
95	5f	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6a	j
107	6b	k
108	6c	l
109	6d	m
110	6e	n
111	6f	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7a	z
123	7b	{
124	7c	:
125	7d	}
126	7e	~
127	7f	DEL

Index

analyse lexicale, 23
arguments `argc`, `argv`, 123
assembleur, 9
attribut, 69
`auto`, 16

bloc d'instructions, 34
`bool`, 15

chaînage, 95
champs de *bits*, 67
`char`, 11
choix, 39
  multiple, 40
  simple, 39
  simple répété, 39
`cin`, 7, 31
classe, 69
  dérivée, 85
  de base, 85
classe abstraite, 104
classe d'allocation, 16
classe patron, 107
commentaires, 8
compilateur, 9
compilation, 9
`const`, 16
constantes, 13
  booléennes, 15
  caractères, 14
  chaînes de caractères, 14
  entières, 13
  réelles, 13
conversions, 21
`cout`, 7, 31

`dec`, 32
`default`, 40
destructeur, 78
directives, 8
`do ...while`, 36
`double`, 13

éditeur de liens, 9

`endl`, 32
`ends`, 32
entrée standard, 25
`enum`, 64
énumérations, 64
exceptions, 126
expressions booléennes, 22
`extern`, 16

`false`, 15
`fclose`, 112
`feof`, 112
`fgetc`, 112, 113
`fgets`, 113
fichiers, 111
  accès sélectif, 111, 114
  binaires, 111, 114
  séquentiels, 111
  textes, 111
`FILE`, 112
`float`, 12
flux
  état, 119
  formatage, 119
  redirection, 77
fonction amie, 77
fonction virtuelle, 91
`fopen`, 112
`for (... ; ... ; ...)`, 37
`fprintf`, 114
`fputc`, 114
`fputs`, 114
`friend`, 77
`fscanf`, 113
`fseek`, 114
`fwrite`, 114

`g++`, 10
`getchar`, 26
`gets`, 27

héritage, 85
  privé, 90
  protégé, 90

public, 85
hex, 32

identificateurs, 11
if ...else, 39
ifstream, 118
inline, 47
instancier, 69
int, 12
interface, 72
istream, 117

Kerningham, 6

liste d'initialisations, 71
listes chaînées, 93
long, 12
lvalue, 19

méthode, 69
méthode virtuelle pure, 104
manipulateurs, 32, 120
mise en œuvre, 72
mots réservés, 11

new, 95

objet, 69
oct, 32
ofstream, 118
opérateur
arithmétiques, 17
associativité, 23
conditionnel, 23
d'affectation, 19
d'incrémentation, 20
de décrémentation, 20
de séquence, 20
de taille, 20
delete[], 56
delete, 56, 79
logiques, 22
new, 55, 73, 79
priorité, 23
relationnels, 21
sur les bits, 18
surcharge, 76
opérateurs, 17
ostream, 116

paramètres
effectifs, 43
formels, 43
passage de paramètre
par référence, 44
par valeur, 43
pointeur, 54
polymorphisme, 91
préprocesseur, 9
printf, 27
private, 86
putchar, 26
puts, 27

récursif, 97
répétitions, 35
résolution de portée, 86
register, 16
Ritchie, 6

scanf, 29
setbase, 32
setfill, 32
setprecision, 32
setw, 32
short, 12
sortie standard, 25
sous–programmes, 42
spécification, 72
sscanf, 113
static, 16
stdin, 25
stdout, 25
strcat, 53
strcmp, 52
strcpy, 52
strlen, 52
Stroustrup, 6
structure, 57
accès aux membres, 58
déclaration, 58
définition, 57
structures de commandes, 34
structures imbriquées, 58
switch ...case, 40

tableaux, 49
tableaux de caractères, 50
tableaux de dimension 2, 53
tableaux de structures, 59
template, 106
throw, 127
toupper, 26
true, 15
try..catch, 126

type
  caractère, 11
  entier, 12
  réel, 12

`union`, 65
unions, 65
`unsigned`, 12

variables, 15
`virtual`, 91
`volatile`, 16

`while`, 35