CNAM Montpellier

Année universitaire : 2006-07
Cycle : A
Unité d'Enseignement : Algorithmique – Programmation
(NFA001, NFA002, NFA005)
Nature : cours

Algorithmes et programmes en C++

Support de cours
novembre 2006

Enseignant : REITZ Philippe

Adresse : LIRMM
161, rue Ada
34392 Montpellier cedex 5

Mél : reitz@lirmm.fr
Tél : +33 662 32 20 50

1. INTRODUCTION

L'objectif de ce cours est de s'initier à la construction de programmes informatiques. Une telle construction s'effectue en deux phases :

- définition abstraite du programme : implique en particulier l'écriture d'algorithmes, dans un langage pas nécessairement compréhensible par un ordinateur ; nous parlerons de **langage de spécification** (ou de définition) d'algorithmes.
- définition concrète du programme : écriture d'un texte traduisant les algorithmes précédents en un langage compréhensible par l'ordinateur ; nous parlerons alors de **langage de programmation**.

1.1. UNE INTERPRETATION DE LA POSITION DU CNAM

L'une des ambitions du CNAM Paris pour cette unité de valeur *Algorithmique – Programmation* du cycle A est d'user d'un langage qui permette à la fois de spécifier et de programmer des algorithmes. Des langages fonctionnels comme CAML répondent en partie à cette ambition : des règles d'écriture (syntaxe) réduites au minimum, une grande puissance d'expression (peu de lignes suffisent en général pour décrire des algorithmes, même compliqués), et la spécification obtenue est immédiatement opérationnelle. L'approche fonctionnelle souffre toutefois de deux défauts :

- elle considère un programme comme une composition de fonctions, et met donc le concept de fonction au cœur du système de pensée. Tout individu désireux de s'approprier cette approche doit donc être familier de ce concept.
- les langages de programmation fonctionnelle sont très peu exploités dans le monde professionnel, plus friand d'autres approches de la programmation, en particulier la programmation par objets, très prisée ces temps-ci.

Le pari du CNAM Paris est que l'investissement par les auditeurs du CNAM dans un langage comme CAML leur donnera de bonnes bases de programmation, et qu'ils ne seront pas tentés par les mauvaises habitudes observées chez des programmeurs chevronnés, qui ralentissent le temps de développement des programmes ; ces mauvaises habitudes se traduisent en effet par un très grand nombre d'erreurs, coûteuses à corriger.

Afin de donner un verni moins académique à cette approche de la programmation, le CNAM Paris fait suivre cet apprentissage de la programmation fonctionnelle par une approche plus classique, la programmation impérative. Cette dernière comble le second défaut (elle est très utilisée dans le monde professionnel), mais perd dans la pureté de la description du programme : de nombreuses considérations techniques (choix de réalisation) viennent interférer dans l'écriture des algorithmes, et il devient difficile de séparer ce qui relève de l'algorithmique de ce qui relève de choix techniques.

Le langage impératif préconisé par le CNAM Paris est ADA, reconnu par toute la communauté (professionnelle et académique) comme un langage rigoureux et puissant. L'un de ses concurrents est le langage C++. Ces deux langages ont pour intérêt de mixer deux approches de la programmation : une approche impérative et une approche objet. Cette dernière pénètre aujourd'hui en profondeur tous les secteurs du monde professionnel (le monde académique a été séduit depuis longtemps…), mais son apport reste presque nul quand aux aspects algorithmiques. Pour résumer, une approche objet d'un programme conduit à le penser en terme d'objets qui interagissent pour résoudre un problème donné. Autrement dit, c'est sur l'aspect décomposition d'un problème qu'elle focalise l'attention. Toutefois, à un moment donné, il faut bien résoudre un sous problème, et donc décrire un algorithme.

Une critique souvent formulée par les auditeurs ayant assisté à ce cours à Montpellier (CAML puis ADA) est que la première partie en CAML est perçue comme inutile : pourquoi écrire un algorithme en CAML pour le traduire ensuite en ADA ? Autant écrire directement le programme en ADA, puisque si problème de conception il y a, ce problème apparaîtra dans les deux versions.

Après diverses évolutions, nous avons choisi à Montpellier d'adopter C++ comme langage de programmation. Il sera aussi le langage de spécification pour des algorithmes décrits dans une forme impérative. Nous adopterons un langage ad hoc pour décrire des algorithmes récursifs. Clairement, le langage C++ n'atteint pas la pureté de description des algorithmes, comme peuvent le faire des langages comme CAML. Dans cette unité de valeur, seuls les aspects impératifs de C++ sont étudiés. Très peu d'éléments seront donnés sur ses aspects orientés objets, afin que les auditeurs qui le souhaitent puissent se référer aux ouvrages de référence du CNAM Paris sans être trop dépaysés.

Les premiers programmes décrits dans ce support de cours ne sont pas opérationnels en l'état ; il leur manque un enrobage qui ne sera présenté que plus tard, durant les Travaux Pratiques (directives d'inclusion pour le pré-processeur, et fonction `main` pour les blocs).

1.2. GENERALITES

Un **algorithme** est un **plan d'actions**, décrit selon un certain **langage**. Autrement dit, c'est une description d'une méthode à mettre en œuvre pour résoudre un problème.

Exemple : une recette de cuisine est un algorithme : en suivant scrupuleusement ses instructions, la recette vous garantit l'obtention d'un bon petit plat !

Résoudre un **problème** c'est, partant de son énoncé initial, transformer cet énoncé en une suite d'énoncés. Si cette suite se termine, le dernier énoncé produit s'interprète comme la **solution**. Chaque transformation d'un énoncé en son successeur résulte de l'**application** d'une **opération élémentaire** de résolution. Un algorithme spécifie justement dans quel ordre doivent être appliquées ces opérations élémentaires pour obtenir la solution à un problème donné, quand elle existe.

La notion d'énoncé est attachée à celle de **langage** ; intuitivement, un énoncé est une **phrase** ou séquence de phrases exprimée dans un langage, par exemple la langue française. Cette langue s'appuie d'abord sur des caractères (lettres de l'alphabet, chiffres, signes de ponctuations), lesquels s'assemblent pour former des mots, lesquels forment des phrases, puis des textes.

La notion de langage a été formalisée plus mathématiquement par N. Chomsky : un langage est un ensemble de **mots**, chaque mot étant une séquence de **symboles**. L'ensemble des symboles admissibles est appelé l'**alphabet** du langage. La **syntaxe** d'un langage est l'ensemble des règles qui permettent de construire des mots corrects ; cette syntaxe est décrite sous forme d'une **grammaire**. La **sémantique** d'un langage consiste à préciser comment ces mots, lorsqu'ils sont correctement formés, doivent être interprétés (quel est leur sens, i.e. leur sémantique). En général, interpréter un mot, c'est effectuer une suite d'actions qui traduit le sens qui lui a été attribué.

Exemple : le langage qui permet de construire des expressions arithmétiques possède une syntaxe du type :

- un chiffre est l'un des caractères 0, 1, …, 9
- un nombre est une succession de chiffres, éventuellement précédée du caractère + ou -
- un nombre est une expression arithmétique
- si *a* et *b* sont deux expressions, alors (-*a*), *a*+*b*, *a*-*b*, … sont des expressions
- etc.

Voici des exemples de constructions correctes (i.e. chacune forme un mot du langage) :

-125+(34-87) (2)+(4+(4))

Les mots suivants ne font pas partie du langage des expressions arithmétiques :

(34 : parenthèse fermante manquante

89a+3 : les lettres ne sont pas autorisées

La sémantique de ce langage précise comment calculer la valeur d'une expression :

- la valeur d'une expression composée d'un nombre est ce nombre
- la valeur d'une expression de la forme *a*+*b* est le résultat de l'addition de la valeur de l'expression *a* avec la valeur de l'expression *b*.
- etc.

Réécrit en termes d'instructions exécutables dans un langage de programmation (c'est à dire codé), l'algorithme devient **programme**, autrement dit un énoncé compréhensible par la machine.

2. NOTIONS DE VARIABLE ET DE TYPE

Un programme qui s'exécute dans un ordinateur est un processus qui transforme le contenu de la mémoire. Chaque opération élémentaire ne modifie que quelques (i.e. un nombre fini de) cases mémoire à la fois. Cette façon d'observer un programme est trop primitive, car trop proche de la réalité des machines, c'est à dire à un niveau de détail beaucoup trop fin (c'était le point de vue adopté aux débuts de l'informatique, jusque dans les années 60).

Une vision plus abstraite, développée ci-après, consiste à considérer qu'un programme manipule des ensembles structurés de variables dont les valeurs évoluent, appelés environnements.

2.1. VARIABLE

Une **variable** se caractérise par son **nom**, sa **valeur** et son **type**. En pratique, une variable correspond à une zone de la mémoire centrale, i.e. un ensemble contigu de cases mémoire. Ainsi le nom d'une variable est une abstraction de celle d'**adresse** de la première case de la zone mémoire qui lui correspond. En revanche, la valeur d'une variable est une abstraction de la notion de **contenu** de cette même zone. La règle permettant d'interpréter le contenu effectif des cases mémoire associées comme une valeur est définie par le **type**.

Exemple : indiquer qu'une variable de nom *X* et de type entier a pour valeur le nombre 517, c'est, en termes concrets pour l'ordinateur :

- spécifier comment sont codés ces entiers ; posons qu'un entier est codé sur 2 octets consécutifs, sachant que, si le premier octet vaut a ($0 \leq a \leq 255$) et le second b ($0 \leq b \leq 255$), alors le nombre qui est codé vaut $256 \times a + b$.
- indiquer à quelle adresse est stockée *X* ; posons qu'il s'agit de l'adresse 125
- en déduire le contenu des deux cases d'adresses 125 et 126 : dans la case 125 est écrit le nombre 2, et en 126 est écrit 5. En effet, compte tenu du codage des entiers ci-dessus, le nombre représenté est $256 \times 2 + 5$, soit 517. Il est aisé de vérifier que, compte tenu des contraintes posées sur a et b, ce sont les seules valeurs possibles pour a et b permettant de coder 517.

Le nom d'une variable doit respecter les règles suivantes :

- le nom commence nécessairement par une lettre (minuscule ou majuscule) ou un blanc souligné (_)
- cette lettre peut être suivie de n'importe quelle séquence des caractères suivants : une lettre, un chiffre, un blanc souligné (_)
- un mot clé du langage C++ (par exemple **int**, **float**, **struct**, **if**) ne peut pas servir de nom de variable.

La casse des lettres (le fait qu'elles soient minuscules ou majuscules) est significative : le nom `vA` est différent du nom `va`.

2.2. TYPE

Le **type** d'une variable définit la nature de sa valeur : **nombre** (réel, entier), **caractère**, **chaîne** de caractères (ou **texte**), ou encore une **valeur de vérité** (ou **booléen**), etc. En terme d'informatique, choisir un type pour une variable, c'est nécessairement se donner les règles permettant de coder toute valeur du type, et donc d'interpréter (i.e. décoder) des données stockées en mémoire comme des valeurs.

Dans tout langage de programmation, il existe des types prédéfinis, et des constructions de types que l'utilisateur peut exploiter pour définir de nouveaux types.

2.2.1. Types prédéfinis en C++

Les types prédéfinis en C++ permettent de représenter :

- des nombres entiers ou réels
- des caractères ou des textes
- des valeurs de vérité (ou booléens)
- la notion de rien, i.e. aucun résultat

2.2.2. Les types entiers

Le langage C++ possède plusieurs types entiers, adaptés aux informations manipulées au niveau du processeur. Le tableau suivant résume ces types, et indique pour chaque type le nombre d'octets nécessaires au codage d'une valeur, ainsi que le domaine de définition du type (les bornes inférieure et supérieure des entiers représentables) :

	Signé (**`signed`**)	Non signé (**`unsigned`**)
Entier court *Codé sur 1 octet*	**`signed short int`** -128 .. 127	**`unsigned short int`** 0 .. 255
Entier standard *Codé sur 2 octets*	**`signed int`** -32 768 .. 32 767	**`unsigned int`** 0 .. 65 536
Entier long *Codé sur 4 octets*	**`signed long int`** - 2 147 483 648 .. 2 147 483 647	**`unsigned long int`** 0 .. 4 294 967 295

Il existe 2 variantes du type **`int`** standard, une variante courte (**`short`**) et une variante longue (**`long`**). Pour chacune de ces trois variantes, il en existe une version signée (**`signed`**) ou non signée (**`unsigned`**).

Le mot clé **`signed`** peut être omis. Dans le cas d'une variante courte ou longue, le mot clé **`int`** peut être omis. Autrement dit, sur quelques exemples :

- **`signed long int`** peut être abrégé en **`long`**
- **`signed int`** peut être abrégé en **`int`**
- **`unsigned short int`** peut être abrégé en **`unsigned short`**

De façon générale, l'obtention des bornes du domaine de définition d'un type entier suit le schéma suivant : si *n* est le nombre de bits nécessaire au codage (1 octet = 8 bits), alors :

- si le type entier est signé, les bornes sont -2^{n-1} .. $2^{n-1}-1$; en effet, 1 bit est consommé pour coder le signe (0 si positif, 1 si négatif) ; il reste donc *n*-1 bits pour coder le nombre.
- si le type est non signé, ces bornes deviennent 0 .. $2^{n}-1$

L'occupation mémoire de ces types est caractéristique des technologies 16 ou 32 bits de ces dernières années. Le passage à des processeurs 64 bits devrait provoquer un doublement des tailles indiquées (2 octets pour les entiers courts, 4 pour les standard et 8 pour les longs).

Les littéraux

Un **littéral entier** est l'écriture d'une valeur entière. En C++, il existe plusieurs manières de désigner un entier :

- s'il s'agit d'une écriture décimale d'un nombre, un littéral entier est une simple séquence de chiffres (`0` à `9`), éventuellement précédée d'un signe (`+` ou `−`)

Exemple : 10 +10 -100

- s'il s'agit d'une écriture octale d'un nombre, un littéral entier est une simple séquence de chiffres (`0` à `7`) commençant nécessairement par le chiffre `0`, éventuellement précédée d'un signe (`+` ou `−`) ; techniquement, le premier `0` est appelé **préfixe** du littéral.

Exemple : 010 (8 en décimal) +010 (idem) -010 (-8 en décimal)

- s'il s'agit d'une écriture hexadécimale d'un nombre, un littéral entier est une simple séquence de chiffres (`0` à `9`) ou de lettres (`A` à `F`, majuscules ou minuscules) commençant nécessairement par `0x`, éventuellement précédée d'un signe (`+` ou `−`)

Exemple : 0x1F (31 en décimal) +0x10 (16 en décimal) -0x10 (-16 en décimal)

Par défaut, de tels littéraux sont de type **`int`** s'ils sont dans les bornes du type, de type **`long`** sinon.

Il est possible de forcer un littéral entier à être considéré comme d'un type non signé en le faisant suivre d'un `U` (majuscule ou minuscule) ; techniquement, `U` est appelé **suffixe** du littéral.

Exemple : 0x1FU +10U -010U

Il est possible de forcer un littéral entier à être considéré comme d'un type long en ajoutant le suffixe `L` (majuscule ou minuscule).

Exemple : 0x1FL +10L -010L

Les deux suffixes `U` et `L` sont combinables (ordre sans importance).

Exemple : 0x1FUL représente le nombre 31 codé comme un entier long non signé

Opérateurs prédéfinis

Si *a* et *b* sont deux entiers d'un même type entier *T*, alors chacune des opérations ci-après retourne un résultat de type *T*, quand il existe :

$+a$	retourne *a* (identité)
$-a$	retourne l'opposé de *a*
$a+b$	retourne la somme de *a* et *b*
$a-b$	retourne la différence de *a* par *b*
$a*b$	retourne le produit de *a* par *b*
$a<<b$	retourne le résultat de $a*2^b$
a/b	retourne le quotient de la division entière de *a* par *b*
$a>>b$	retourne le résultat de $a/2^b$ (quotient de la division entière)
$a\%b$	retourne le reste de la division entière de *a* par *b*

Exemple : $17/3$ vaut 5, alors que $17\%3$ vaut 2. De même, $3<<2$ vaut 12 ($3*2^2$), alors que $25>>3$ vaut 3 ($25/2^3$)

Lorsque ces opérateurs sont combinés les uns avec les autres, des règles de priorité et d'associativité permettent de lever toute ambiguïté sur l'ordre des calculs à effectuer (voir la section 3.2).

Règles de transformation de type

Le compilateur C++ peut automatiquement transformer une valeur entière d'un type donné *T* en une valeur équivalente d'un autre type entier *T'*, si le contexte l'exige.

Exemple : 1 peut être transformé en 1L, sans perte de précision ; l'inverse est également vrai. Mais 1 000 000 001L ne peut être converti en une valeur équivalente du type **`short int`**.

Si la conversion de type est impossible, soit la conversion opère, avec à la clef un résultat incohérent, soit votre programme s'arrête avec une levée d'exception : une erreur est générée, qu'il est possible de traiter pour éventuellement poursuivre le programme. Le comportement dépend des options du compilateur C++ utilisé.

2.2.3. Les types réels

Il existe 3 types C++ permettant de coder les réels :

Type	Désignation en clair	Codage	Bornes positives	Précision
`float`	simple précision	4 octets	10^{-39} à 10^{38}	7 chiffres
`double`	double précision	8 octets	10^{-308} à 10^{308}	16 chiffres
`long double`	double précision longue	10 octets	10^{-4932} à 10^{4932}	19 chiffres

Tout nombre réel *x* peut être écrit sous forme **flottante**, c'est à dire $x = m \times 10^e$:

- $-1<m<1$ est la **mantisse** de *x*, sachant que le chiffre des dixièmes de *m* doit impérativement être non nul, sauf si $x=0$

- *e* est **l'exposant** de *x*

Exemples :

Nombre	Notation flottante	Mantisse	Exposant
1	$0{,}1 \times 10^{1}$	0,1	1
0	0×10^{0}	0	0
765,27	$0{,}76527 \times 10^{3}$	0,76527	3
-0,0086	$-0{,}86 \times 10^{-2}$	-0,86	-2
π	$0{,}31415927\ldots \times 10^{1}$	0,31415…	1

Du point de vue informatique, la forme flottante est redéfinie afin de mieux se conformer au système binaire : tout nombre *x* sera mis sous la forme $m \times 2^{e}$, les contraintes sur *m* restant celles énoncées ci-dessus. Seuls les *n* premiers chiffres de la mantisse sont retenus (notion de *précision*), *n* dépendant du codage considéré ; par exemple, un réel $x = m \times 2^{e}$ selon le type **`float`** est codé sur 4 octets :

- 1 octet est exploité pour coder l'exposant *e*, limité à l'intervalle –128 .. 127
- 3 octets sont exploités pour coder la mantisse *m*. Ces trois octets codent un nombre entier *n* limité à l'intervalle $-\ 8\ 388\ 608$ .. 8 388 607, la mantisse valant alors $n \div 8\ 388\ 608$ (noter que $8\ 388\ 608 = 2^{23}$, avec $23 = 3 \times 8 - 1$)

Notons que nous pouvons retrouver les informations annoncées pour ce type : la précision obtenue est $1 \div 8\ 388\ 608$, soit environ 10^{-7}, l'exposant permettant de représenter des réels allant de 2^{-128}, soit environ 10^{-39}, à 2^{127}, soit environ 10^{38}.

Exemple : le codage du nombre π est tel que l'exposant vaut 2 et la mantisse 6 588 397. En effet, c'est la meilleure approximation possible : $\pi \approx (6\ 588\ 397 \div 8\ 388\ 608) \times 2^{2}$

Pour le type **`double`**, l'exposant est codé sur 11 bits (il en reste 53 pour coder la mantisse) ; pour le type **`long double`**, 15 bits codent l'exposant (restent 65 bits pour la mantisse).

Les littéraux

Un **littéral flottant** est l'écriture d'une valeur flottante. Un tel littéral se construit ainsi :

1. écrire un nombre entier, éventuellement signé
2. faire suivre impérativement du caractère '**.**' (point)
3. faire suivre éventuellement d'un entier non signé
4. faire suivre éventuellement de la lettre `E` (ou `e`) suivie d'un entier signé ou non

Les points 1 à 3 permettent d'écrire la mantisse, le dernier l'exposant. La mantisse peut avoir n'importe quelle valeur.

Exemples : 1.0 +1.0 -10.0 1. -1.0E4 31415.927E-4

Par défaut, de tels littéraux sont de type **`double`**.

Il est possible de forcer un littéral flottant à être considéré comme un **`float`** en le suffixant par un `F` (majuscule ou minuscule).

Exemple : 1.0F +1.0E-20F

Il est possible de forcer un littéral flottant à être considéré comme **`long double`** en ajoutant le suffixe `L` (majuscule ou minuscule).

Exemple : 1.0L +1.0E-20L

Opérateurs prédéfinis

Si *a* et *b* sont deux flottants d'un même type *T*, alors chacune des opérations ci-dessous retourne un résultat de type *T*, quand il existe :

$+a$	retourne *a* (identité)
$-a$	retourne l'opposé de *a*
$a+b$	retourne la somme de *a* et *b*
$a-b$	retourne la différence de *a* par *b*
$a*b$	retourne le produit de *a* par *b*
a/b	retourne la division de *a* par *b*

Exemple : 17.0/3.0 retourne 5.666667

Lorsque ces opérateurs sont combinés les uns avec les autres, des règles de priorité et d'associativité permettent de lever toute ambiguïté sur l'ordre des calculs à effectuer (voir la section 3.2).

Toutes les fonctions mathématiques classiques (*sinus*, *racine carrée*, etc.) nécessitent d'inclure les déclarations du fichier d'en-tête `cmath` (partie développée en TP) :

```
#include <cmath>
```

Règles de transformation de type

Si un flottant était attendu dans une expression de calcul, alors que la valeur effective est un entier, cet entier est automatiquement converti en un flottant l'approchant au mieux ; l'inverse est possible également. Au cours de l'opération de transformation, il peut y avoir perte de précision, et donc dégradation de l'information.

Exemples :

1.0 est converti en 1 si nécessaire, sans perte de précision

1 est converti en 1.0 si nécessaire, sans perte de précision

4 000 000 001L est converti en 4.0E+9F, avec perte de précision

-1.0E-4 est converti en 0, avec perte de précision

1.0E+20 ne peut pas être converti en un entier long

Il existe aussi des règles de transformation de type entre flottants ; à l'évidence, passer d'un flottant moins précis (par exemple **`float`**) à un flottant plus précis (par

exemple **`long double`**) ne pose pas de problème, alors que l'inverse peut conduire à une perte de précision, voire une erreur.

2.2.4. Le type booléen

Pendant une époque, contrairement à la plupart des langages évolués, C++ ne définissait pas de type particulier pour représenter les valeurs de vérité, encore appelées **booléens**. Ce n'est plus le cas depuis sa normalisation ISO (norme ISO/IEC 14882, publiée en 1998, révisée en 2003) : ce type est noté **`bool`**.

Les littéraux

Il n'existe que deux littéraux désignant des valeurs de vérité : vrai et faux, respectivement notées **`true`** et **`false`** en C++.

Opérateurs prédéfinis

Si *a* et *b* sont deux booléens, alors chacune des opérations ci-après retourne un booléen :

`!`*a*	retourne la négation de *a*
a`&&`*b*	retourne la conjonction (ET logique) de *a* et *b*
a`||`*b*	retourne la disjonction (OU logique) de *a* et *b*

Par définition, si *a* est un booléen quelconque :

`!` **false** vaut **true**
`!` **true** vaut **false**
false `&&` *a* vaut **false**
true `&&` *a* vaut *a*
faux `||` *a* vaut *a*
true `||` *a* vaut **true**

Si *a* et *b* sont deux valeurs d'un même type *T* prédéfini, alors chacune des opérations ci-après retourne un booléen :

a`==`*b*	retourne vrai si *a* égale *b*, faux sinon
a`!=`*b*	retourne vrai si *a* diffère de *b*, faux sinon
a`>`*b*	retourne vrai si *a* est strictement plus grand que *b*, faux sinon
a`>=`*b*	retourne vrai si *a* est plus grand ou égal à *b*, faux sinon
a`<`*b*	retourne vrai si *a* est strictement plus petit que *b*, faux sinon
a`<=`*b*	retourne vrai si *a* est plus petit ou égal à *b*, faux sinon

Si *a* et *b* sont deux valeurs d'un même type *T* quelconque, et *t* est un booléen, alors :

t `?` *a* `:` *b* retourne *a* si *t* est vrai, sinon retourne *b*

Exemple : `1==0 ? 1.0 : 3.0` retourne 3.0, car l'égalité testée est fausse

Règles de transformation de type

En C++, tout booléen peut être converti en un entier : **`false`** est transformé en l'entier 0, **`true`** en l'entier 1. A l'inverse, tout entier nul est considéré comme **`false`** dans un contexte booléen, et tout entier non nul comme **`true`**.

Extension des opérateurs booléens aux entiers

Tout nombre, en particulier entier, est codé sur un certain nombre d'octets, donc de bits (éléments binaires). Les opérateurs logiques booléens présentés précédemment (`&&`, `||`, `!`) produisent un résultat booléen (en principe un bit devrait suffire pour le coder). Comme il n'est pas rare de trouver des calculs logiques manipulant des tableaux (des juxtapositions) de booléens, il peut s'avérer judicieux de coder par exemple un tableau de 16 booléens sous la forme d'un entier de type **`int`**, qui nécessite 2 octets, soient 16 bits. L'entier résultant ne se manipule plus exactement comme un nombre entier classique, mais comme une juxtaposition de bits.

Une extension assez naturelle consiste donc à définir des opérations logiques qui manipulent désormais les entiers vus comme des tableaux de bits. Les opérateurs suivants étendent donc en C++ le jeu des opérateurs booléens sur les entiers :

- *~a* retourne un entier (appelé **complément à 1** de *a*) dont chaque bit est obtenu par négation du bit de même position dans *a*.

  Cet opérateur permet de définir le codage des nombres entiers négatifs : si *n* est un entier négatif, son codage binaire est obtenu en calculant 1+~-*n* (prendre *n*, changer son signe, calculer son complément à 1, puis ajouter 1) ; l'entier naturel obtenu est alors appelé **complément à 2** de *n*.
- *a*`&`*b* retourne un entier dont chaque bit est obtenu par conjonction logique (ET logique) des bits de même position dans *a* et *b*
- *a*`|`*b* retourne un entier dont chaque bit est obtenu par disjonction logique (OU logique) des bits de même position dans *a* et *b*
- *a*`^`*b* retourne un entier dont chaque bit est obtenu par disjonction logique exclusive (OU exclusif logique) des bits de même position dans *a* et *b*. La propriété suivante permet de définir cet opérateur :

$$a \hat{} b = (a \,\&\, (\sim b)) \mid ((\sim a) \,\&\, b)$$

Exemple : `9&3` vaut 1, `9|3` vaut 11, et `9^3` vaut 10.

2.2.5. Les types caractères et chaînes

Une valeur caractère permet de représenter n'importe quel caractère, saisissable au clavier ou non, affichable en tant que caractère à l'écran ou non, ou y provoquant une action (par exemple son effacement, ou l'émission d'un bip). Le type des caractères est noté **`char`** (*character* en anglais). Un code entier est associé à chaque caractère.

En général, le code exploité est l'ASCII (codage sur 1 octet) :

Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.	Cod.	Car.
		40	(	50	2	60	<	70	F	80	P	90	Z	100	d	110	n	120	x
		41	)	51	3	61	=	71	G	81	Q	91	[	101	e	111	o	121	y
32		42	*	52	4	62	>	72	H	82	R	92	\	102	f	112	p	122	z
33	!	43	+	53	5	63	?	73	I	83	S	93	]	103	g	113	q	123	{
34	"	44	,	54	6	64	@	74	J	84	T	94	^	104	h	114	r	124	|
35	#	45	-	55	7	65	A	75	K	85	U	95	_	105	i	115	s	125	}
36	$	46	.	56	8	66	B	76	L	86	V	96	`	106	j	116	t	126	~
37	%	47	/	57	9	67	C	77	M	87	W	97	a	107	k	117	u		
38	&	48	0	58	:	68	D	78	N	88	X	98	b	108	l	118	v		
39	'	49	1	59	;	69	E	79	O	89	Y	99	c	109	m	119	w		

Le tableau précédant ne montre que les caractères affichables. Les européens lui préfèrent l'ISO-8859-1, encore appelé Latin-1, lequel enrichit le code ASCII des lettres accentuées ; ces codages doivent bientôt être remplacés par Unicode (codage sur 2 octets) au niveau international.

Une chaîne de caractère est un texte, i.e. une séquence de caractères. Par convention, toute chaîne de caractère est représentée comme la séquence des caractères qui la composent suivie du caractère de code 0. Le type des chaînes de caractères est **`char*`** (il s'agit d'un **tableau** de caractères, comme nous le verrons par la suite).

Il y a équivalence entre le type **`char`** et le type **`short int`** : tout caractère est équivalent à son code entier. Il est donc possible de signer (**`signed char`**, soit **`char`**) ou non (**`unsigned char`**) les caractères.

Les littéraux

Tout littéral caractère indique le caractère souhaité encadré d'apostrophes (')

Exemple : `'A' 'a' ' ' '1' ','` sont des littéraux caractères

Certains caractères spéciaux peuvent être écrits via le caractère dit d'**échappement** **`'\'`** (liste non exhaustive) :

Caractère	Caractère représenté
`\\`	\
`\'`	apostrophe '
`\a`	*alert* (code 7)
`\b`	*backspace* (code 8)
`\t`	tabulation (code 9)
`\n`	*new line* (code 10)
`\r`	*carriage return* (code 13)
`\"`	guillemet "
`\0`*ccc*	code donné en octal (chaque *c* est un chiffre 0..7)
`\`*nnn*	code donné en décimal (chaque *n* est un chiffre 0..9)
`\0x`*hhh*	code donné en hexadécimal (chaque *h* est 0..9 ou A..F)

Exemple : `'\0'` est le caractère de code 0, `'\''` est le caractère ', `'\\'` est le caractère \

Un littéral chaîne de caractères spécifie tous les caractères qui composent la chaîne encadrés de guillemets.

Exemple : `"Chaîne contenant:\n\t- un guillemet (\")\n\t- une lettre : \65."` est la chaîne de caractère qui, lorsqu'elle est affichée, produit à l'écran :

```
Chaîne contenant:
        - un guillemet (")
        - une lettre : A.
```

Chaque `\n` force un retour à la ligne suivante, chaque `\t` produit un décalage à droite (tabulation). Le code de la lettre finale (A) est donné en décimal (code ASCII).

Les opérateurs prédéfinis

Il n'existe pas d'opérateurs prédéfinis sur ces types, en dehors de ceux définis sur les types entiers (puisqu'il y a équivalence caractère – entier).

Règles de transformation de type

Pour le type **`char`**, ce sont les mêmes que celles des types entiers. Les règles portant sur **`char*`** sont celles des pointeurs (cf. chapitre 9).

2.2.6. Le type vide

Ce type particulier, noté **`void`**, sert à représenter l'absence de résultat ; il n'a de sens que pour les fonctions (une fonction peut ne pas avoir de résultat, ou n'avoir besoin d'aucun paramètre). Il ne possède ni valeur (donc en particulier pas de littéraux) ni opérateur.

3. NOTION D'ENVIRONNEMENT

Un **environnement** est une liste ordonnée de variables. Il existe trois règles pour construire des environnements :

- l'environnement vide est noté ø
- si (*nom*, *valeur*, *type*) est une variable et *E* un environnement,
  alors (*nom*, *valeur*, *type*) • *E* est un environnement.
- si *E* un environnement,
  alors ✸ • *E* est un environnement. Le symbole ✸ est appelé **marque de bloc**.

3.1. OPERATIONS DE BASE SUR LES ENVIRONNEMENTS

Nous définissons 5 opérations sur les environnements :

- lire(*n*, *E*) : lecture de la valeur d'une variable de nom *n* dans un environnement *E*. La valeur retournée est celle de la première variable de l'environnement qui porte le nom indiqué. Si aucune variable ne porte le nom indiqué, retourne une erreur.

Une définition plus précise :

lire(*n*, *E*)	= erreur	si *E* = ø
	= *v*	si *E* = (*n*, *v*, *t*) • *F*
	= lire(*n*, *F*)	si *E* = (*m*≠*n*, *v*, *t*) • *F* ou *E* = ✸ • *F*

Exemple : valeur lue pour la variable d'un nom spécifié dans un environnement donné

Environnement *E*	Opération	Valeur lue
(x, 2, int) • (y, 4, int) • (i, 1, int) • ø	lire(i, E)	1
(x, 3, int) • ✸ • (x, 8, int) • ø	lire(x, E)	3

Dans le premier cas, en parcourant l'environnement de gauche à droite, la première variable qui porte le nom i a pour valeur 1. Dans le second cas, toujours en parcourant de gauche à droite, la valeur trouvée est 3.

- définir(*n*, *v*, *t*, *E*) : création dans un environnement *E* d'une nouvelle variable (*n*, *v*, *t*). Le résultat est le nouvel environnement. Il y a une contrainte à la création d'une nouvelle variable : aucune variable du bloc en cours ne doit porter le même nom.

Une définition plus précise :

définir(*n*, *v*, *t*, *E*)	= (*n*, *v*, *t*) • *E*	si *E* = V_1 • V_2 • … • V_k • *F* où chaque V_i est une variable de nom≠*n* et *F* un environnement ne commençant pas par une variable (V_k est la dernière variable de la série)
	= erreur	sinon

Il se peut que k (indice de la dernière variable) soit nul, i.e. k=0, ce qui signifie que l'environnement E ne commence pas par une variable : soit E est l'environnement vide, soit il commence par une marque de bloc.

Exemple : nouvel environnement après définition d'une variable de valeur et de nom spécifiés dans un environnement donné

Environnement *E*	Opération	Environnement résultant
(x, 2, int) • (y, 4, int) • (i, 1, int) • ø	définir(i, 3, int, E)	Erreur : i existe déjà dans le bloc
(x, 3, int) • ✸ • (y, 7, int) • ø	définir(y, 8, int, E)	(y, 8, int) • (x, 3, int) • ✸ • (y, 7, int) • ø
✸ • (y, 7, int) • ø	définir(y, 8, int, E)	(y, 8, int) • ✸ • (y, 7, int) • ø
Ø	définir(y, 8, int, E)	(y, 8, int) • ø

- affecter(n, v, t, E) : modification dans un environnement E de la valeur d'une variable de nom n. Le résultat est un environnement E' identique à E sauf pour la première variable de nom n, dont la valeur est v dans E'. Le type de la valeur t doit être compatible avec celui (soit t') de la variable dans E.

Une définition plus précise :

affecter(n, v, t, E)
- = erreur si E = ø
- = ✸ • affecter(n, v, t, F) si E = ✸ • F
- = erreur si $E = (n, v', t') \bullet F$ et t incompatible avec t'
- = $(n, v', t') \bullet F$ si $E = (n, v'', t') \bullet F$ et t compatible avec t'
- = $(m, v', t') \bullet$ affecter(n, v, t, F) si $E = (m \neq n, v', t') \bullet F$

Deux remarques :

- un type t est dit *compatible* avec un type t' s'il existe une règle transformant toute valeur de type t en une valeur équivalente de type t' (règles de transformation de types). Un type est trivialement compatible avec lui-même ; la plupart des types prédéfinis sont compatibles entre eux.
- notons que ce n'est pas v qui est la nouvelle valeur de la variable, mais v', l'équivalent de v pour le type t' (notion de coercion).

Exemple : nouvel environnement après affectation de la valeur d'une variable d'un nom spécifié dans un environnement donné

Environnement *E*	Opération	Environnement résultant
(x, 2, int) • (y, 4, int) • (i, 1, int) • ø	affecter(i, 3, int, E)	(x, 2, int) • (y, 4, int) • (i, 3, int) • ø
(x, 3, int) • ✸ • (x, 7, int) • ø	affecter(x, 8.9, float, E)	(x, 8, int) • ✸ • (x, 7, int) • ø

Dans le premier cas, en parcourant l'environnement de gauche à droite, la première variable qui porte le nom i est la troisième de l'environnement ; c'est celle dont la valeur va être changée.

Dans le second cas, toujours en parcourant de gauche à droite, c'est la valeur de la première variable qui doit être modifiée ; le type **`int`** est compatible avec **`float`**, et la valeur 3 est remplacée par 8 car c'est l'équivalent de 8.9 pour le type **`int`** (troncature de la partie fractionnaire du flottant pour en faire un entier).

- ouvrir(E) : insertion d'une marque de bloc au début de l'environnement E.

Une définition plus précise :

ouvrir(E) $= ✸ \bullet E$

- fermer(E) : effacement de toutes les variables situées à gauche de la première marque de bloc d'un environnement E, y compris cette marque.

Une définition plus précise :

$$\begin{array}{lll} \text{fermer}(E) & = E & \text{si } E = ø \\ & = \text{fermer } (F) & \text{si } E = (n, v, t) \bullet F \\ & = F & \text{si } E = ✸ \bullet F \end{array}$$

Exemples :

Environnement *E*	**Environnement fermé**
(x, 2, int) • ✸ • (y, 4, int) • ✸ • (i, 1, int) • ø	(y, 4, int) • ✸ • (i, 1, int) • ø
(x, 2, int) • (y, 4, int) • ✸ • (i, 1, int) • ø	(i, 1, int) • ø
(x, 3, int) • (y, 7, int) • ø	ø

Noter que la fermeture efface aussi la première marque de bloc rencontrée, si elle existe.

Les opérations de lecture ou d'affectation ne changent pas le nombre de variables de l'environnement sur lequel elles opèrent.

3.2. EXPRESSIONS DE CALCUL ET LEUR EVALUATION

Une **expression de calcul** suit les règles de construction suivantes :

- si n est le nom d'une variable, alors n peut être considéré comme une expression de calcul
- si c est un littéral, alors c peut être considéré comme une expression de calcul
- si e est une expression de calcul, alors `(`e`)` est une expression de calcul (parenthésage)
- si e et f sont deux expressions de calcul, alors $+e$ est une expression de calcul, ainsi que $-e$, ainsi que $e+f$, $e-f$, $e*f$, etc., et de façon générale toute expression de ce type impliquant l'un des opérateurs associés aux types prédéfinis.
- si e est une expression de calcul et T un nom de type (plus généralement une expression de type), alors `(`T`)` e est une expression de calcul (conversion de type).

Exemple : voici quelques expressions de calcul valides :

`12` (littéral seul) `X` (nom de variable seul) `A*(3-B)+2`

Les opérateurs (+, -, *, etc.) peuvent être combinés, mais une difficulté peut survenir quand à leur ordre d'application :

Exemple : dans quel ordre faut-il appliquer les opérateurs pour cette expression d'entiers :

```
12*3-5/2/2*3>>3+1<<2*2+1
```

Une première façon de résoudre la difficulté consiste à ajouter des parenthèses, qui ne laissent plus aucune ambiguïté sur l'ordre d'application. Néanmoins, ces parenthèses alourdissent considérablement l'écriture, et sont source de nombreuses erreurs d'omission (il est facile d'oublier de refermer une parenthèse).

Exemple : une expression qui ne pose aucun problème d'ordre d'application des opérateurs, mais qui pose des difficultés d'écriture (il faut être sûr que toute parenthèse ouverte est bien fermée) et aussi de lecture :

```
(((12*3)-(((5/2)/2)*3))>>(3+1))<<(2*(2+1))
```

C'est exactement la version parenthésée de l'expression de l'exemple précédent.

Comme la majorité des langages de programmation, C++ autorise l'omission des parenthèses, mais impose des conventions qui permettent de reconstituer le parenthésage, basées sur deux propriétés, la **priorité** et l'**associativité**, associées à chaque opérateur.

Un opérateur est dit **unaire** s'il ne possède qu'un seul paramètre. Nous distinguerons les opérateurs **unaire à gauche** – dits encore **en position préfixe** – (de la forme ©*a* où *a* est le paramètre et © le symbole de l'opérateur) des opérateurs **unaires à droite** – dits encore **en position suffixe** – (de la forme *a*©).

Le tableau suivant (non exhaustif) résume les propriétés des différents opérateurs unaires du langage C++ (avec *T* un nom de type) :

`+a`	`!a`	`++a`	`a--`
`-a`	`*a`	`--a`	
`~a`	`&a`	`a++`	`(T) a`

Un opérateur est dit **binaire** s'il possède deux paramètres ; il est alors toujours de la forme *a*©*b*, où *a* et b sont les paramètres et © le symbole de l'opérateur ; ce symbole est dit **en position infixe** (au milieu).

Le tableau suivant résume les propriétés des différents opérateurs binaires :

opérateurs	**famille**	**priorité**	**associativité**
`a*b  a/b  a%b`	produit arithmétique	10	à gauche
`a+b  a-b`	somme arithmétique	9	à gauche
`a<<b  a>>b`	décalage	8	à gauche
`a<b  a<=b  a>b  a>=b`	relationnel	7	à gauche
`a==b  a!=b`	égalité	6	à droite
`a&b`	produit bit à bit	5	à gauche
`a|b  a^b`	somme bit à bit	4	à gauche
`a&&b`	produit logique	3	à gauche
`a||b`	somme logique	2	à gauche

Si © et ® sont deux opérateurs binaires tels que la priorité de © est plus forte que celle de ®, alors une expression de la forme *a*©*b*®*c* sera interprétée comme (*a*©*b*) ®*c* ; dans le cas contraire, elle sera interprétée comme *a*© (*b*®*c*) .

Exemple : l'expression `12*3-5/2` est en fait interprétée comme `(12*3)-(5/2)`

Si © et ® sont deux opérateurs binaires de même priorité (ils appartiennent alors à la même famille), alors une expression de la forme *a*©*b*®*c* sera interprétée comme (*a*©*b*) ®*c* si leur associativité est de type *à gauche*, et interprétée comme *a*© (*b*®*c*) si l'associativité est *à droite*.

Exemple : l'expression `12*3/5%2` est en fait interprétée comme `((12*3)/5)%2`

Les opérateurs unaires sont toujours prioritaires sur les opérateurs binaires.

Exemple : l'expression `+12*-3+-2` est en fait interprétée comme `((+12)*(-3))+(-2)`

Le seul opérateur **ternaire** du langage (*a*?*b*:*c*) est le moins prioritaire de tous.

Exemple : l'expression `+12*-3==-2?3+2:4*5/2` est en fait interprétée comme `(((+12)*(-3))==(-2))?(3+2):((4*5)/2)`

Lorsqu'il est combiné, il ne pose aucune difficulté particulière quand les parenthèses sont omises :

a?*b*:*c*?*d*:*e*	s'interprète comme	*a*?*b*:(*c*?*d*:*e*)
a?*b*?*c*:*d*:*e*	s'interprète comme	*a*?(*b*?*c*:*d*):*e*

3.2.1. Evaluation d'une expression

Si *e* est une expression de calcul, l'objectif du programme qui la contient est d'en calculer le résultat ; techniquement, nous dirons que l'expression est **évaluée**, i.e. nous cherchons à lui associer une valeur résultat. Cette expression pouvant contenir des noms de variables, l'évaluation nécessite la donnée d'un environnement *E*, permettant ainsi de récupérer les valeurs associées aux variables correspondantes.

Nous noterons *v* = évaluer(*e*, *E*) la valeur de l'expression *e* dans l'environnement *E* ; évaluer(*e*, *E*) est l'opération qui décrit comment procéder à l'évaluation, et se définit par :

- si *x* est un littéral d'un type *T* dont la valeur est *v*, alors évaluer(*x*, *E*) = *v*

- si *n* est un nom de variable, alors évaluer(*n*, *E*) = lire(*n*, *E*)

- si *a* est une expression, alors évaluer((*a*) , *E*) = évaluer(*a*, *E*)

- si *a* et *b* sont deux expressions d'un type *T* définissant l'opérateur +, alors évaluer(*a*+*b*, *E*) = évaluer(*a*, *E*) + évaluer(*b*, *E*). Attention, le + d'évaluer à gauche du signe = est de nature syntaxique, alors que le + entre les évaluer à droite du signe = est de nature sémantique (il s'agit de la somme de deux valeurs du type *T*, dont le résultat est en général une valeur de type *T*).

- idem pour tous les autres opérateurs (−, /, *, etc.), en respectant toutefois les règles de priorité et d'associativité quand il y a ambiguïté sur leur ordre d'application.

- si *a* est une expression et *T* un type, alors évaluer((*T*) *a* , *E*) = *v*, où *v* est la valeur équivalente dans le type *T* de évaluer(*a*, *E*).

Cette définition de l'opération d'évaluation évaluer n'est que provisoire ; nous y reviendrons lors de notre étude du passage par référence. De plus, nous laissons de coté le problème des types associés aux valeurs manipulées (que se passe-t-il quand les deux paramètres d'un opérateur binaire n'ont pas le même type ?).

4. NOTION D'INSTRUCTION

Une **instruction** est une action à effectuer sur un environnement. **Exécuter** une instruction, c'est transformer un environnement selon l'action correspondante :

$$E_{initial} \xrightarrow{\ i\ } E_{final}$$

Programmer, i.e. construire un programme, c'est agencer des instructions qui, exécutées une à une, dans l'ordre défini par le programme, transformeront l'environnement courant. Dans ce type d'approche, concevoir un programme, c'est anticiper comment un environnement va devoir évoluer pour aboutir, partant d'un environnement de départ, à un environnement final duquel pourra être tirée la réponse au problème que le programme est censé résoudre.

Il y a trois classes principales d'instructions : les instructions d'**affectation**, de **définition** de variables et de **contrôle** (qui se décomposent en instructions **conditionnelles** et de **boucle**).

4.1. INSTRUCTION VIDE

C'est la forme la plus simple : elle s'écrit simplement :

```
;
```

et son exécution ne modifie pas l'environnement de calcul.

4.2. INSTRUCTION DE CALCUL

Si *e* est une expression de calcul, alors :

```
e ;
```

est une instruction. Son exécution consiste simplement à calculer la valeur de *e* dans l'environnement de travail *E*, soit évaluer(*e* , *E*), sans que l'environnement ne soit modifié (ce qui sera faux dès que nous aurons étudié pointeurs et références).

4.3. BLOC D'INSTRUCTIONS

Un *bloc* d'instructions est une séquence d'instructions qui devront être exécutées dans l'ordre dans lequel elles sont écrites. Un bloc peut être vide (aucune instruction). Un bloc est lui-même une instruction.

4.3.1. Syntaxe

La syntaxe des blocs d'instructions, c'est à dire leur notation, est donnée par : si i_1, …, i_n sont *n* instructions, alors le bloc composé de ces *n* instructions est noté :

```
{
    i1 ;
    …
    in ;
}
```

Les accolades servent simplement à délimiter les instructions du bloc.

Notons que, puisque les blocs sont aussi des instructions, nous pouvons avoir des blocs imbriqués les uns dans les autres :

Exemple :

```
{
    i1 ;
    {
        {
            i3 ;
        }
    };
    i4 ;
    {
        i5 ;
    }
}
```

En principe, chaque instruction d'un bloc doit se terminer par le caractère point-virgule (`;`). Toutefois, si un sous-bloc se termine, et que son bloc englobant se termine aussi, alors la partie finale :

```
}; }
```

peut s'abréger en :

```
}}
```

4.3.2. Sémantique

La sémantique d'un bloc d'instructions, c'est à dire comment il doit être exécuté, est la suivante :

si *E* est l'environnement avant exécution et que l'on considère le bloc :

```
{
    i1 ;
    …
    in ;
}
```

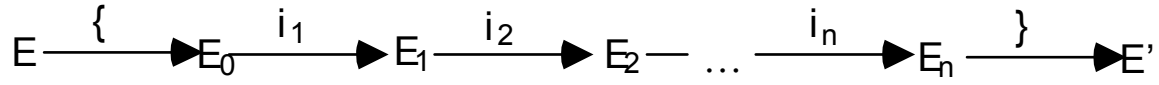
alors exécuter le bloc d'instructions consiste à :

- transformer *E* en un environnement E_0 = ouvrir(*E*) (ouverture bloc `{`)
- transformer E_0 en un environnement E_1 en exécutant i_1
- transformer E_1 en un environnement E_2 en exécutant i_2

…

- transformer E_{n-1} en un environnement E_n en exécutant i_n
- transformer E_n en un environnement *E*' = fermer(E_n) (fermeture bloc `}`)

Le résultat de l'exécution du bloc est l'environnement *E'* ; en image :



4.4. INSTRUCTION DE DEFINITION DE VARIABLES

4.4.1. Syntaxe

Si *T* est un type, *n* le nom d'une variable et *e* une expression de calcul ayant pour résultat une valeur *v* de type *T*, alors la définition d'une nouvelle variable dans l'environnement courant s'écrit :

```
T n ;
```

ou

```
T n = e ;
```

La partie `= e` est appelée **initialisation** de la variable. Si cette initialisation est omise, alors le compilateur ajoute de lui-même ou non cette initialisation, considérant qu'il s'agira de `= 0`, le 0 étant celui du type considéré (autrement dit, si vous souhaitez initialiser une variable à 0, écrivez explicitement cette initialisation).

Exemple :

```
long int varX = 1 ;
char lettre = 'A' ;
float z ;
```

Si plusieurs instructions de définition de variables d'un même type se suivent, il est possible de les fusionner en une seule instruction (définitions multiples), se présentant ainsi :

```
T d1, d2, …, dn ;
```

où chaque d_i est une définition se résumant à un nom de variable, ou un nom suivi d'une initialisation.

Exemple : ces trois définitions :

```
long int varX = 1 ;
long int varY ;
long int varZ = varX-2 ;
```

peuvent être résumées en une seule :

```
long int varX = 1, varY, varZ = varX-2 ;
```

4.4.2. Sémantique

Soit l'instruction suivante :

```
T n = e ;
```

à exécuter, et *E* l'environnement avant exécution. Après exécution, l'environnement est transformé en :

$$E' = \text{définir}(n, v, T, E)$$

où $v = \text{évaluer}(e, E)$ est la valeur de l'expression *e* dans l'environnement *E*.

Rappelons que l'opération définir vérifie que la variable n'existe pas encore dans le bloc courant ; ainsi, si la variable a déjà été définie dans le bloc courant, l'instruction provoque une erreur.

4.5. INSTRUCTION DE DEFINITION DE CONSTANTES

4.5.1. Syntaxe

Si *T* est un type, *n* le nom d'une variable et *e* une expression de calcul ayant pour résultat une valeur *v* de type *T*, alors la définition d'une nouvelle constante dans l'environnement courant s'écrit :

```
const T n = e ;
```

Exemple :

```
const long int un = 1 ;
const char lettre_A = 'A' ;
```

La partie = *e* est appelée **initialisation** de la constante, et est obligatoire.

Le nom d'une constante respecte les mêmes règles que celles des variables. Plusieurs définitions de constantes d'un même type peuvent être fusionnées en une seule, à l'instar des définitions de variables.

Exemple : ces deux définitions :

```
const long int one = 1 ;
const long int two = one+one ;
```

peuvent être résumées en une seule :

```
const long int one = 1, two = one+one ;
```

4.5.2. Sémantique

Soit l'instruction suivante :

```
const T n = e ;
```

à exécuter, et *E* l'environnement avant exécution. Après exécution, l'environnement est transformé en :

$$E' = \text{définir}(n, v, \texttt{const}\ T, E)$$

où v = évaluer(e, E) est la valeur de l'expression *e* dans l'environnement *E*. Attention : l'opération affecter(n, v, t, E) de modification de valeur d'une variable dans un environnement retourne une erreur si cette variable est une constante, i.e. d'un type du genre `const` *T*.

4.6. INSTRUCTION D'AFFECTATION

Une instruction d'**affectation** permet de remplacer la valeur d'une variable d'un environnement par le résultat du calcul d'une expression (le résultat est dit *affecté* - ou *assigné* - à la variable). Une affectation peut modifier une variable, mais pas une constante.

4.6.1. Syntaxe

Sa première forme est la suivante :

```
nom = expression ;
```

où *nom* est le nom d'une variable, suivie d'une *expression* de calcul quelconque.

4.6.2. Sémantique

Sa sémantique est la suivante : l'expression est évaluée dans l'environnement courant, le résultat étant une valeur qui est écrite dans la variable dont le nom est spécifié. L'environnement résultant devient le nouvel environnement courant.

Autrement dit : soit *E* l'environnement de départ, l'instruction à exécuter étant :

```
n = e ;
```

Alors l'environnement est transformé en :

$$E' = \text{affecter}(n, v, T, E)$$

où $v = \text{évaluer}(e, E)$ valeur d'un type T de l'expression e dans l'environnement E.

Exemple : si l'environnement courant est (x, 3, **int**) • (y, 1, **int**) • ø, après exécution de l'instruction :

```
y = x*x - 5 ;
```

l'environnement devient (x, 3, **int**) • (y, 4, **int**) • ø

Exemple : si l'environnement courant est (x, 2, **int**) • (y, 1, **int**) • ø, après exécution du bloc :

```
{
    y = 2*x - y ;
    x = y + 5 ;
}
```

l'environnement devient (x, 8, **int**) • (y, 3, **int**) • ø.

En effet, après exécution de la première affectation, l'environnement devient (x, 2, **int**) • (y, 3, **int**) • ø. La seconde affectation transforme bien ce dernier en l'environnement annoncé.

4.6.3. Formes abrégées

Il existe plusieurs formes abrégées de l'affectation pour les types prédéfinis :

Forme abrégée	Forme équivalente
`v += e`	`v = v+e`
`v -= e`	`v = v-e`
`v *= e`	`v = v*e`
`v /= e`	`v = v/e`
`v %= e`	`v = v%e`
`v &= e`	`v = v&e`
`v |= e`	`v = v|e`
`v ^= e`	`v = v^e`
`v <<= e`	`v = v<<e`
`v >>= e`	`v = v>>e`
`v++`	`v = v+1`
`++v`	`v = v+1`
`v--`	`v = v-1`
`--v`	`v = v-1`

Les quatre dernières formes seront étudiées plus tard.

4.6.4. Forme plus générale

En fait, la forme la plus générale de l'affectation est :

expression$_1$ `=` *expression*$_2$ `;`

où *expression*$_1$ et *expression*$_2$ sont deux expressions, la première devant impérativement retourner une zone mémoire accessible en lecture/écriture (**zone affectable**, ou encore **référence** – voir chapitre 10). Un nom de variable désigne une telle zone ; par contre, un nom de constante ne convient pas.

Exemple : il est possible d'écrire :

```
(x == y ? x : y) = 3 ;
```

Si `x` égale `y`, alors `x` se voit affecter la valeur 3 ; sinon, c'est `y` qui est affecté à 3.

4.7. INSTRUCTIONS D'ENTREE – SORTIE

Nous introduisons dans cette section le minimum vital pour qu'un programme puisse afficher des résultats à l'écran ou demander à l'usager qu'il saisisse des informations. Les deux instructions présentées nécessitent que le texte C++ commence par deux lignes du genre :

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

dont le rôle sera expliqué en TP (la première est une directive d'inclusion, destinée au préprocesseur, laquelle indique de charger le fichier d'en-tête prédéfini `iostream`, lequel contient toutes les définitions utiles à l'exploitation des entrées-sorties).

4.7.1. Instruction de sortie

L'affichage du résultat d'une expression de calcul *e* à l'écran est réalisé par l'instruction suivante :

```
cout << e ;
```

Si plusieurs instructions de sortie se suivent, il est possible de les combiner en une seule ; ainsi :

```
cout << e1 ;
cout << e2 ;
cout << e3 ;
```

pourra simplement s'écrire :

```
cout << e1 << e2 << e3 ;
```

Exemple : l'exécution du bloc suivant :

```
{
    cout << "Bonjour " ;
    cout << 0.5 << ' ' << 12 << "\n" << " A" ;
}
```

provoque l'affichage :

```
Bonjour 0.5 12
 A
```

Noter que le caractère `\n` force l'affichage à se poursuivre à la ligne suivante.

Ecrire `cout << "\n"` est équivalent à `cout << endl` (end of line).

La sémantique d'une telle instruction :

```
cout << e ;
```

est la suivante : la valeur évaluer(*e* , *E*) est calculée, puis affichée à l'écran. L'environnement de travail n'est donc pas modifié.

4.7.2. Instruction d'entrée

Si *n* est le nom d'une variable de type *T* dans l'environnement, alors l'instruction d'entrée s'écrit :

```
cin >> n ;
```

Comme les sorties en cascade, plusieurs instructions d'entrée peuvent être combinées en une ; ainsi :

```
cin >> n1 ;
cin >> n2 ;
cin >> n3 ;
```

pourra simplement s'écrire :

```
cin >> n1 >> n2 >> n3 ;
```

La sémantique d'une telle instruction :

```
cin >> n ;
```

est la suivante :

1. une valeur *v* de type *T* est demandée à l'utilisateur.

2. cette valeur une fois saisie, l'environnement de travail *E* est transformé en *E'* = affecter(*n*, *v*, *T*, *E*).

Exemple : l'exécution du bloc suivant :

```
{
    int x ;
    cout << "un entier SVP : " ;
    cin >> x ;
    cout << "2*" << x << '=' << 2*x ;
}
```

produit le dialogue suivant (les caractères saisis par l'usager sont écrits en **gras souligné** ; l'appui sur la touche *Entrée* – ou touche de validation – est noté ↵) :

```
un entier SVP : 43↵
2*43=86
```

4.8. INSTRUCTIONS DE CONTROLE

Une instruction de **contrôle** indique comment l'exécution d'un bloc d'instructions doit être menée. Il existe deux formes d'instructions de contrôle : la **conditionnelle**, dont une variante appelée **sélection par cas**, et la **boucle**, dont il existe trois variantes.

4.8.1. La conditionnelle

Syntaxe

L'instruction de **test**, ou **conditionnelle**, encore appelée l'**alternative**, se présente ainsi :

```
if (test)
    bloc1
else
    bloc2 ;
```

sachant que la partie **else** est facultative, l'instruction se limitant alors à :

```
if (test)
    bloc1 ;
```

Le *test* est une expression dont le résultat doit être une valeur de vérité (booléen), c'est à dire soit vrai, soit faux.

Un *bloc* est soit une instruction seule, soit un bloc d'instructions. Attention : si $bloc_1$ est un bloc d'instructions, il ne faut pas faire suivre son accolade fermante d'un **;** avant **else**. Si, au contraire, il s'agit d'une instruction simple seule, il faut impérativement faire suivre celle-ci d'un point-virgule.

Exemple :

```
if (false) a=1 ; else a=3 ;     // ; obligatoire avant else
if (a>4) { a=3 ; } else b=4 ;   // surtout pas de ; avant else
if (b==2) { b=0 ; } ;           // ; obligatoire si pas de else
```

Sémantique

Pour la première forme, l'instruction indique que si le résultat du *test* est vrai, alors le $bloc_1$ est exécuté, et le $bloc_2$ ignoré; si le résultat est faux, alors c'est le $bloc_2$ qui est exécuté, et le $bloc_1$ ignoré. Dans les deux cas, le programme poursuit son déroulement après l'exécution de l'un des deux blocs.

Plus schématiquement : soit *E* l'environnement de départ, l'instruction à exécuter étant :

```
if (test)
    bloc1
else
    bloc2 ;
```

Alors l'environnement est transformé en *E'* ainsi :

1. calculer *t* = évaluer(*test*, *E*) valeur de l'expression *test* dans l'environnement *E*.
2. si *t* est vrai alors exécuter $bloc_1$, lequel transforme *E* en *E'*
3. sinon si *t* est faux alors exécuter $bloc_2$, lequel transforme *E* en *E'*

Exemple : étant donné un environnement contenant une variable flottante de nom `x`, une autre de nom `y`, et une dernière booléenne de nom `indéfini`, écrire une instruction produisant un environnement tel que `y = 1/x`, quand cela est possible.

```
if (x==0.0)
    indefini = true ;
else {
    indefini = false ;
    y = 1/x ;
}
```

Remarquez que la variable `indéfini` a pour valeur *vrai* si *x=0*, *faux* sinon. Autrement dit, la valeur de vérité qu'elle contient nous indique si la valeur de *y* a été calculée ou non. Si oui, *y* contient bien la valeur attendue.

4.8.2. La sélection par cas

L'instruction de **sélection par cas** est une forme particulière de conditionnelle, où plusieurs conditions s'enchaînent.

Syntaxe

```
switch (expression) {
    case expression1: bloc1
    …
    case expressionn: blocn
    default: blocn+1
};
```

sachant que la partie **`default`** est facultative.

Chaque $bloc_i$ est une séquence d'instructions (si au moins deux instructions, les accolades ne sont pas nécessaires), éventuellement vide. Chacun de ces blocs peut contenir une instruction spéciale **`break`**.

Exemple :

```
switch (a+1) {
    case 1 :          // premier cas sans instruction
    case 3 :          // second cas avec une instruction
        x = x+a ;
    case 2 :          // troisième cas avec 2 instructions ; noter l'absence
        x = x-a ;     //  d'accolades – elles auraient pu y figurer.
        break ;
    default :
        x = x+3;
};
```

Sémantique

Le schéma d'exécution suit le principe suivant : soit *E* l'environnement de départ, l'instruction à exécuter étant :

```
switch (expression) {
    case expression1: bloc1
    …
    case expressionn: blocn
    default: blocn+1
};
```

Alors l'environnement *E* est transformé en *E'* ainsi :

1. calculer v = évaluer(*expression*, *E*) valeur de *expression* dans l'environnement *E*.
2. soit i=1
3. calculer v_i = évaluer($expression_i$, *E*), valeur de $expression_i$ dans l'environnement *E*.

4.1. si $v = v_i$, alors exécuter en séquence toutes les instructions de $bloc_i$, $bloc_{i+1}$,… $bloc_{n+1}$; l'exécution de **`switch`** est alors achevée. Si, au cours de cette séquence, l'instruction **`break`** est exécutée, toutes les instructions restantes sont ignorées, et **`switch`** est immédiatement terminée

4.2. si $v \neq v_i$ et $i \neq n$, alors revenir au point 3 en ajoutant 1 à i (test du cas suivant)

4.3. si $v \neq v_n$ et $i = n$, alors exécuter en séquence toutes les instructions de $bloc_{n+1}$; l'exécution de **`switch`** est alors achevée. Si, au cours de cette séquence, l'instruction **`break`** est exécutée, toutes les instructions restantes sont ignorées, et **`switch`** est immédiatement terminée.

Exemple : posons que l'environnement courant définit deux variables `a` et `x`, où `x` = 3.

```
switch (a) {
    case 1 :          // premier cas sans instruction
    case 3 :          // second cas avec une instruction
        x = x+a ;
    case 2 :          // troisième cas avec 2 instructions ; noter l'absence
        x = x-a ;     //  d'accolades
        break ;
    default :
        x = x+3;
};
```

Si `a` vaut 0 au départ, `x` vaudra 6 après exécution : seule l'affectation du **`default`** opère, puisqu'aucun cas ne correspond à la valeur de `a`.

Si `a` vaut 1 au départ, `x` vaudra 3 après exécution : le premier cas est activé, lequel provoque l'exécution de `x = x+a` puis `x = x-a` ; l'exécution du **`break`** empêche toute exécution des instructions des cas suivants du **`switch`**

Si `a` vaut 2 au départ, `x` vaudra 1 après exécution : le troisième cas est activé, lequel provoque l'exécution de `x = x-a`, le **`break`** empêchant toute exécution des instructions des cas suivants du **`switch`**.

Si `a` vaut 3 au départ, `x` vaudra 3 après exécution : le second cas est activé, lequel provoque l'exécution de `x = x+a` puis `x = x-a`.

Remarque : dans la spécification des divers cas (**`case`** $expression_i$), chaque $expression_i$ est une expression ne devant en aucun cas contenir de nom de variable (sauf s'il s'agit de constante) ou d'appel à une fonction. Une telle expression est dite *expression constante*.

4.8.3. La boucle

L'instruction de contrôle précédente (**conditionnelle** ou **sélection par cas**) permet d'exécuter un bloc d'instructions sous condition. L'instruction de **boucle** permet d'exécuter plusieurs fois un même bloc d'instructions, et ce tant qu'aucune indication contraire n'interrompe cette boucle.

Syntaxe

Il existe deux formes de base possibles :

```
while (test)
    bloc ;
```

ou :

```
do
    bloc
while (test) ;
```

où *bloc*, aussi appelé **corps** de la boucle, peut contenir une instruction spéciale de sortie (**`break`**) ou de continuation (**`continue`**) de la boucle. L'expression *test* doit être de type booléen, et est appelée **condition d'arrêt** (ou de **terminaison**) de la boucle.

Exemple : un bloc contenant une boucle :

```
{
    x = 3 ;
    y = 5 ;
    do {
        x = x-1 ;
        y = 2+y ;
    } while (x>0) ;
}
```

Il existe une troisième forme possible :

```
for( i1 ; test ; i2 )
    bloc ;
```

qui est une forme simplifié de :

```
i1 ;
while (test) {
    bloc ;
    i2 ;
} ;
```

Sémantique

Si *E* est l'environnement de départ, alors l'exécution de :

```
while (test)
    bloc ;
```

produit un environnement *E'* obtenu ainsi :

1. calculer *t* = évaluer(*test*, *F*), où *F* est l'environnement courant (au départ, *F*=*E*)
2. si *t* est faux, l'exécution de la boucle est terminée, et *E'* = *F*
3. si *t* est vrai, alors exécuter le *bloc* (lequel transforme *F* en *F'*, qui deviendra le nouvel *F*), puis revenir au point 1.

Pour la seconde forme de boucle, la sémantique est pratiquement identique, excepté que le test est effectué après exécution du bloc, plutôt qu'avant ; ainsi :

```
do
    bloc
while (test) ;
```

produit un environnement *E'* tel que :

1. exécuter le *bloc*, lequel transforme l'environnement courant *F* en *F'* (au départ, *F*=*E*)
2. calculer *t* = évaluer(*test*, *F'*)
3. si *t* est vrai, alors revenir au point 1.
4. si *t* est faux, l'exécution de la boucle est terminée, et *E'* = *F'*.

Exemple : que devient l'environnement (x, 10, int) • (y, 10, int) • ø après exécution de :

```
{
    x = 3 ;
    y = 5 ;
    do {
        x = x-1 ;
        y = 2+y ;
    while (x>0) ;
}
```

Le résultat est le suivant : après exécution du bloc, la variable x vaut 0 et y vaut 11 (la variable x est un compteur contenant le nombre de passages dans la boucle, diminué d'une unité à chaque passage).

Ce bloc aurait pu être écrit ainsi :

```
{
    y = 5 ;
    for ( x = 3 ; x>0 ; x = x-1)
        y = 2+y ;
}
```

La sémantique développée ci-dessus est celle d'une boucle dont le corps ne contient aucune des 2 instructions spéciales **`break`** ou **`continue`**. En effet, chacune de ces instructions permet de rompre le déroulement normal décrit par cette sémantique.

Cas d'un corps de boucle exécutant une instruction `break`

Si le corps de la boucle contient l'instruction **`break`**, l'exécution du corps se déroule normalement jusqu'à exécution du **`break`** ; cette exécution se traduit alors par un abandon du reste des instructions du corps de la boucle, et arrête immédiatement la boucle.

Exemple : lorsque la boucle suivante se termine, la variable `x` a pour valeur 5, alors que `y` vaut 6 :

```
{
    int x = 10, y = x ;
    while (x>0) {
        x = x-1 ;
        if (x==5) break ;
        y = y-1;
    }
}
```

En effet, dès que `x` prend la valeur 5 dans la boucle, le **`if`** lance l'exécution du **`break`**, ce qui a pour effet d'interrompre immédiatement la boucle.

Cas d'un corps de boucle exécutant une instruction `continue`

Si le corps de la boucle contient l'instruction **`continue`**, l'exécution du corps se déroule normalement jusqu'à exécution du **`continue`** ; cette exécution se traduit alors par un abandon du reste des instructions du corps de la boucle, suivi d'un retour à la phase du test d'arrêt de la boucle. Si le test est vrai, la boucle est relancée, sinon elle se termine.

Exemple : lorsque la boucle suivante se termine, la variable `x` a pour valeur 0, alors que `y` vaut 1 :

```
{
    int x = 10, y = x ;
    while (x>0) {
        x = x-1 ;
        if (x==5) continue ;
        y = y-1;
    }
}
```

En effet, dès que `x` prend la valeur 5 dans la boucle, le **`if`** lance l'exécution du **`continue`**, ce qui a pour effet de forcer la boucle à abandonner le reste du corps (ici enlever 1 à `y`), à tester immédiatement sa condition d'arrêt, et d'agir en conséquence (arrêt ou relance).

Boucles imbriquées et instruction **`goto`**

Puisqu'un corps de boucle peut être quelconque, il peut lui-même contenir une boucle ; les boucles sont alors dites **imbriquées**.

Exemple :

```
{
    int x = 10, y = x ;
    while (x>0) {          // boucle extérieure
        x = x-1 ;
        y = x ;
        while (y>0)        // boucle intérieure
            y = y-1 ;
    }
}
```

Si, dans le corps de la boucle intérieure, nous souhaitons sortir de la boucle extérieure, une instruction **`break`** ne suffira pas, puisqu'elle ne nous fera sortir que de la boucle intérieure (i.e. celle dont le corps contient ce **`break`**), et donc rester dans le corps de la boucle extérieure.

Exemple :

```
{
    int x = 10, y = x ;
    while (x>0) {          // boucle extérieure
        x = x-1 ;
        y = x ;
        while (y>0) {      // boucle intérieure
            y = y-1 ;
            if (x==3) break ;  // sortie de la boucle intérieure
        }
    }
}
```

Idem si nous avions souhaité continuer la boucle extérieure par un **`continue`** dans le corps de la boucle intérieure.

La seule solution offerte par C++ à ce problème est d'user de l'instruction spéciale **goto**, qui permet de poursuivre l'exécution à partir d'une instruction quelconque repérée par une **étiquette**.

Pour marquer une *instruction* avec une *étiquette*, il suffit d'écrire :

```
étiquette : instruction ;
```

L'instruction peut être vide. L'écriture d'une étiquette suit les mêmes règles que celles d'un nom de variable : elle commence par une lettre ou un blanc souligné, suivi de lettres, chiffres ou blancs soulignés.

L'instruction **goto** doit simplement spécifier à quelle étiquette sauter.

```
goto étiquette ;
```

Exemple : nous voulons interrompre la boucle extérieure dans le corps de la boucle intérieure :

```
{
    int x = 10, y = x ;
    while (x>0) {        // boucle extérieure
        x = x-1 ;
        y = x ;
        while (y>0) {    // boucle intérieure
            y = y-1 ;
            if (x==3) goto fin_exterieure;
        } ;
        y = x ;
    } ;
    fin_exterieure : ;
}
```

Noter que l'étiquette est située juste après la fermeture du corps de la boucle extérieure.

Exemple : nous voulons continuer la boucle extérieure dans le corps de la boucle intérieure :

```
{
    int x = 10, y = x ;
    while (x>0) {        // boucle extérieure
        x = x-1 ;
        y = x ;
        while (y>0) {    // boucle intérieure
            y = y-1;
            if (x==3) goto continue_exterieure;
        } ;
        y = x ;
        continue_exterieure : ;
    }
}
```

Noter que l'étiquette est située juste avant la fermeture du corps de la boucle extérieure.

Cette instruction spéciale permet de se passer des instructions de boucle ; en effet, l'exemple ci-dessus peut se récrire :

```
{
    int x = 10, y = x ;
    debut_ext : if (x>0) {        // boucle extérieure
        x = x-1 ;
        y = x ;
        debut_int : if (y>0) {   // boucle intérieure
            y = y-1;
            if (x==3) goto continue_ext;
            goto  debut_int ;
        };
        y = x ;
        continue_ext : goto  debut_ext ;
    }
}
```

Le problème est que le programme devient difficile à lire (les programmes abusant de **goto** sont appelés des **programmes spaghetti** : si l'on devait tracer un trait à chaque saut provoqué par un **goto**, le résultat ressemblerait à un véritable plat de…)

Autrement dit, à chaque fois qu'il est possible de se passer de l'instruction **goto**, ne surtout pas hésiter à s'en priver.

Il est à remarquer une particularité dans la sémantique de l'instruction **goto** : l'étiquette visée doit appartenir au bloc contenant le **goto** ou à un bloc l'englobant (i.e. un bloc contenant le bloc portant le **goto**).

Exemple : ce bloc est incorrect :

```
{
  là : int a = 1;
  {
    ici : int b = 2;
    goto là; // OK : l'étiquette est dans un bloc englobant
  };
  goto ici; // Erreur : l'étiquette n'est pas dans un bloc englobant
  goto là; // OK : l'étiquette est dans le même bloc que goto
}
```

Si l'étiquette appartient à un bloc englobant, l'exécution du **goto** provoque la fermeture de tous les blocs intermédiaires, i.e. tous ceux compris entre le bloc portant le **goto** et le bloc portant l'étiquette.

Exemple : exécution du bloc suivant dans un environnement initialement vide :

```
{           // début du bloc 1
  int a = 1;
  {         // début du bloc 2
    int b = 2;
    int c = 3;
    {       // début du bloc 3
      int d = 4;
      goto fin;
      a = b+c+d;
    };      // fin du bloc 3
    a = b+c;
  };        // fin du bloc 2
  a = 2*a;
  fin : ; // point ①
}           // fin du bloc 1
```

Au point ①, situé juste après l'étiquette, l'environnement courant ne contient que la variable (`a`, 1, **`int`**). En effet, l'exécution de l'instruction **`goto`** `fin` interrompt l'exécution normale du bloc 3. Le bloc 3 supportant l'instruction **`goto`** est englobé par le bloc 2, lui-même englobé par le bloc 1, lequel porte l'étiquette visée par le **`goto`**. Les blocs 3 et 2 sont donc fermés, dans cet ordre, l'exécution se poursuivant à partir de l'étiquette `fin`. Les trois instructions d'affectation sont donc ignorées.

5. TEXTE C++ ET PORTEE

Un programme écrit en C++ est une suite de définitions ou de déclarations rangées dans un fichier appelé **source**. Ce fichier source sera analysé par un programme appelé **compilateur** pour produire un fichier **exécutable**, lequel pourra être exécuté, et donc produire les effets attendus.

5.1. LES COMMENTAIRES

Si le programmeur souhaite écrire un texte qui n'est pas du C++ (appelé **commentaire**) dans un fichier source, il a deux possibilités :

- si les deux caractères `//` sont présents dans une ligne, alors ces 2 caractères, ainsi que tous les caractères situés à leur droite sur cette même ligne, seront ignorés par le compilateur.
- si les deux caractères `/*` sont présents dans une ligne, alors ces 2 caractères, ainsi que tous les caractères, voire lignes, qui suivent, sont ignorés jusqu'à la lecture des caractères `*/` (ignorés également) par le compilateur.

Attention : si ces couples de caractères sont présents dans un littéral texte (un texte encadré de guillemets), ils sont considérés que des caractères banals, donc ne délimitant pas une zone de commentaires.

Exemple :

```
{
    int a ;     // un commentaire d'une ligne
    cin >> a ;
    {
        while (a>0) {
            /* ici il y a un long
               commentaire, c'est à dire faisant au moins
               deux lignes */
            a = a-1 ;
            cout << "Attention // et /* ne sont pas des"
                 << " marqueurs de commentaires ici" ;
}   }   }
```

5.2. LA PORTEE

La **portée** d'un nom désigne la zone du fichier source pour laquelle ce nom possède une définition. Tout usage d'un nom en dehors de sa portée provoque une erreur du compilateur du genre : *nom indéfini*.

Si une définition est écrite dans un bloc, la portée de cette définition commence dès que cette définition se termine, et s'arrête à l'accolade fermante du bloc. La définition est alors dite **locale**.

Si une définition est écrite hors d'un bloc, sa portée commence immédiatement après la définition et s'arrête à la fin du fichier source la contenant. La définition est alors dite **globale**.

Exemple :

```
int a = b+1 ;          // erreur : b hors de portée (pas encore défini)
int b = 3 ;
{
    int i = a+1 ;
    {
        int j = b+i ;
    } ;
    int k = i+j ;      // erreur : j hors de portée
} ;
int c = a+k ;          // erreur : k hors de portée
```

En général, toute tentative de définir une entité C++ avec un nom déjà existant (cette définition survient donc dans la portée d'une autre définition exploitant déjà ce nom) provoque une erreur du compilateur du genre : *tentative de redéfinition*. C'est le cas par exemple lorsque, dans un même bloc, nous définissons deux fois une même variable.

Exemple : le texte C++ suivant contient une erreur de redéfinition :

```
{
    int i = 1 ;
    cout << i ;
    char i = '?' ; // erreur de redéfinition ici
    cout << i ;
} ;
```

Il existe toutefois des exceptions ; c'est le cas notamment des définitions de variables écrites dans des blocs imbriqués. Il est alors question de **masquage** de définitions et de **surcharge** de noms.

5.2.1. Surcharge et masquage

Nous avons évoqué, lors de notre étude des blocs d'instructions, une contrainte dans les blocs stipulant que toute nouvelle définition de variable doit nécessairement exploiter un nouveau nom.

Toutefois, rien n'interdit de définir plusieurs variables portant un même nom, à condition que ces définitions ne soient pas écrites dans un même bloc.

Exemple : le nom a sert à plusieurs définitions ici :

```
float a = -1.0 ;
{
    cout << a ;          // ici a est un float
    int a = 2 ;          // maintenant a est un int
    cout << a ;
    {
        char a = '?' ; // maintenant a est un char
        cout << a ;
    } ;                  // ici a est à nouveau un int
    cout << a ;
} ;                      // ici a est à nouveau un float
```

L'exécution de ce bloc produit à l'affichage :

```
-1.0  2  ?  2
```

L'exécution du bloc donné dans l'exemple ci-dessus montre que, lorsque nous sommes dans le bloc le plus profond (celui où la variable est un caractère), l'environnement courant contient bien trois variables distinctes. Chaque fermeture de bloc fait disparaître la définition introduite par le bloc, ce qui permet de retrouver une définition antérieure.

Dans l'exemple, le nom `a` est dit **surchargé** : à un moment donné, il existe plusieurs définitions usant de ce même nom. Par contre, une seule de ces définitions est active, les autres étant temporairement mises en sommeil (elles sont dites **masquées**). Nous dirons que cette définition **masque** les autres.

6. NOTION DE FONCTION

Le concept de fonction est fondamental en informatique : il en constitue l'un des principaux outils d'abstraction. Les fonctions du monde informatique ne sont pas en général de vraies fonctions au sens mathématique du terme, mais sont supposées en être une bonne concrétisation. Néanmoins, il serait facile de transformer ces fonctions informatiques en véritables objets mathématiques, au prix toutefois d'une certaine lourdeur d'écriture.

6.1. MOTIVATION

L'intérêt des fonctions est double :

- un programme, i.e. un bloc d'instructions, peut être découpé en divers sous-blocs indépendants (appelés fonctions dès lors qu'ils sont nommés et paramétrés), rendant ainsi sa lecture ou son analyse plus faciles.
- ces sous-blocs peuvent aussi être réutilisés plusieurs fois au sein d'un même programme, permettant ainsi en quelque sorte de factoriser le code. Par extension, ils pourront être réutilisés dans d'autres programmes, permettant ainsi de capitaliser l'écriture de code. Programmer devient alors une sorte de jeu de Lego, où il s'agit d'assembler (d'exploiter) certains blocs afin de résoudre un problème.

Exemple : dans le bloc suivant

```
{
    int x ;
    cout << "Donnez-moi une valeur entière : ";
    cin >> x ;
    cout << "Voici la valeur cherchée : "
    int a = 1, b = 2;
    int c = 3, d = 4;
    if (x>0) {
        if (a>b)
            cout << 2*a ;
        else
            cout << -b/20 ;
    } else {
        if (c>d)
            cout << 2*c ;
        else
            cout << -d/20 ;
    } ;
    cout << "\n";
}
```

il est facile de remarquer qu'un bloc apparaît par deux fois de façon quasi identique, au nom de 2 variables près :

```
{
    if (U>V)
        cout << 2*U ;
    else
        cout << -V/20 ;
}
```

Dans un premier cas, le bloc apparaît dans la partie *alors* du **if** (x>0) où U doit être remplacé par a et V par b. L'autre occurrence du bloc apparaît dans la partie *sinon* du **if**, où U est remplacé cette fois par c et V par d. Les variables U et V sont appelés les **paramètres** du bloc.

L'idée consiste simplement à donner un nom à ce bloc (appelons-le traitement), et d'indiquer explicitement quels en sont les paramètres (ici U et V, deux paramètres de type **int**). La réécriture du bloc s'écrira ainsi :

```
void traitement(int U, int V)
{
    if (U>V)
        cout << 2*U ;
    else
        cout << -V/20 ;
};

{
    int x ;
    cout << "Donnez-moi une valeur entière : ";
    cin >> x ;
    cout << "Voici la valeur cherchée : "
    int a = 1, b = 2;
    int c = 3, d = 4;
    if (x>0)
        traitement(a, b);
    else
        traitement(c, d);
    cout << "\n";
}
```

Noter l'écriture en deux parties : une première définit la fonction traitement, la seconde le bloc initial modifié. Dans ce dernier, faire référence au bloc nommé traitement (techniquement appelé une **fonction**) dans le code se limite à indiquer le nom de la fonction souhaitée, complété de la liste des valeurs à donner aux paramètres de la fonction lorsqu'elle sera exécutée.

La notion mathématique de fonction suggère que toute application d'une fonction produit un résultat. C'est bien entendu le cas des fonctions en C++ .

Exemple : la fonction suivante calcule la distance entre deux points (x_1, y_1) et (x_2, y_2) d'un plan :

```
float distance(float x1, float y1, float x2, float y2) {
    return sqrt(pow(x1-x2, 2)+pow(y1-y2, 2)) ;
    // sqrt(x) est la fonction calculant la racine carrée de x
    // pow(x, y) est la fonction élevant x à la puissance y
}
```

Cette fonction peut être exploitée ainsi :

```
{
    float a, b, c, d ;
    cout << "coordonnées x et y d'un premier point : ";
    cin >> a >> b ;
    cout << "coordonnées x et y d'un second point  : ";
    cin >> c >> d ;
    cout << "La distance entre ces 2 points vaut "
         << distance(a, b, c, d);
}
```

Exemple : dans l'exemple précédent de la fonction `traitement`, le résultat de la fonction est de type **`void`**, qui signifie… pas de résultat !

6.2. CONTEXTES D'USAGE DES FONCTIONS

La section précédente montre qu'une fonction n'est jamais qu'un bloc paramétré auquel est attribué un nom, et que l'exécution de ce bloc peut produire un résultat pouvant être exploité une fois cette valeur calculée.

Le nom d'une fonction dans un programme C++ peut apparaître dans trois contextes différents :

- une **déclaration** de fonction : il s'agit d'une simple indication au compilateur C++ lui permettant de savoir ce que signifie un nom.
- une **définition** de fonction : le compilateur C++ sait alors non seulement comment se nomme la fonction et quels sont ses paramètres, mais connaît désormais le bloc d'instructions qui lui est associé.
- un **appel** de fonction : le programmeur indique dans quel contexte il souhaite exploiter la fonction.

Exemple : dans l'exemple précédent, le nom de la fonction `traitement` apparaît une première fois pour définir la fonction, puis par deux fois pour l'appeler.

6.2.1. Déclaration d'une fonction

Il peut exister plusieurs déclarations d'une même fonction au sein d'un même programme C++. Deux fonctions sont identiques au sens d'une déclaration si elles possèdent la même **signature** ; une section ci-après précise ce qu'est la signature d'une fonction.

Syntaxe

La forme générale d'une déclaration de fonction est la suivante :

R *nom* ($param_1$, $param_2$, …, $param_n$, $param_{n+1}=e_1$, …, $param_{n+m}=e_m$) ;

où :

- $n \geq 0$ et $m \geq 0$, le **nombre de paramètres** de la fonction étant $n+m$. Ce nombre est aussi appelé **arité** de la fonction. Une fonction d'arité 1 est dite **unaire**, **binaire** quand son arité est 2, **ternaire** lorsqu'elle vaut 3.
- R est le nom d'un type, et représente le type du **résultat** de la fonction

- *nom* est le nom de la fonction, et suit les mêmes contraintes que les noms de variables
- $param_i$ est la spécification du $i^{ème}$ **paramètre** de la fonction ; il est de la forme :

  $$T_i\ n_i \quad \text{ou} \quad T_i$$

  où T_i est le type du paramètre (excepté **`void`**) et n_i son nom ; noter que ce nom peut être omis.
- e_i est une expression de calcul constante (i.e. ne contenant que des opérations prédéfinies de base de C++, en particulier aucun appel de fonctions). La valeur de cette expression est dite **valeur par défaut** du paramètre.

Exemple : la fonction calculant la distance entre deux points précédente se déclare par :

```
float distance(float, float, float, float) ;
```

Le type du résultat *R* peut être omis ; dans ce cas, le compilateur considère que ce type est **`int`**.

Dès lors qu'un paramètre possède une valeur par défaut, tous les paramètres figurant à sa droite doivent en posséder une à leur tour.

Sémantique

L'exécution d'une déclaration de fonction ne provoque aucun effet sur l'environnement d'exécution courant. Une déclaration ne sert que dans la phase d'analyse du programme (compilation), afin d'aider le compilateur à disposer d'un minimum d'informations sur la fonction.

Une déclaration de fonction peut se trouver dans un bloc ou hors d'un bloc. Les règles de portée sont celles déjà évoquées.

Il va sans dire que le compilateur vérifie que, pour une même fonction, toutes les déclarations qui s'y rapportent sont cohérentes quand elles précisent des valeurs par défaut.

6.2.2. Signature d'une fonction

La signature de la fonction est formée par les types caractérisant son résultat et ses paramètres, ainsi que son nom ; autrement dit, c'est sa déclaration privée des éventuels noms de paramètres et valeurs par défaut.

Exemple : la fonction suivante :

```
float fonct(float, float y, int z=0, float=-1.0) ;
```

a pour signature :

```
float fonct(float, float, int, float)
```

6.2.3. Définition d'une fonction

Dans un programme C++, il ne peut exister qu'une seule définition d'une fonction, à signature égale. Ce dernier point montre que plusieurs fonctions peuvent

porter un même nom à condition que leurs signatures diffèrent. Cette possibilité du langage C++ a déjà été évoquée : il s'agit de la **surcharge** des identificateurs, i.e. le droit de pouvoir exploiter un même nom pour désigner différents objets, leur contexte d'usage devant permettre de lever l'ambiguïté sur l'objet effectivement désigné.

Une définition de fonction n'est pas une instruction, i.e. elle ne peut pas apparaître dans un bloc. Toute définition de fonction doit donc nécessairement être écrite en dehors de tout bloc ; autrement dit, toute définition de fonction est globale.

La définition d'une fonction contient au moins tous les éléments de sa déclaration. Le seul élément ajouté est le bloc d'instructions associé à la fonction, appelé son **corps**. Toutefois, si au moins une déclaration de la fonction spécifiait des valeurs par défaut, alors ces valeurs doivent être omises dans sa définition, donc ne pas être reprises ; autrement dit :

- soit il existe une déclaration de la fonction indiquant des valeurs par défaut, et dans ce cas la définition ne doit pas repréciser ces valeurs
- soit aucune déclaration n'a précédé la définition, laquelle peut alors porter des valeurs par défaut.

Syntaxe

La forme générale d'une définition de fonction ressemble à celle d'une déclaration :

R nom (*param*$_1$**,** *param*$_2$**,** …**,** *param*$_n$**,** *param*$_{n+1}$=*e*$_1$**,** …**,** *param*$_{n+m}$=*e*$_m$)

corps **;**

où :

- $n \geq 0$ et $m \geq 0$, le **nombre de paramètres** de la fonction étant $n+m$
- *R* est le nom d'un type, et représente le type du **résultat** de la fonction
- *nom* est le nom de la fonction
- *param*$_i$ est la spécification du $i^{ème}$ **paramètre** de la fonction ; il est de la forme :

  $T_i\ n_i$

  où T_i est le type du paramètre (excepté **`void`**) et n_i son nom. Noter que le nom est obligatoire, cette fois (sauf cas très particuliers, sans intérêt ici).
- e_i est une expression simple de calcul constituant une valeur par défaut.
- *corps* est le bloc d'instructions associé à la fonction, qui décrit véritablement ce à quoi sert la fonction.

Toutes les contraintes étudiées pour la déclaration s'appliquent à une définition.

Le corps de la fonction est un bloc d'instructions vérifiant les propriétés suivantes :

- les variables ou constantes manipulables dans le corps, en dehors de celles qui y seront explicitement définies, sont les variables jouant le rôle de paramètres de la fonction, ainsi que toute entité (type, variable, fonction) dont la portée contient la définition.
- si le type du résultat de la fonction est **`void`**, le corps peut contenir plusieurs occurrences (y compris aucune) de l'instruction spéciale :

  `return ;`

- si le type *R* du résultat est autre que **`void`**, alors l'exécution du corps doit nécessairement se terminer par celle de l'instruction **`return`**, de la forme :

  `return` *exp* **`;`**

  où *exp* est une expression de calcul d'un type compatible avec *R*

Sémantique

Etant donné qu'une définition de fonction n'est pas une instruction, étudier la sémantique de l'exécution d'une telle définition n'a pas de sens.

La portée d'une définition de fonction est tout le reste du fichier source à partir de la fin de sa signature (dès la parenthèse fermante), puisqu'une telle définition est obligatoirement globale.

6.2.4. Appel d'une fonction

Dès lors qu'une fonction est définie ou déclarée, elle devient exploitable. Nous parlerons alors d'**appel** (ou d'**application**) de fonction.

Un appel de fonction est nécessairement écrit dans une expression de calcul, dont la forme syntaxique est étudiée ci-après. Sachant qu'il doit être possible d'écrire une instruction se contentant d'appeler une fonction (voir l'exemple de la fonction `traitement` ci-avant), rappelons qu'il est possible d'écrire une instruction se limitant à une expression de calcul seule (cf. section 4.2).

L'instruction d'affectation étudiée plus haut n'est qu'une forme particulière d'instruction se réduisant à un calcul d'expression : l'opération d'affectation est un calcul comme un autre, excepté qu'il modifie l'environnement courant, alors que la plupart des calculs laissent cet environnement inchangé.

Syntaxe

Si la fonction *nom* a été au préalable au moins déclarée par une ligne du genre :

R nom (*param*$_1$**,** *param*$_2$**,** …**,** *param*$_n$**,** *param*$_{n+1}$=*e*$_1$**,** …**,** *param*$_{n+m}$=*e*$_m$)

en reprenant exactement les mêmes notations que celles introduites pour la déclaration des fonctions, alors il existe m+1 formes possibles pour appeler la fonction *nom* dans une expression de calcul :

(0) $nom\,(exp_1, \ldots, exp_n)$

(1) $nom\,(exp_1, \ldots, exp_n, exp_{n+1})$

…

(m-1) $nom\,(exp_1, \ldots, exp_n, exp_{n+1}, \ldots, exp_{n+m-1})$

(m) $nom\,(exp_1, \ldots, exp_n, exp_{n+1}, \ldots, exp_{n+m-1}, exp_{n+m})$

Toutes les formes d'appel (0) à (m-1) sont équivalentes à une forme d'appel du type (m), c'est à dire dont tous les paramètres sont explicitement définis ; en effet, si $déf_i$ est la valeur par défaut du paramètre $n+i$ (c'est donc la valeur de l'expression e_i), alors :

(0) est équivalent à $nom\,(exp_1, \ldots, exp_n, déf_1, \ldots, déf_m)$

(1) est équivalent à $nom\,(exp_1, \ldots, exp_n, exp_{n+1}, déf_2, \ldots, déf_m)$

…

(m-1) est équivalent $nom\,(exp_1, \ldots, exp_n, exp_{n+1}, \ldots, exp_{n+m-1}, déf_m)$

Les paramètres manquants ont simplement été complétés avec les valeurs par défaut.

Sémantique

Soit *e* une expression de calcul figurant dans une instruction *i* et faisant un appel à une fonction *f* d'arité *n* (donc *n* paramètres). Nous supposons que la définition de cette fonction est du genre :

$$R\,f\,(T_1\,p_1,\ T_2\,p_2,\ \ldots,\ T_n\,p_n)\ corps$$

L'appel à *f* est supposé être complet (i.e. tous les paramètres sont donnés explicitement – cette hypothèse n'est pas restrictive, puisque nous avons noté dans la section précédente que tout appel incomplet est transformé en un appel complet en usant des valeurs par défaut pour les paramètres manquants), autrement dit de la forme :

$$f(e_1, e_2, \ldots, e_n)$$

où chaque e_i est une expression de calcul d'un type compatible avec T_i.

Exécuter l'instruction :

```
i ;
```

est équivalent à exécuter le bloc suivant :

```
{
    R résultat ;
    {
        T1 p1 = e1 ;
        T2 p2 = e2 ;
        …
        Tn pn = en ;
        corps' ;
    };
    fin : i' ;
}
```

où :

- dans ce bloc, tous les noms introduits doivent être nouveaux, i.e. les noms des variables *résultat*, p_1, …, p_n ne doivent pas exister dans l'environnement courant, quelque soit leur bloc d'appartenance, et le nom d'étiquette *fin* n'a jamais été exploité. Si nécessaire, créer de nouveaux noms, et les substituer aux originaux partout où ils doivent l'être (voir exemple ci-après).
- le *corps* spécifié lors de la définition de la fonction est récrit en *corps'*, sachant que les seules différences possibles sont :
  - soit au moins l'un des paramètres a été renommé, afin de respecter la contrainte de nouveauté des noms
  - soit il existe dans le corps original une instruction **`return`** de la forme :
    ```
    return ;
    ```
    et dans ce cas elle doit être substituée par :
    ```
    goto fin ;
    ```
  - soit il existe dans le corps original une instruction **`return`** de la forme :
    ```
    return exp ;
    ```
    et dans ce cas elle doit être substituée par le bloc :
    ```
    { résultat = exp ; goto fin ; }
    ```
- l'instruction *i* contenant l'appel est transformée en remplaçant l'appel par le nom de la variable *résultat*.

Exemple : supposons que nous devions exécuter le bloc suivant :

```
{
    float a, b, c, d ;
    cin >> a >> b >> c >> d ;
    if (a>c)
        cout << "valeur = " << monBloc(a+1, b-1) << "." ;
    else
        cout << "valeur = " << monBloc(2*c, d) << "." ;
}
```

avec la définition de fonction suivante :

```
float monBloc(float a, float b) {
    if (a!=0.0)
        return -b/a ;
    else
        return -1.0 ;
}
```

L'idée est de ne transformer l'instruction contenant l'appel de fonction que quand cela est nécessaire, i.e. lorsque nous sommes sûrs que cette instruction doit être exécutée. Pour l'exemple, l'exécution commence par l'ouverture d'un bloc, la définition de 4 variables puis la demande de leur valeur à l'usager. La conditionnelle **if** est alors atteinte ; posons pour l'exemple que l'usager ait saisi les valeurs suivantes :

variable	valeur saisie
a	2
b	5
c	0
d	-1

L'environnement courant avant **if** contient alors :

(d, -1, **float**) • (c, 0, **float**) • (b, 5, **float**) • (a, 2, **float**) • ✸ • …

Le test étant vérifié, c'est donc la partie *alors* du **if** qui doit être exécutée, c'est à dire :

```
cout << "valeur = " << monBloc(a+1, b-1) << "." ;
```

Or cette instruction contient un appel à la fonction `monBloc` ; son exécution revient donc à exécuter :

```
{
    float résultat ;
    {
        float p1 = a+1 ;
        float p2 = b-1 ;
        {
            if (p1!=0.0) {
                résultat = -p2/p1 ;
                goto fin ;
            } else {
                résultat = -1.0 ;
                goto fin ;
        }   };
    };
    fin : cout << "valeur = " << résultat << "." ;
}
```

Le bloc équivalent a été obtenu en suivant à la lettre les indications :

- la première variable définie est celle devant recevoir le résultat ; nous devons nous assurer que son nom n'existe pas dans l'environnement courant.
- les deux variables qui suivent sont les paramètres de la fonction ; il se trouve qu'il existe déjà dans l'environnement d'exécution deux variables appelées `a` et `b` (elles font partie des quatre variables déclarées dans le bloc initial). C'est pourquoi nous sommes obligés de les renommer, en l'occurrence ici `a` est renommée en p_1, et `b` en p_2. Ce changement est ensuite répercuté dans tout le corps de la fonction `monBloc`.
- toute instruction **return** est remplacée par l'équivalent spécifié ci-dessus.

- l'instruction originale `cout << …` est marquée par l'étiquette *`fin`*, et telle que l'appel à la fonction est substitué par le nom de la variable contenant le résultat.

  Ce programme affiche donc :

```
valeur = -1.333333
```

Attention : le schéma de substitution ne fonctionne pas dans le cas d'une définition de variable avec une initialisation comprenant un appel de fonction.

Exemple : soit la fonction suivante :

```
int fonc(int x) { return -x+3; }
```

et considérons le bloc suivant :

```
{
    int a = 3, b = fonc(a), c = a+b ;
    cout << a << ' ' << b << ' ' << c << endl;
}
```

L'instruction de définition doit tout d'abord être décomposée :

```
{
    int a = 3 ;
    int b = fonc(a) ;
    int c = a+b ;
    cout << a << ' ' << b << ' ' << c << endl;
}
```

La seconde instruction contenant un appel de fonction devrait être transformée en le bloc équivalent suivant :

```
…
int a = 3 ;
{
    int r_fonc ;
    {
        int x_fonc = a ;
        r_fonc = -x_fonc+3;
        goto fin_fonc ;
    }
    int b = r_fonc ;
} ;
int c = a+b ;
…
```

Le principe de substitution est alors mis en défaut : la définition de la variable `b`, qui était définie dans le bloc initial, devient une définition locale à un sous-bloc : dès la fermeture du bloc équivalent, la définition de `b` sera perdue…

Une première correction à apporter au principe de substitution consiste donc à signaler que, si l'instruction contenant l'appel à substituer est une instruction de définition avec initialisation, alors la décomposer en deux instructions : une instruction de définition sans initialisation suivie d'une instruction d'affectation :

original	après transformation
`T n = e ;`	`T n ;`
	`n = e ;`

Il reste toutefois quelques cas qui résistent à cette correction : la définition de variables d'un type référence (voir le chapitre 10) en est un. De façon générale, si se présente un bloc de la forme :

```
…
in-2 ;     // chaque ik est une instruction
in-1 ;
in ;       // ← instruction in contenant un appel, à substituer
in+1 ;
in+2 ;
…
```

alors une substitution correcte consiste en le bloc suivant (nous avons repris les notations données au début de l'exposé du principe de substitution – noter que cette version améliorée de la substitution pose encore un problème dans le cas d'un résultat de fonction qui serait d'un type référence) :

```
…
in-2 ;     // chaque ik est une instruction
in-1 ;
{          // ← bloc se substituant à l'instruction in
     R résultat ;
     {
          T1 p1 = e1 ;
          T2 p2 = e2 ;
          …
          Tn pn = en ;
          corps' ;
     };
     fin : in' ; // ← instruction in substituée
     in+1 ;
     in+2 ;
     …
}          // ← fin du bloc de substitution
```

Noter la différence avec le schéma de substitution initialement présenté : la place des instructions qui suivent celle à substituer, qui sont maintenant intégrées au bloc de substitution. Avec le premier schéma de substitution proposé, nous avions :

```
…
in-2 ;     // chaque ik est une instruction
in-1 ;
{          // ← début du bloc de substitution
     R résultat ;
     {
          T1 p1 = e1 ;
          T2 p2 = e2 ;
          …
          Tn pn = en ;
          corps' ;
     };
     fin : in' ;
};          // ← fin du bloc de substitution
in+1 ;
in+2 ;
…
```

6.3. RECURSIVITE ET ITERATION

6.3.1. Généralités

Une définition de fonction est dite **récursive** si cette définition contient au moins un appel à la fonction définie.

Exemple : la fonction `monBloc` de l'exemple précédent n'est pas récursive.

Exemple : la fonction suivante est récursive, car sa définition contient un appel à elle-même :

```
unsigned long factorielle(unsigned short n) {
     if (n==0)
          return 1 ;
     else
          return n * factorielle(n-1) ;
} ;
```

Dans une telle définition, il doit exister au moins un cas ne contenant pas d'appel récursif. Ce cas est appelé **clause d'arrêt** ou de **terminaison** de la récursion.

Exemple : dans l'exemple précédant, la clause d'arrêt correspond au cas `n=0`

L'une des principales difficultés dans la conception d'une définition récursive d'une fonction est de s'assurer que les appels récursifs convergent bien vers l'une des clauses d'arrêt. Les outils exploités s'appuient sur les techniques de preuve de convergence développées pour les suites mathématiques.

Toute définition de fonction non récursive et contenant au moins une boucle est dite **itérative**.

Exemple : la fonction suivante est itérative, et produit le même résultat que `factorielle` :

```
unsigned long factorielleb(unsigned short n) {
    unsigned long f = 1 ;
    while (n>0) { f = f * n ; n = n - 1 ; } ;
    return f ;
} ;
```

Il a été prouvé que :

> Théorème : toute définition récursive d'une fonction peut être transformée en une définition itérative, et inversement.

Malheureusement, il n'existe aucune procédure automatique opérant cette transformation dans le cas général (c'est à dire quelle que soit la définition récursive ou itérative de départ).

6.3.2. Récursivité terminale

Une définition de fonction est dite **récursive terminale** si et seulement si :

- elle est récursive
- toute valeur résultant d'un appel récursif à la fonction est aussi le résultat final de la fonction.

Exemple : la définition de la fonction `factorielle` donnée en exemple ci-avant n'est pas récursive terminale ; en effet, le résultat de l'appel récursif `factorielle(n-1)` n'est pas le résultat final, puisque cette valeur doit être multipliée par `n` pour le devenir :

```
return n * factorielle(n-1) ;
```

Exemple : la fonction `factorielle_rt` suivante est récursive terminale, car le seul appel récursif apparaissant dans sa définition est tel que le résultat de l'appel est le résultat final :

```
unsigned long factorielle_rt(unsigned short n,
                             unsigned long f) {
    if (n==0)
        return f ;
    else
        return factorielle_rt(n-1, f*n) ;
} ;
```

La première définition de la fonction `factorielle` se réduit alors à une définition non récursive, qui s'appuie sur une fonction récursive terminale :

```
unsigned long factorielle (unsigned short n) {
    return factorielle_rt(n, 1) ;
} ;
```

Noter qu'il était possible de définir directement `factorielle` sans passer par la fonction `factorielle_rt` intermédiaire, en exploitant simplement la notion de valeur par défaut :

```
unsigned long factorielle(unsigned short n,
                          unsigned long f = 1) {
    if (n==0)
        return f ;
    else
        return factorielle(n-1, f*n) ;
} ;
```

Il a été prouvé que :

> Théorème : toute définition récursive d'une fonction peut être transformée en une définition récursive terminale.

Malheureusement, il n'existe là encore aucune procédure automatique opérant cette transformation dans le cas général (c'est à dire quelque soit la définition récursive de départ). Néanmoins, une technique consiste à introduire des paramètres supplémentaires (en plus de ceux de la définition initiale) appelés **accumulateurs**, avec l'idée suivante :

> au lieu d'attendre la valeur d'un appel récursif (non terminal) pour ensuite la transformer puis retourner le résultat final, faire en sorte que ces paramètres supplémentaires portent toute l'information nécessaire pour que, lorsque la clause d'arrêt sera atteinte, cette information puisse permettre d'élaborer le résultat final.

Exemple : dans un premier temps, nous devons trouver une définition de la fonction `plus(a, b)`, où `a` et `b` sont des entiers naturels, calculant la somme de `a` et `b` en s'appuyant seulement sur les opérations +1 et –1.

Nous pouvons remarquer que :

plus(a, 0) = a
plus(a, b) = 1+plus(a, b-1) si b>0

La définition est bien récursive, et respecte les contraintes sur les opérations exploitées. Il est facile d'en déduire l'écriture C++ de la fonction demandée :

```
unsigned int plus(unsigned int a, unsigned int b) {
    if (b==0)
        return a ;
    else
        return 1+plus(a, b-1) ;
} ;
```

Cette définition récursive n'est pas terminale : le résultat de l'appel récursif n'est pas le résultat final, puisqu'il faut lui ajouter 1. La transformation en une version récursive terminale va consister à introduire une nouvelle fonction (appelons-la `plus_rt`), admettant un paramètre de plus que la version originale. Cette nouvelle fonction aura exactement la même structure que la première, au paramètre additionnel près. Notre fonction `plus` s'appuiera simplement sur cette nouvelle fonction (nous avons écrit des points de suspension pour le paramètre supplémentaire, afin de mieux mettre en évidence cette première étape de la transformation) :

```
unsigned int plus_rt(unsigned int a, unsigned int b, …) {
    if (b==0)
        return … ;
    else
        return plus_rt(a, b-1, …) ;
} ;

unsigned int plus(unsigned int a, unsigned int b) {
    return plus_rt(a, b, …) ;
} ;
```

Noter que l'appel récursif dans `plus_rt` est bien terminal. Le problème est de caractériser ce nouveau paramètre, i.e. comment il doit être géré. Dans cet exemple, il suffit de remarquer que, dans la version originale, le problème est d'ajouter 1 au fur et à mesure que `b` diminue. Si nous considérons qu'au départ nous disposons d'une variable `s` contenant `a`, chaque appel récursif devrait ajouter 1 à `s`, ce qui, lorsque la clause d'arrêt sera atteinte, nous permettra d'affirmer que le résultat final est `s`. D'où une première écriture :

```
unsigned int plus_rt(unsigned int a, unsigned int b,
                     unsigned int s) {
    if (b==0)
        return s ;
    else
        return plus_rt(a, b-1, s+1) ;
} ;

unsigned int plus(unsigned int a, unsigned int b) {
    return plus_rt(a, b, a) ;
} ;
```

Il est aisé de remarquer que le paramètre `a` dans `plus_rt` est devenu inutile, et donc de récrire tout le code ainsi :

```
unsigned int plus_rt(unsigned int a, unsigned int b) {
    if (b==0)
        return a ;
    else
        return plus_rt(a+1, b-1) ;
} ;

unsigned int plus(unsigned int a, unsigned int b) {
    return plus_rt(a, b) ;
} ;
```

Nous aurions pu découvrir plus vite cette définition si, au niveau mathématique, nous avions remarqué que :

plus(a, 0) = a
plus(a, b) = plus(a+1, b-1) si b>0

Noter que toutes les propriétés mathématiques ne sont pas nécessairement intéressantes d'un point de vue algorithmique ; par exemple, la définition suivante souffre d'un grave défaut algorithmique, alors qu'elle parfaitement valide mathématiquement : les appels récursifs ne convergent pas vers le cas d'arrêt :

plus(a, 0) = a
plus(a, b) = plus(a, b+1)-1 si b>0

Dans cet exemple, l'ajout d'un seul paramètre supplémentaire a suffi pour obtenir une version terminale de la récursion. Mais il est parfois nécessaire d'en ajouter plusieurs…

6.4. LES REDEFINITIONS D'OPERATEURS

Pratiquement tous les opérateurs C++ peuvent être surchargés, à condition qu'au moins l'un de leurs paramètres d'entrée soit d'un type non prédéfini (il s'agit donc d'un type construit, notion développée dans le chapitre qui suit).

Exemple : supposons que nous ayons défini un type nommé `Matrice` qui représente des matrices de *N* lignes à *N* colonnes, où *N* est une constante fixée arbitrairement. Mathématiquement, si *A* et *B* sont des matrices, il est possible de définir une matrice *C* comme étant la somme de *A* et *B*, ce qui se résume par *C*=*A*+*B*. En C++, une première manière de définir cette fonction consisterait à définir une fonction `plus` telle que :

```
Matrice plus(Matrice X, Matrice Y) { … } ;
```

et exploitée ainsi :

```
{
    Matrice A=…, B=…, C ;
    C = plus(A, B) ;
}
```

Il serait plus élégant d'écrire quelque chose comme :

```
{
    Matrice A=…, B=…, C ;
    C = A+B ;
}
```

C'est exactement ce que permet C++.

Si © est un opérateur C++ surchargé, sa définition s'écrit comme une définition de fonction, excepté que le nom de la fonction sera remplacé par **`operator`**©.

Exemple : avec notre somme de matrices précédente, la surcharge de l'opérateur binaire + s'écrira :

```
Matrice operator+(Matrice X, Matrice Y) { … } ;
```

et pourra être exploité comme annoncé :

```
{
    Matrice A=…, B=…, C ;
    C = A+B ;
}
```

Nous pourrions de même définir l'opérateur unaire (1 argument) :

```
Matrice operator-(Matrice X) { … } ;
```

permettant de calculer l'opposé d'une matrice, qui autoriserait alors une écriture du type :

```
C = A + -B ;
```

6.5. LA FONCTION SPECIALE MAIN

La fonction `main` (son nom est réservé) joue un rôle fondamental : définie par le programmeur (une seule fonction `main` par programme) , c'est elle, et elle seule, qui

sera appelée quand le programme sera exécuté ; libre au programmeur de la définir comme il se doit pour que le programme ait le comportement attendu.

Cette fonction possède plusieurs signatures :

`void main()`	Version de base, sans aucun paramètre ni résultat
`int main()`	La fonction retourne un entier qui sera interprété par le système d'exploitation comme le code d'erreur final du programme. Par convention, 0 signifie : pas d'erreur.

`int main(`
`  int c,`
`  char* t[]`
`)`
	La fonction prend en entrée un tableau `t` de `c` chaînes de caractères, qui sont les arguments passés à la ligne de commande ayant servi au système d'exploitation à lancer le programme. L'entier résultat s'interprète comme ci-dessus : code d'erreur final.

7. DE LA VALIDATION DU CODE PRODUIT

L'un des problèmes fondamentaux du génie logiciel est la validation de code : une fois rédigé le cahier des charges spécifiant la liste des problèmes que doit résoudre le futur programme, comment garantir ensuite que le programme en cours de conception, voire achevé, satisfera bien à toutes les contraintes qui étaient posées ? Nous donnons dans ce chapitre quelques éléments sur cette question, sachant que nous n'exploiterons qu'une infime partie de tous les outils ou techniques développés au fil des ans par le monde industriel ou académique de l'informatique.

7.1. LES TESTS DE PROGRAMME

Un programme C++ est constitué, comme les chapitres précédents le suggèrent, d'un ensemble de fonctions, lesquelles exploitent des valeurs se conformant à des règles induites par leur type. Ce découpage en fonctions, somme toute classique en programmation, permet de décomposer un programme en plusieurs modules, relativement indépendants, qu'il sera possible par la suite de réutiliser dans d'autres programmes.

7.1.1. Test de cas

La technique la plus empirique pour tester un programme est de l'essayer sur divers cas d'entrée. Si, pour chaque cas, la sortie produite correspond bien au résultat attendu, la confiance dans la correction du programme augmente, sans pour autant assurer qu'il fonctionnera dans tous les cas possibles.

Un premier problème consiste à choisir des cas suffisamment différents pour garantir que toutes les instructions composant le programme seront exécutées au moins une fois. En gros, chaque instruction conditionnelle nécessite deux cas différents, le premier obligeant le programme à considérer une alternative (test de la conditionnelle vrai), le second l'autre alternative (test de la condition faux). Il est aisé de montrer que le nombre de cas augmente de façon exponentielle avec le nombre d'instructions conditionnelles. Les cas idéalement sélectionnés sont appelés des **cas critiques** : chacun doit impliquer l'exécution d'une séquence d'instructions distincte des autres cas.

Il s'avère parfois impossible ou trop coûteux de recenser tous les cas critiques d'un programme ; un pis-aller consiste à ne sélectionner que ceux qui, statistiquement, ont le plus de chance d'être effectivement soumis un jour au programme. Le test par cas n'est alors plus exhaustif, mais couvre un échantillon de cas jugé représentatif ; le programme est jugé valide avec un risque d'erreur, parfois quantifiable.

7.1.2. Test unitaire et d'intégration

Tout programme étant un agencement de composants (les fonctions), il est possible d'adopter une stratégie de test plus fine que la précédente : au lieu de tester le programme comme un tout, testons en ses composants.

Le test d'un composant (une fonction, par exemple) isolé des autres est appelé **test unitaire**. Dès lors que tous les composants ont été testés individuellement, ils sont alors testés dans la construction particulière qu'est le programme qui les intègre. Le test est alors dit **test d'intégration**.

Il peut paraître surprenant, en première approximation, que des composants tous individuellement fiables (succès des tests unitaires) ne forment pas un tout fiable lorsqu'ils fonctionnent en synergie (échec du test d'intégration). L'un des problèmes mis en évidence dans ce type de test est celui des ressources partagées : deux composants peuvent, par exemple, s'appuyer sur une ressource informatique commune : un même fichier, une même ligne téléphonique, etc. Lorsque chacun fonctionne individuellement, il dispose de la ressource à lui tout seul, et aucun problème n'est constaté. L'intégration des composants peut conduire les composants à fonctionner en concurrence, la ressource ne permettant pas forcément d'être exploitée par deux composants à la fois (cas d'une ligne téléphonique, par exemple) : dès lors, un composant s'accapare la ressource, interdisant à l'autre, de fait, de fonctionner, puisque la ressource est exploitée de façon exclusive, par l'un ou par l'autre. Des tests unitaires passés avec succès ne garantissent donc pas le succès d'un test d'intégration.

7.1.3. Test de non régression

Lorsqu'un programme remplit le cahier des charges, un premier pas dans sa validité est atteint. Un programme n'est toutefois pas immuable : des parties sont améliorées, d'autres ajoutées, voire supprimées. Le problème est que le nouveau programme obtenu n'est peut-être plus capable de réaliser ce qu'il faisait avant ces modifications : les tests de **non régression** sont justement chargés de contrôler la pérennité des services rendus.

7.1.4. Les autres tests

Bien évidemment, un programme peut être étudié sous de nombreux critères : ses performances (par exemple sa capacité à répondre dans un délai donné), son respect de règles de sécurité, etc. Chaque critère peut conduire à des tests spécifiques.

7.2. INTEGRER DES POINTS DE CONTROLE

Les tests développés ci-dessus n'ont de sens qu'une fois le programme quasiment achevé, donc opérationnel. Or il peut s'avérer utile d'introduire dans un programme, dès sa conception, des points de contrôle qui permettront, si nécessaire, de vérifier son bon comportement. Comme les quelques exercices que nous avons résolus ensemble ont pu le montrer, la conception de fonctions récursives ou de blocs

intégrant des boucles n'est pas une tâche aisée : il n'est pas rare qu'un test de terminaison soit erroné, ce qui se traduit par un programme qui ne s'arrête plus. A supposer même que cette terminaison survienne, qu'est-ce qui peut nous garantir que les résultats, intermédiaires ou finaux, sont valides dans tous les cas de figure ?

Pour répondre à ces difficultés, la plupart des langages de programmation modernes permettent au programmeur d'affirmer que, en un certain point de son programme, une certaine propriété est soit toujours vraie, soit toujours fausse. Cette affirmation est appelée une **assertion**. La propriété qualifiée de toujours vraie ou fausse est appelée un **invariant**.

En pratique, toute assertion se traduit en général sous la forme d'une instruction spéciale. En phase de validation (test) du programme, toutes les assertions sont contrôlées. Si une assertion est violée, le programme s'arrête en expliquant exactement quelle assertion a été violée, à quel endroit dans le code, et donne accès à tout l'environnement d'exécution au moment de l'erreur ; le programmeur peut alors analyser le contexte dans lequel l'erreur s'est produite, et tenter ainsi de la corriger.

En phase de mise en production du programme (il a été testé et est désormais exploité par ses utilisateurs habituels), les assertions ne sont plus vérifiées : l'exécution des contrôles pourrait en dégrader les performances ; le programme peut alors rencontrer une erreur, mais aucune indication directe ne permettra alors de remonter à l'origine du problème.

7.2.1. La notion d'invariant

Dans ce cours, nous essaierons, tant que cela est possible, d'écrire des invariants permettant de vérifier que certaines propriétés sont toujours satisfaites, et d'en déduire d'autres propriétés. La recherche des invariants pertinents dans un programme est une tâche loin d'être triviale : les invariants ont toujours un lien avec les résultats attendus, mais définir exactement quels liens : tout dépend du problème traité, i.e. il n'y a pas de règle qui fonctionne à tous les coups. Ce sont les propriétés mathématiques qui caractérisent les objets manipulés par le programme qui doivent guider la découverte des bons invariants ; il s'agit donc d'une tâche éminemment abstraite…

Les invariants que nous étudions s'intègrent essentiellement au code des fonctions. Le langage C++ dispose d'une fonction, `assert`, laquelle est accessible en incluant le fichier d'en-tête `cassert`, qui admet en paramètre une expression *i* de type booléen. En phase de test, à chaque fois qu'une assertion est rencontrée, la valeur logique associée à l'expression *i* est calculée ; si cette valeur est vraie, le programme poursuit son exécution ; sinon, le programme s'interrompt en précisant l'assertion violée (quelle assertion, à quelle ligne de quel fichier source).

Dans ce cours, toutes les assertions seront toujours écrites sous forme de commentaires C++.

7.2.2. Pré et post conditions

Une **pré-condition** est une propriété qui doit toujours être vérifiée sur les paramètres d'entrée d'une fonction, juste avant son appel. Cette propriété permet par exemple de définir le domaine de définition de certains paramètres, ou toute autre propriété jugée nécessaire. Lorsqu'un appel à la fonction est demandé, la pré-condition est vérifiée une fois les paramètres liés aux valeurs qui leur correspondent, mais juste avant d'exécuter le corps de la fonction proprement dite.

Une **post-condition** est une propriété qui doit toujours être vérifiée sur des paramètres ou le résultat d'une fonction lorsque son appel est terminé. Cette propriété précise en général tout ce qui caractérise le résultat.

Exemple : prenons une fonction capable de calculer le quotient de la division entière de deux entiers naturels.

```
unsigned long quotient(unsigned long a, unsigned long b) {
    unsigned long q = 0 ;
    while (a>=b) { q = q+1 ; a = a-b ; } ;
    return q ;
} ;
```

Une pré-condition pour cette fonction est que `b` doit être non nul, soit :

```
// pré-condition : b>0
```

Une post-condition pourrait être : le résultat `quotient(a, b)` vérifie les propriétés du quotient :

```
// post-condition : 0 <= a-b*quotient(a, b) < b
```

7.2.3. Les invariants de boucle

Un **invariant de boucle** est une propriété associée à une boucle, et vérifiant :

- il est vrai juste avant d'exécuter la boucle
- s'il est vrai avant d'exécuter le corps de la boucle, il reste vrai une fois le corps de la boucle exécuté, quelque soit le nombre d'itérations.

La condition d'arrêt de la boucle ne doit avoir aucune incidence sur la véracité de cet invariant.

Exemple : la fonction suivante calcule la somme de deux entiers naturels :

```
unsigned long somme(unsigned long a, unsigned long b) {
    unsigned long s = a, y = b ;
    while (y>0) { s = s+1 ; y = y-1 ;} ;
    return s ;
} ;
```

Un invariant `I` pour cette boucle est :

```
(I) s == a + (b - y)
```

Vérifions qu'il s'agit bien d'un invariant :

1. juste avant d'entrer dans la boucle, les variables `s` et `y` sont initialisées respectivement à `a` et `b` ; en remplaçant ces variables dans l'invariant I, nous obtenons :

```
(I) a == a + (b - b)
```

soit :

```
(I) a == a
```

ce qui est toujours vrai.

2. supposons que `I` est vrai juste avant d'exécuter le corps de la boucle ; vérifions que sa véracité n'est pas modifiée par l'exécution de ce corps de boucle : après exécution, `s` a été remplacée par `s+1`, alors que `y` a été remplacée par `y-1`. Pour `I`, cela implique que :

```
(s+1) == a + (b - (y-1))
```

soit :

```
s+1 == a + (b - y)+1
```

soit :

```
s == a + (b - y)
```

Or cette dernière égalité est justement notre invariant `I`, supposé être vrai ; autrement dit, si `I` est vrai avant d'exécuter le corps de la boucle, il reste vrai une fois le corps de la boucle exécuté.

Le test de la boucle n'a aucune incidence sur `I` : il s'agit donc bien d'un invariant pour cette boucle.

L'intérêt d'un invariant de boucle est le suivant : si nous réussissons à montrer qu'une boucle finit par s'arrêter, alors nous pouvons affirmer que son invariant de boucle sera encore vrai juste après la fin de la boucle, puisque :

1. l'invariant est vrai juste avant d'entrer dans la boucle
2. à chaque fois que le corps de la boucle est exécuté, l'invariant reste vrai
3. le test d'arrêt n'a aucune incidence sur l'invariant.

Cet invariant peut alors être exploité pour démontrer ce à quoi sert effectivement la boucle.

Exemple : reprenons la fonction `somme` ci-dessus ; nous avons démontré que `I` est un invariant de la boucle.

Montrons tout d'abord que cette boucle se termine toujours : à l'entrée de la boucle, `y` est un entier naturel `b` quelconque. La boucle s'arrête si `y==0`. Quand `y` est non nul, le corps est exécuté au moins une fois, ce qui a pour effet de diminuer `y` de 1. Si nous étudions toutes les valeurs successives prises par `y`, nous constatons que la première valeur est `b`, puis `b-1`, `b-2`, etc. jusqu'à `b-b`, soit 0. Il s'agit donc d'une suite strictement décroissante, qui converge à coup sûr vers 0. La boucle se termine donc toujours, quel que soit `b`. La preuve de l'arrêt est donc acquise.

Lorsque la boucle se termine, nous sommes sûrs que la condition d'arrêt est vérifiée, soit pour notre boucle que `y==0`. A cet instant, `I` est toujours un invariant, i.e. nous sommes certains que :

```
s == a + (b - 0)
```

soit :

```
s == a + b
```

Autrement dit, à chaque fois que la boucle se termine, nous sommes assurés que `s` contient une valeur égale à la somme de `a` et `b`, quelles que soient les valeurs de `a` et `b`. Nous venons donc de démontrer que la fonction `somme` est correcte, i.e. qu'elle retourne bien toujours le bon résultat quand lui sont donnés deux entiers naturels à ajouter.

7.2.4. Les invariants d'état

Un **invariant d'état** est une propriété qui s'applique à un sous-ensemble de variables, et qui reste vrai quelles que soient les transformations que subissent ces variables. Ce type d'invariant est en général exploité avec des valeurs de types construits, en particulier les objets (des valeurs de types **`struct`**, **`union`** ou **`class`**, développés dans les chapitres qui suivent).

8. LES TYPES CONSTRUITS

Toute valeur manipulée par un programme est typée. Jusqu'ici, tous les exemples étudiés traitaient de valeurs dont les types sont élémentaires, en général entiers ou réels. Toutefois, un programme peut nécessiter d'exploiter des valeurs plus compliquées.

Un premier exemple consiste à examiner la notion mathématique de vecteur : un vecteur est défini par ses composantes, chaque composante étant numérotée (de 1 en 1, la première étant numérotée 1) et associée à une valeur réelle ; mathématiquement, un vecteur V est souvent décrit ainsi :

$$V = (v_1, v_2, \ldots, v_n)$$

où chaque v_i est la valeur de la composante i. Informatiquement parlant, il semblerait normal de pouvoir définir une variable de type vecteur, manipulable comme un tout, pour laquelle il serait possible d'accéder à la valeur de n'importe quelle composante, à condition d'en préciser l'indice. Un vecteur serait donc une juxtaposition de plusieurs valeurs réelles ; cette juxtaposition de valeurs ayant toutes un même type nous conduira à la notion de *tableau*, étudiée ci-après.

Un deuxième exemple est d'étudier un programme manipulant des informations relatives à des individus. Chaque personne serait définie par exemple par son nom, son prénom, sa taille, son âge, son sexe, … Nous aurions alors des valeurs de type *personne* à manipuler. Comme dans l'exemple précédent, nous remarquons qu'il s'agit d'une juxtaposition de valeurs plus élémentaires : le nom et le prénom sont des valeurs de type texte, l'âge et la taille sont des réels positifs, le sexe est au choix *masculin* ou *féminin*, etc. Toutefois, à la différence d'un vecteur, les valeurs juxtaposées ne sont pas toutes d'un même type. La notion de *tableau* évoquée ci-dessus sera alors remplacée par celle de *structure*.

8.1. RENOMMAGE DE TYPES EXISTANTS

Une première possibilité offerte par C++ pour introduire de nouveaux types est de nous permettre de nommer de façon différente des types déjà existants. Dans ce cas, il ne s'agit pas véritablement de nouveaux types, simplement d'introduire d'autres manières de nommer des types existants.

Sa syntaxe est la suivante :

```
typedef existant nouveau ;
```

où *existant* est une expression de type, et *nouveau* le nom donné à cette expression de type. Dès lors que cette définition est écrite, tout texte dans la portée de cette définition peut user du nom de type introduit.

Exemple : supposons que nous souhaitions donner des noms français à quelques-uns des types élémentaires que nous connaissons, puis que nous les exploitions :

```
typedef unsigned long unNaturel ;
typedef double unRéel ;
typedef char* unTexte ;

unNaturel multiplier(unNaturel a, unNaturel b) {
    return a*b;
};

void afficher(unTexte t) { cout << t; };

unRéel valeurAbsolue(unRéel x) {
    if (x<0) return -x ; else return x ; };
```

8.2. GENERALITES SUR LES CONSTRUCTIONS DE TYPES

Tout type bâti sur des types existants est dit *construit* ; c'est pourquoi nous employons l'expression de **construction de types**. Tout travail de programmation implique deux facettes :

- une définition des types manipulés (tâche de nature conceptuelle)
- une définition des fonctions (tâche de nature algorithmique) qui manipulent les valeurs de ces types

Il existe des constructions algébriques de types (notions de somme, de produit ou d'exponentielle de types) et des constructions ad hoc, par exemple les types pointeurs ou types références. Les constructions algébriques s'appuient sur une théorie des types, et se retrouvent dans tous les langages de programmation. Les constructions ad hoc, elles, ne sont présentes que dans certains langages.

Dans cette section, nous serons parfois amenés à considérer un type comme étant un ensemble de valeurs (au sens ensembliste classique). En général, deux types T_1 et T_2 de noms distincts sont supposés être disjoints d'un point de vue ensembliste, i.e. il n'existe aucune valeur ayant pour type T_1 et pour type T_2 : une valeur n'a qu'un seul type.

8.2.1. Produit, somme et exponentielle de types

Produit de types

Si T_1 et T_2 sont deux types quelconques, alors leur type produit est noté $T_1 \times T_2$. Toute valeur v de type $T_1 \times T_2$ est la juxtaposition d'une valeur v_1 de type T_1 et d'une valeur v_2 de type T_2 ; la notation classique consiste à écrire :

$$v = (v_1, v_2)$$

Si nous assimilons un type à un ensemble de valeurs, un produit de types est simplement leur produit cartésien, au sens ensembliste commun.

Si les types T_1 et T_2 sont identiques, soit $T_1=T_2=T$, C++ introduit une notion de *tableau* pour représenter un type qui est le produit d'un même type T. Si T_1 et T_2 diffèrent ou non, C++ introduit alors la notion de *structure* ou de *classe*. Bien entendu, une structure permet le produit de types identiques ; le choix *tableau* ou *structure* est alors laissé à la discrétion du programmeur.

Somme de types

La somme des types T_1 et T_2 est notée T_1+T_2. Dans ce cas, toute valeur v de type T_1+T_2 est égale à soit une valeur v_1 de type T_1, soit une valeur v_2 de type T_2.

Si nous assimilons un type à un ensemble de valeurs, une somme de types est simplement leur union, au sens ensembliste commun.

Si les types T_1 et T_2 sont des types n'ayant qu'une seule valeur, C++ introduit une notion d'*énumération* pour représenter leur somme T_1+T_2. Si T_1 et T_2 sont quelconques, C++ introduit alors la notion d'*union*. Le choix *énumération* ou *union* est laissé à la discrétion du programmeur si la question se présente.

Exponentielle de types

L'exponentielle du type T_2 par le type T_1 est notée $T_2^{T_1}$ ou encore $T_1 \rightarrow T_2$. Ce type est celui de toutes les fonctions de T_1 dans T_2, c'est à dire que toute valeur v de ce type est une fonction associant à toute valeur v_1 de type T_1 une valeur $v_2=v(v_1)$ de type T_2.

Cette construction de type a donc déjà été étudiée : il s'agit des fonctions.

Exemple : la fonction `multiplier` donnée dans l'exemple de la section précédente est de type :

$$\text{UnNaturel}^{\text{unNaturel} \times \text{unNaturel}}$$

c'est à dire une fonction qui associe un entier naturel à un couple d'entiers naturels.

8.2.2. Pointeurs et références

Ces deux constructions de types n'ont pas le caractère universel qu'ont les autres constructions (somme, produit, exponentielle) en théorie des types ; elles sont principalement motivées par des considérations techniques.

9. LES POINTEURS

9.1. MOTIVATIONS

La notion de pointeur en C++ est principalement motivée par leur aptitude à autoriser :

- la définition de types récursifs (étudiés plus loin)
- le passage de fonctions en paramètres d'autres fonctions

De par son héritage du langage C, il est possible en C++ d'exploiter les pointeurs pour permettre un passage de paramètres particulier, dit **passage par adresse**. Toutefois, ce mode de passage est désuet en C++, puisqu'une notion de référence a justement été introduite par les concepteurs du langage pour remplacer les pointeurs dans ce cas ; le passage de paramètres est alors dit **passage par référence**.

9.2. REDEFINITION DE LA NOTION D'ENVIRONNEMENT

La notion de pointeur nous oblige à redéfinir notre première notion d'environnement (cf. chapitre 3). Au début de notre étude, un environnement consistait une séquence de variables ou de marques de bloc. Chaque variable était représentée par un triplet (*nom*, *valeur*, *type*).

Désormais, un environnement est toujours constitué d'une séquence de variables ou de marques de bloc, mais nous lui associons aussi une **mémoire**. La mémoire de l'environnement est un ensemble de cases mémoires. Chaque case mémoire est repérée par un numéro, appelé **adresse** de la case, contient une valeur, laquelle a un type. Chaque variable de l'environnement est définie par un couple (*nom*, *adresse*), où *adresse* est l'adresse de la case mémoire servant à stocker la valeur associée à la variable.

Les opérations définies sur les environnements restent les mêmes, mais sont bien évidemment adaptées à la nouvelle structure. La sémantique de toutes les instructions étudiées précédemment ne change pas. Nous ferons l'hypothèse que toute variable effacée d'un environnement, qui consommait donc une case mémoire avant effacement, libère sa case mémoire, i.e. après effacement, la case mémoire perd sa valeur et le type associé, et peut être exploitée à nouveau par toute nouvelle variable à venir. Ce comportement de l'effacement ne sera toutefois pas valable pour les variables d'un type référence (cf. chapitre suivant).

Exemple : si, dans la première partie de notre étude, nous avions défini l'environnement E suivant :

$$E = (\texttt{x}, 7, \texttt{int}) \bullet \blacksquare \bullet (\texttt{y}, 4, \texttt{int}) \bullet (\texttt{i}, 1, \texttt{int}) \bullet \varnothing$$

ce même environnement sera désormais représenté ainsi :

E = (x, 2) • ✸ • (y, 1) • (i, 0) • ø

adresse ⇓	mémoire	type ⇓
0	1	**int**
1	4	**int**
2	7	**int**
…	…	…
…	…	…

La variable `i` est ainsi représentée dans l'environnement par le couple (`i`, 0), le 0 désignant l'adresse de la case mémoire stockant la valeur associée à `i`. L'examen de la case mémoire d'adresse 0 montre en effet qu'est rangée dans cette case la valeur 1, de type **`int`**.

De façon générale, si nous sommes incapables (ce sera presque toujours le cas) de connaître avec précision l'adresse de la case mémoire stockant la valeur d'une variable de nom `n`, nous noterons cette adresse `@n`.

Exemple : l'environnement E précédant peut se représenter ainsi si nous n'avons aucune information quand aux adresses effectivement exploitées par les variables :

E = (x, @x) • ✸ • (y, @y) • (i, @i) • ø

adresse ⇓	Mémoire	type ⇓
…	…	…
@i	1	**int**
…		…
@y	4	**int**
…		…
@x	7	**int**
…	…	…

Notons que la mémoire associée à un environnement, telle qu'elle est décrite ci-dessus, est une abstraction des véritables mémoires que l'on trouve dans les ordinateurs. En effet, nous faisons l'hypothèse que toute valeur, quel que soit son type prédéfini (entier, flottant, caractère ou booléen), peut être stockée dans une case mémoire et une seule, ce qui n'est pas le cas dans la réalité informatique ; rappelons que, par convention, une case mémoire réelle *mesurant* un **octet** ne peut stocker qu'une valeur parmi 256 possibles ; selon le type de la valeur à stocker, une ou plusieurs cases mémoire seront nécessaires pour coder des nombres n'appartenant pas à cet intervalle.

9.3. LE CONSTRUCTEUR DE TYPE POINTEUR

La valeur d'un pointeur est tout simplement une adresse, et occupe une seule case mémoire. En C++, si besoin, une adresse peut être convertie en un entier long. Attention : il existe une arithmétique élémentaire sur les pointeurs (opérateurs + et -), que nous étudierons dans la section sur les tableaux.

Si `T` est un type, alors `T*` désigne le type des « *pointeurs sur des valeurs de type* `T` », que nous résumons simplement par « *pointeurs sur* `T` ».

Exemple : nous voulons définir deux pointeurs sur des entiers :

```
int* p ; int* q ;
```

Attention : si nous souhaitons définir ces deux variables en une seule définition, nous serions tentés d'écrire :

```
int* p, q ;
```

Or ceci est équivalent à :

```
int* p ; int q ;
```

car le caractère * doit être attaché au nom de la variable, pas au type (héritage du langage C)… Nous devons donc écrire :

```
int *p, *q ;
```

Il est possible de nommer un type pointeur existant via une définition usant de **`typedef`** ; si *T* est un nom de type, alors :

```
typedef T* nom ;
```

définit *nom* comme le nom d'un type équivalent à l'expression de type *T**.

Exemple : cette série de définitions :

```
typedef int* unPointeurInt ;
int a = 1 ;
unPointeurInt p = &a ;
```

est strictement équivalente à :

```
int a = 1 ;
int* p = &a ;
```

ou encore :

```
int a = 1, *p = &a ;
```

9.4. OPERATIONS DE BASE SUR LES POINTEURS

Il existe deux opérateurs de base sur les pointeurs. D'autres opérations existent, mais sont liées à celles sur les tableaux ; elles seront donc développées dans la section consacrée aux tableaux.

9.4.1. Opérateur d'adresse

Si *n* est le nom d'une variable de type *T*, alors `&`*n* renvoie comme valeur l'adresse de la case mémoire stockant la valeur de *n* ; la valeur de `&`*n* est de type *T**.

Exemple : après exécution des deux instructions suivantes :

```
int v = 3 ;
int* p = &v ;
```

l'environnement est enrichi des deux variables suivantes :

E = (p, @p) • (v, @v) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@v	3	**int**
…		…
@p	@v	**int***
…	…	…

9.4.2. Opérateur d'indirection ou de déréférencement

Dans une expression de calcul, si *p* est une valeur pointeur de type *T**, i.e. une adresse, alors **p* retourne la valeur de type *T* stockée à l'adresse *p* (plus précisément, retourne une référence sur la valeur, i.e. une valeur de type *T*& - voir chapitre 10 sur les références). Par extension, si *n* est le nom d'une variable de type pointeur, disons *T**, alors **n* retourne la valeur de la case dont l'adresse est la valeur de *n*.

Exemple : après exécution des trois instructions suivantes :

```
int  n = 3 ;
int* p = &n ;
int  a = *p ;
```

l'environnement est enrichi des variables suivantes :

E = (a, @a) • (p, @p) • (n, @n)• …

adresse ⇓	mémoire	type ⇓
…	…	…
@n	3	**int**
@p	@n	**int***
@a	3	**int**
…	…	…

En effet, la valeur de p est @n, donc *(@n), dans la dernière instruction, retourne la valeur stockée à l'adresse @n, soit l'entier 3.

Notons que, si *p* est un pointeur sur un type *T* et *n* une variable de type *T*, alors les deux propriétés suivantes sont toujours vraies :

$$*(\&n) = *\&n = n \qquad \text{et} \qquad \&(*p) = \&*p = p$$

9.4.3. Conséquences sur l'opérateur d'affectation

La définition ci-dessus n'est correcte que si l'opération d'indirection débouche sur la lecture d'une valeur. En effet, si l'opération d'indirection se trouve en partie gauche d'une affectation, la valeur à affecter est alors stockée dans la case correspondant à l'adresse déréférencée.

Exemple : après exécution des deux instructions suivantes :

```
int v = 5 ;
int* p = &v ;
```

l'environnement est enrichi des deux variables suivantes :

E = (p, @p) • (v, @v) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@v	5	**int**
…		…
@p	@v	**int***
…	…	…

Si, partant de ce dernier environnement, nous exécutons l'affectation suivante :

```
*p = v-3 ;
```

alors l'environnement est modifié selon :

E = (p, @p) • (v, @v) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@v	2	**int**
…		…
@p	@v	**int***
…	…	…

En effet, l'énoncé ci-dessus nous indique que la valeur calculée à droite du signe = de l'affectation, soit 2, doit être stockée dans la case dont l'adresse (déréférencée) est la valeur de p, soit @v ; c'est donc bien la case @v qui est affectée, donc la variable v. Cette modification indirecte d'une variable est appelée un **effet de bord**.

9.5. FONCTIONS ET PASSAGE DE PARAMETRES PAR ADRESSE

L'un des principaux intérêts des pointeurs est qu'ils permettent d'étendre les capacités du langage quand au passage des paramètres des fonctions.

Le passage de paramètres étudié jusqu'ici est appelé **passage par valeur** : les seules variables accessibles à la fonction sont ses paramètres et les variables définies dans son corps. Avec cette nouvelle construction de type pointeur, nous enrichissons le type des paramètres qu'il est possible de définir : un paramètre de fonction peut être un pointeur.

9.5.1. Passage de fonctions en paramètre

Par définition, une variable pointeur, en particulier un paramètre pointeur, contient une valeur adresse. Notre premier cours sur la machine de Turing nous a conduit, après quelques transformations du modèle original, à remarquer qu'un programme était lui-même stocké en mémoire. Chacune des instructions qui le compose est codée comme une séquence de nombres (le codage dépend du processeur utilisé). En C++, un programme est un ensemble de fonctions. Chaque fonction est un bloc d'instructions, ces instructions étant codées en mémoire. Par convention, l'adresse d'une fonction est l'adresse de la case codant la première instruction de son corps (un bloc d'instructions).

Si *f* est une fonction déclarée par :

```
R f(…P…)
```

où *…P…* est la liste des types de ses paramètres, alors un pointeur *p* sur cette fonction sera déclaré ainsi :

```
R (*p)(…P…)
```

Les parenthèses autour de **p* sont impératives. Dès lors, il sera possible d'écrire :

```
p = &f ;
```

Si *p* est un pointeur sur une fonction *f*, l'appel à cette fonction via *p* sera écrit ainsi :

```
(*p)(…v…)
```

où *…v…* est la liste des valeurs à donner aux paramètres, les parenthèses autour de **p* étant là aussi impératives. Cet appel sera strictement équivalent à l'appel :

f(…*v*…)

Exemple : le programme suivant permet à la fonction `main` de tester deux fonctions en factorisant le code de test dans une fonction `tester` :

```
float fonc1(float x, float y) { return x*y ; } ;

float fonc2(float x, float y) { return x-y*2 ; } ;

void tester(float x,
            float y,
            float (*f)(float, float),
            char* n)
{
    cout << n << '(' << x << ',' << y << ") = "
         << (*f)(x, y) << "\n";
};

main() {
    tester(2, 3, &fonc1, "fonc1") ;
    tester(2, 3, &fonc2, "fonc2") ;
} ;
```

A l'exécution, ce programme affiche à l'écran :

```
fonc1(2, 3) = 6
fonc2(2, 3) = -4
```

9.5.2. Passage par adresse et effets de bord

Un paramètre de type pointeur permet de modifier des variables qui ne sont pas dans la portée de la fonction ; nous parlerons alors de **passage par adresse**. Le passage des paramètres pointeurs est un passage par valeur classique, mais le comportement de l'affectation lorsqu'à sa gauche se trouve un pointeur déréférencé montre qu'il est possible de modifier la valeur d'une variable sans donner son nom explicitement, en l'occurrence en se contentant simplement de l'adresse de la case qui stocke sa valeur. C'est pourquoi ce mode de passage de paramètre est dit passage par adresse.

Exemple : l'exécution de la fonction `main` suivante :

```
void f1(int a) { a = 1 ; } ;

void f2(int *a) { *a = 1 ; } ;

main() {
    int x=3 ;              // point [1]
    cout << "repère 1 : x = " << x << "\n" ;
    f1(x) ;                // point [2]
    cout << "repère 2 : x = " << x << "\n" ;
    f2(&x) ;               // point [3]
    cout << "repère 3 : x = " << x << "\n" ;
} ;
```

provoque l'affichage de :

```
repère 1 : x = 3
repère 2 : x = 3
repère 3 : x = 1
```

En effet, au point [1], l'environnement est défini ainsi :

E = (x, @x) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@x	3	**int**
…	…	…

D'où la première ligne affichée. Pour déterminer l'état de l'environnement au point [2], il faut examiner les effets de l'appel à `f1`. Rappelons que cet appel est strictement équivalent à exécuter le sous-bloc suivant :

```
{
    {
        int a = x ;
        {
            a = 1 ;
        }
    };
    fin_f1 : ;
}
```

Autrement dit, une variable `a` est bien définie dans l'environnement, laquelle est détruite dès lors que le bloc qui contient sa définition est fermé. L'affectation `a=1` porte donc sur une variable qui disparaîtra par la suite, et qui n'a aucune incidence sur la valeur de `x` ; d'où la seconde ligne affichée, identique à la première.

Examinons le second appel `f2(&x)` afin de définir le nouvel état de l'environnement. Le sous-bloc équivalent à l'appel est celui-ci :

```
{
    {
        int *a = &x ;        // point [4]
        {
            *a = 1 ;         // point [5]
        }
    };
    fin_f2 : ;
}
```

Au point [4], l'environnement est défini ainsi :

E = (a, @a) • ✸ • ✸ • (x, @x) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@x	3	**int**
…		…
@a	@x	**int***
…	…	…

Les deux marques de bloc introduites correspondent à l'ouverture des blocs avant la définition de `a`.

Au point [5], l'environnement devient :

E = ✸ • (a, @a) • ✸ • ✸ • (x, @x) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@x	1	**int**
…		…
@a	@x	**int***
…	…	…

En effet, l'affectation contient en partie gauche un pointeur déréférencé ; la valeur affectée sera donc stockée à l'adresse qui est la valeur de `a`, soit `@x`.

Au point [3], tous les sous-blocs équivalents à l'appel sont fermés, ce qui nous laisse :

E = (x, @x) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@x	1	**int**
…	…	…

D'où la dernière ligne affichée. La fonction `f2` a donc été capable de modifier la variable `x`, alors que celle-ci est hors de portée (seule variable significative dans la portée de `f2` : son paramètre `a`). Cette affectation indirecte (ou effet de bord) sur `x` a été rendue possible par le passage par adresse.

Exemple : un grand classique : la fonction `échanger` suivante est capable d'échanger les valeurs de deux variables entières :

```
void échanger(int* x, int* y) {
    int t = *x ; *x = *y ; *y = t ; }
```

L'exécution de ce bloc :

```
{
    int a = 2, b = 5 ;
    cout << a << ',' << b << "\n" ;
    échanger(&a, &b) ;
    cout << a << ',' << b << "\n" ;
}
```

provoque l'affiche de :

```
2 , 5
5 , 2
```

Les contenus des variables ont bien été échangés après l'appel, bien que ces variables `a` et `b` soient hors de portée de la fonction `échanger`.

10. LES REFERENCES

10.1. MOTIVATION

L'expérience montre que l'usage des pointeurs pose souvent des problèmes dans la mise au point des programmes, et est à l'origine de nombreuses erreurs pas toujours faciles à déceler, même pour les programmeurs chevronnés.

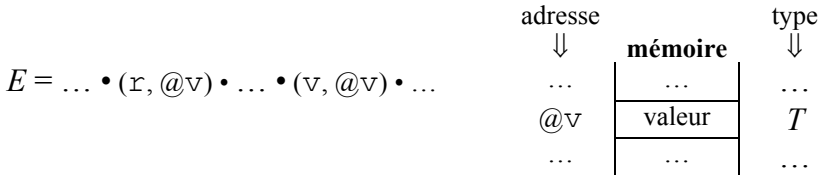
Les concepteurs de C++ ont donc introduit une nouvelle construction de type, dont l'objet est justement d'éliminer au maximum ces erreurs sur les pointeurs, tout en gardant l'esprit des pointeurs (i.e. leur capacité à se référer à une variable via l'adresse de la case mémoire associée).

10.2. LE CONSTRUCTEUR DE TYPE REFERENCE

Si *v* est une variable de type *T*, la définition d'une variable référence *r* initialisée avec *v* sera écrite :

```
T& r = v ;
```

La partie initialisation est obligatoire. Le résultat d'une telle définition sur l'environnement est la création d'une nouvelle variable de type *T*`&` qui partage la même case mémoire que la variable ayant servi à l'initialiser, soit :



Ainsi, lire ou modifier la valeur de *r*, c'est en fait lire ou modifier la valeur de *v*. Nous dirons que la définition de *r* l'a liée à la variable *v* ; autrement dit, *r* est en quelque sorte un autre nom pour la variable *v*. Cette propriété montre qu'une référence de type *T*`&` se manipulera comme n'importe quelle variable de type *T*, puisqu'elle n'est qu'une dénomination différente d'une variable déjà existante de type *T*.

Exemple : le bloc suivant :

```
{
    int a = 2;
    int &r = a;
    cout << a << ',' << r << "\n" ;
    r = 1 ;
    cout << a << ',' << r << "\n" ;
    a = 3 ;
    cout << a << ',' << r << "\n" ;
}
```

provoque l'affichage de :

```
2 , 2
1 , 1
3 , 3
```

Une variable de type référence partage son espace mémoire avec une variable déjà existante ; l'espace mémoire de la variable référence ne lui appartient donc pas, i.e. lorsque cette variable disparaîtra (fermeture de son bloc de définition), son espace mémoire ne sera pas libéré (restitué), puisqu'une autre variable l'exploite encore…

10.3. FONCTIONS ET PASSAGE DE PARAMETRES PAR REFERENCE

Le principal intérêt de cette construction de type est de permettre un passage de paramètre dont le comportement ressemble fortement au passage par adresse, i.e. sa capacité à provoquer des effets de bord, en évitant la lourdeur liée à l'usage explicite de pointeurs. Ce nouveau mode de passage est appelé **passage par référence**.

La déclaration d'un paramètre référence ne pose pas de difficulté particulière ; elle s'écrira de la forme :

```
T& r
```

L'initialisation du paramètre référence ne sera effectuée que lors d'un appel à la fonction. La liaison entre la référence et la variable liée n'a de sens qu'au moment d'un appel. La technique de substitution de l'une instruction contenant un tel appel est toujours valable.

Exemple : reprenons la fonction `échanger` décrite dans la section sur les pointeurs, capable d'échanger les valeurs de deux variables entières :

```
void échanger(int& x, int& y) {
    int t = x ; x = y ; y = t ; }
```

L'exécution de ce bloc :

```
{
    int a = 2, b = 5 ;
    cout << a << ',' << b << "\n" ;
    échanger(a, b) ;
    cout << a << ',' << b << "\n" ;
}
```

provoque l'affiche de :

```
2 , 5
5 , 2
```

Le contenu des variables a bien été échangé après l'appel, comme dans la première version présentée exploitant des pointeurs. Noter toutefois que l'écriture a gagné en simplicité : les erreurs d'omission des déréférencements ou des opérateurs d'adresse sont éliminées.

10.4. POINTEURS OU REFERENCES

La différence fondamentale entre un pointeur et une référence est que nous avons tout le loisir de changer la variable pointée par un pointeur, alors qu'une référence est liée à vie à la variable qui a servi à l'initialiser. Le choix pointeur ou référence est donc fortement conditionné par cette pérennité de la liaison :

- si le paramètre doit être lié toute sa vie à la variable ayant servi à l'initialiser, alors définir ce paramètre comme une référence ; un pointeur pourrait convenir, au prix d'une lourdeur d'écriture accrue.

- si au contraire le paramètre peut être lié durant sa vie à différentes variables, alors définir ce paramètre comme un pointeur ; une référence serait tout à fait inadaptée, puisque sa liaison est immuable.

Exemple : l'exécution du bloc suivant :

```
{
    int a = 2, b = 5 ;
    cout << a << ',' << b << "\n" ;

    int* p = &a ; *p = 1 ;
    cout << a << ',' << b << "\n" ;
    int& r = a ;   r = 3 ;
    cout << a << ',' << b << "\n" ;

    p = &b ; *p = 4 ;
    cout << a << ',' << b << "\n" ;
    r = b ;   r = 6 ;
    cout << a << ',' << b << "\n" ;
}
```

affiche à l'écran :

```
2 , 5
1 , 5
3 , 5
3 , 4
6 , 4
```

Cette dernière ligne montre que l'affectation `r = b` ne change pas la variable liée à `r`. En effet, `r` sera toujours liée à `a`, tant que sa définition sera active. Autrement dit, l'affectation `r = b` est strictement équivalente à `a = b`. Un affichage juste après cette affectation montrerait cette pérennité de la liaison, puisque serait affiché :

```
4 , 4
```

C'est bien entendu le même argument qui, après l'affectation `r = 6`, explique que cette affectation agit sur `a`, et donc l'affichage indiqué.

11. LES PRODUITS DE TYPES

Si T_1, T_2, …, T_n sont n types quelconques, leur produit est noté $P = T_1 \times T_2 \times \ldots \times T_n$. Toute valeur v de type P est un n-uplet, c'est un dire une juxtaposition de n valeurs, chaque valeur v_i étant de type T_i, soit :

$$v = (v_1, v_2, \ldots, v_n)$$

Chaque v_i est la valeur d'une composante de v. En théorie des types, une **projection** π_i (appelée encore **sélecteur** ou **observateur**) est une fonction qui, appliquée à une valeur v d'un type produit P, retourne la valeur de sa $i^{ème}$ composante, soit :

$$\pi_i : P \rightarrow T_i \text{ , avec } \pi_i(v) = v_i$$

11.1. LES TABLEAUX

Un type **tableau** est un produit de types tous identiques. Le nombre de types impliqués dans le produit est appelé **taille** du tableau. L'accès aux composantes (projection) s'effectue via un système d'indices entiers.

Exemple : un tableau de 10 entiers a pour type : produit de 10 types entiers

11.1.1. Définition d'un tableau

La définition d'une variable *nom* de type tableau constitué de n composantes de type T est notée ainsi en C++ :

```
T nom[n] ;
```

ou

```
T nom[n] = { e1, e2, …, em } ;
```

ou

```
T nom[] = { e1, e2, …, en } ;
```

Chaque composante d'un tableau est repérée par un **indice**, nombre entier compris entre 0 et n-1. La première composante a pour indice 0, la suivante 1, etc., la dernière n-1.

La première forme définit le tableau sans préciser comment seront initialisées les composantes ; en général, les composantes sont initialisées avec la valeur par défaut du type T. Le terme n est une expression constante, i.e. ne doit contenir aucun appel à une fonction, quelle qu'elle soit.

La seconde forme définit comment sont initialisées les m premières composantes ($m \leq n$) ; chaque composante reçoit la valeur de l'expression e_i correspondante : la première composante reçoit la valeur de e_1, la seconde celle de e_2, etc.. Si $m=n$, alors toutes les composantes sont explicitement initialisées. Sinon, les n-m composantes non initialisées explicitement reçoivent la valeur par défaut du type T.

La dernière forme montre qu'il est possible d'omettre la taille d'un tableau dont les composantes sont initialisées ; la taille du tableau est alors déduite : elle égale le nombre d'expressions d'initialisation.

Exemple : la variable `t1` est un tableau de 10 composantes de type **`int`**, non initialisées :

```
int t1[10] ;
```

La variable `t2` est un tableau de 5 entiers, dont seules les quatre premières composantes sont explicitement initialisées avec les valeurs respectives 1, 7, 3 et 2. La cinquième composante n'est pas initialisée :

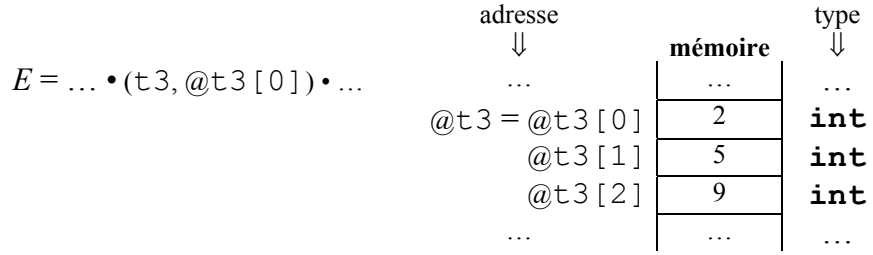
```
int a = 3 ;
int t2[5] = { 1, 2+5, 7-4, a-1 } ;
```

La variable `t3` est un tableau de 3 entiers, dont les composantes sont initialisées explicitement avec les valeurs respectives 2, 5 et 6. La taille du tableau est déduite du nombre de composantes initialisées :

```
int t3[] = { a-1, a+2, 3*a } ;
```

Du point de vue de l'environnement, les composantes d'un tableau sont toutes rangées dans des cases mémoire adjacentes. L'adresse associée à un tableau est celle de sa première composante.

Exemple : le tableau `t3` de l'exemple précédent serait ainsi défini dans l'environnement :



11.1.2. Opérateur d'accès

Si *v* est un tableau de *n* composantes, alors *v*`[`*i*`]` désigne sa composante d'indice *i*. Cette composante est aussi bien accessible en lecture qu'en écriture (elle figure alors à gauche du signe = dans une affectation).

Exemple : le bloc suivant définit un tableau dont les 4 premières composantes sont initialisées explicitement. La cinquième composante (d'indice 4) est modifiée via une affectation (accès en écriture). Une boucle affiche les valeurs de toutes les composantes (accès en lecture) :

```
{   int i = 3 ;
    const int n = 5 ;
    int tab[n] = { i-1, i+4, 2*i, 3 } ;
    tab[4] = -1 ;
    for(i=0 ; i<n; i++)
        cout << "tab[" << i << "] = "
             << tab[i] << "\n" ;
} ;
```

Cet opérateur d'accès `[]` est la concrétisation pour les tableaux de la notion de projection évoquée dans le préambule de cette section sur les produits de types.

11.1.3. Opérations prédéfinies

En dehors de l'opérateur d'accès à une composante, aucune autre opération n'est définie sur les tableaux : pas d'affectation entre tableaux, pas d'initialisation d'un tableau à partir d'un autre tableau, aucun opérateur de comparaison (==, etc.).

11.1.4. Tableaux et pointeurs

Passage de tableaux en paramètres de fonctions

Il est possible de passer des tableaux en paramètres de fonctions. Si *p* est un paramètre d'une fonction devant être associé à un tableau dont les composantes sont d'un type *T*, la déclaration du paramètre s'écrira :

```
T p[]
```

L'omission de la taille est obligatoire (nous ferons en tout cas comme si). Cela signifie que si une fonction doit pouvoir disposer de la taille d'un tableau passé en paramètre, il est nécessaire d'introduire un paramètre supplémentaire afin de recevoir cette taille.

Exemple : la fonction suivante affiche toutes les composantes d'un tableau d'entiers :

```
void afficher(int tab[], int taille, char* nom) {
    for(int i=0 ; i<taille ; i++)
        cout << nom << '[' << i << "] = "
             << tab[i] << "\n" ;
} ;
```

Cette fonction pourrait être exploitée ainsi :

```
{
    int t1[10] = { 5, 4, 3, 2 } ;
    int t2[30] = { 1, 2, 3, 4, t1[1], 4, 3, 2, 1} ;
    afficher(t1, 10, "t1") ;
    afficher(t2, 30, "t2") ;
}
```

Attention : le passage des tableaux en paramètre s'effectue par adresse. Autrement dit, toute modification opérée sur une composante du tableau dans la fonction modifie la composante du tableau passé à l'appel. Cette propriété est due à l'équivalence tableau/pointeur décrite ci-après.

Exemple : l'exécution du programme suivant :

```
void changer(int tab[]) {  tab[1] = -1 ; } ;

main() {
    int t[4] = { 4, 3, 2, 1 } ;
    changer(t) ;
    for(int i=0 ; i<4 ; i++) cout << t[i] << ' ';
} ;
```

provoque l'affichage de :

```
4 -1 2 1
```

Autrement dit, le tableau défini dans `main`, hors de portée de la fonction `changer`, a bien été modifié par l'appel à `changer`. Le passage par adresse explique ce comportement ; a

priori, nous aurions pu penser, à la lecture de la signature de la fonction, que le tableau était simplement passé par valeur ; ce n'est pas le cas…

Equivalence pointeur/tableau

Si *p* est un pointeur sur un type *T* (donc *p* est de type *T**) et *t* un tableau de *n* composantes de type *T*, alors il est possible d'initialiser ou d'affecter *p* avec *t*. L'adresse rangée dans *p* sera l'adresse de la première composante du tableau.

Exemple : le code qui suit est parfaitement licite :

```
void f(int* x) { *x = -1 ; } ;

main (){
    int tab[5] ;
    int *p = tab ;
    int *q = & tab[0] ; // p == q

    f(tab) ;      // la composante tab[0] est modifiée
    f(&tab[0]) ; // strictement équivalent à l'appel précédent
    f(&tab[3]) ; // cette fois tab[3] est modifiée
}
```

Mieux, le déréférencement **p* du pointeur *p* peut s'écrire *p*`[0]` ! En effet, d'une façon générale, si *p* est initialisé ou affecté avec l'adresse de la composante *j* du tableau *t*, donc *p*=`&`*t*`[`*j*`]`, alors *p*+*i* est un pointeur désignant l'adresse de la case mémoire stockant la composante d'indice *j*+*i*, soit :

```
p+i == & t[j+i]
```

et donc :

```
*(p+i) == t[j+i]
```

Cette propriété permet de définir une arithmétique élémentaire sur les pointeurs. Le paragraphe précédent indique que l'on peut ajouter un entier *i* à un pointeur *p*, le résultat étant un nouveau pointeur *p*+*i*.

Il est possible de soustraire un entier *i* à un pointeur *p*. Le résultat est aussi un pointeur, noté *p*−*i*, défini ainsi : si *p* est initialisé ou affecté avec l'adresse de la composante *j* du tableau *t*, donc *p*=`&`*t*`[`*j*`]`, alors :

```
p-i == & t[j-i]
```

et donc :

```
*(p-i) == t[j-i]
```

Si *p* et *q* sont deux pointeurs sur un même type *T*, alors *p*-*q* retourne un entier *i* tel que :

```
p == q+i
```

Exemple : quelques opérations arithmétiques sur les pointeurs :

```
{
    int t[10] ;
    int *p = t ; // autrement dit p = & t[0]
    int *q = & t[5] ;

    p+5 == q ;    // vrai car p+5 = & t[0+5] = q
    q-1 == p+4 ; // vrai car q-1 = & t[5-1] = & t[0+4] = p
    q-p == 5 ;    // vrai
    p-q == -5 ;   // vrai
}
```

Exemple : exploitation de l'arithmétique sur les pointeurs pour appliquer sur plusieurs portions d'un tableau une même modification :

```
void ecrire1sur3composantes(int t[]) {
    t[0] = 1 ; t[1] = 1 ; t[2] = 1 ; }
```

Le bloc qui suit :

```
{
    int tab[20] ;
    ecrire1sur3composantes(tab);       // 0 à 2
    ecrire1sur3composantes(&tab[5]);   // 5 à 7
    ecrire1sur3composantes(&tab[9]);   // 9 à 11
}
```

force à 1 les composantes 0 à 2, 5 à 7 et 9 à 11.

Mélange des constructions de types

Une petite difficulté syntaxique apparaît dès que plusieurs opérateurs sont présents au sein d'une même expression. Concernant les opérateurs sur les pointeurs, les fonctions et les tableaux, les règles sont les suivantes :

Opérateur d'accès ou d'application : `[]` `()` priorité maximale, toujours situés à droite du tableau ou de la fonction, applicables de gauche à droite

Opérateurs de base sur les pointeurs : `*` `&` priorité moindre, toujours situés à gauche du pointeur, applicables de droite à gauche.

Exemple : `*&t[1][4]` est évalué dans cet ordre (les parenthèses sont automatiquement ajoutées par le compilateur lors de son analyse) :

```
*(&((t[1])[4]))
```

Les règles de priorité nous invitent à commencer par les opérateurs situés à droite de `t`, en l'occurrence l'opérateur d'accès `[1]`, à poursuivre avec `[4]` (gauche à droite), puis de passer aux opérateurs situés à gauche, en les appliquant de droite à gauche ; d'où le parenthésage.

Exemple : `&(*f)(1,2)[3]` est évalué dans cet ordre (les parenthèses sont automatiquement ajoutées par le compilateur lors de son analyse) :

```
&(((*f)(1,2))[3])
```

Les règles nous invitent à commencer par les opérateurs situés à droite de `f`, mais la parenthèse fermante nous signale qu'il n'y en a pas. En poursuivant à gauche, nous

rencontrons `*`, puis la parenthèse ouvrante. Ce niveau de parenthèse étant analysé, nous continuons avec le niveau englobant ; nous trouvons d'abord une application de fonction `(1,2)`, puis un accès à un tableau `[3]`. N'ayant plus rien à droite, nous repartons à gauche, pour y trouver `&`.

S'agissant des expressions de type, les constructeurs de type appliquent les mêmes règles.

Ces constructions combinées permettent de construire des tableaux de tableaux, donc des matrices :

Exemple : un type désignant des matrices $L \times C$, où L=3 et C = 4 :

```
const int L=3, C=4 ;
typedef float uneMatrice[L][C] ;
uneMatrice M ;
M[2][3] = 1 ; // la composante en ligne 2 colonne 3 de la matrice vaut 1
```

Exemple : un petit condensé des horreurs qu'il est possible de rencontrer :

`int* f(float)`	`f` une fonction qui à un **`float`** associe un **`int*`**
`int (*p)(float, int*)`	`p` un pointeur sur une fonction qui, à un couple (**`float`**, **`int*`**), associe un **`int`**
`float (*(*t[10])(int))(char)`	`t` un tableau de 10 pointeurs sur des fonctions qui à un **`int`** associent un pointeur sur une fonction qui à un **`char`** associe un **`float`**

11.1.5. Les chaînes de caractères

Nous avons présenté dans la section sur les types de base la notion de chaînes de caractères, jusqu'ici signalée comme étant du type **`char*`**.

En fait, une chaîne de n caractères est un tableau de n+1 caractères, où chaque caractère de la chaîne occupe une composante du tableau, la dernière composante (d'indice n) contenant le caractère spécial de code 0.

Exemple : les deux définitions suivantes sont équivalentes quand au contenu des textes :

```
char* bonjour1 = "bonjour" ;
char bonjour2[] = { 'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0' } ;
```

Dans chacun des cas, l'accès au troisième caractère s'écrit simplement :

`bonjour1[2]` ou `bonjour2[2]`

11.1.6. Quelques défauts et pièges des tableaux

Le mariage entre fonctions et tableaux n'est pas excellent :

- une première limitation des tableaux, déjà signalée dans la section 11.1.3, est qu'un tableau passé en paramètre d'une fonction l'est toujours par adresse, jamais par valeur.
- une seconde limitation, signalée dans la même section, est relative à la taille d'un tableau passé en paramètre : la taille d'un paramètre tableau n'est, en principe, jamais écrite dans la signature de la fonction. Paradoxalement, la

spécification de taille devient obligatoire à partir de la seconde dimension, si le tableau est à plusieurs dimensions…

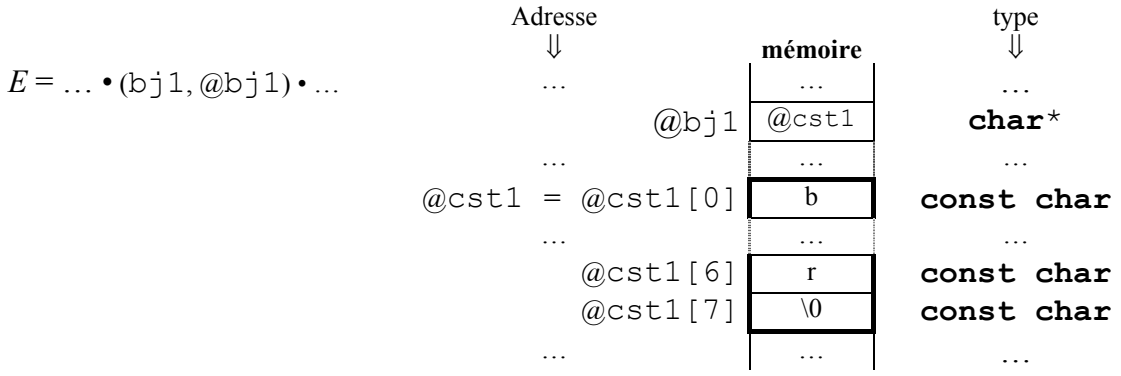
- une troisième limitation : une fonction ne peut pas retourner un résultat de type tableau.

Un autre piège attend le programmeur s'il exploite trop rapidement les équivalences entre tableaux et pointeurs : il s'agit de l'allocation mémoire ; nous pourrions croire que les trois définitions suivantes sont équivalentes :

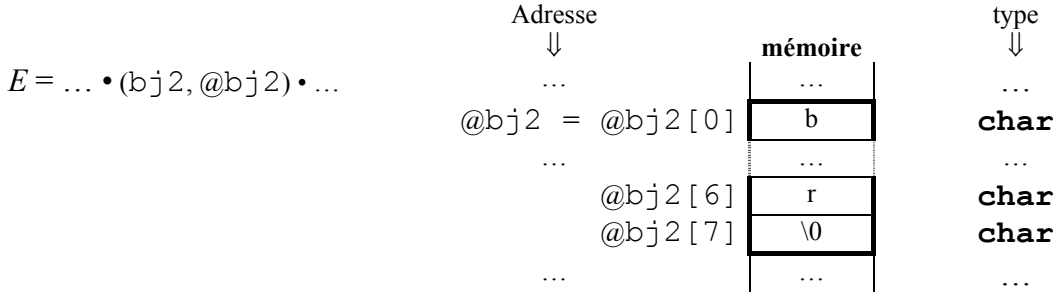
```
char* bj1      = "bonjour" ;
char  bj2[]    = "bonjour" ;
char  bj3[30] = "bonjour" ;
```

Elles le sont tant que les variables sont exploitées en lecture. Un examen plus détaillé sur ce qu'il se passe en mémoire montre toutefois clairement la différence :

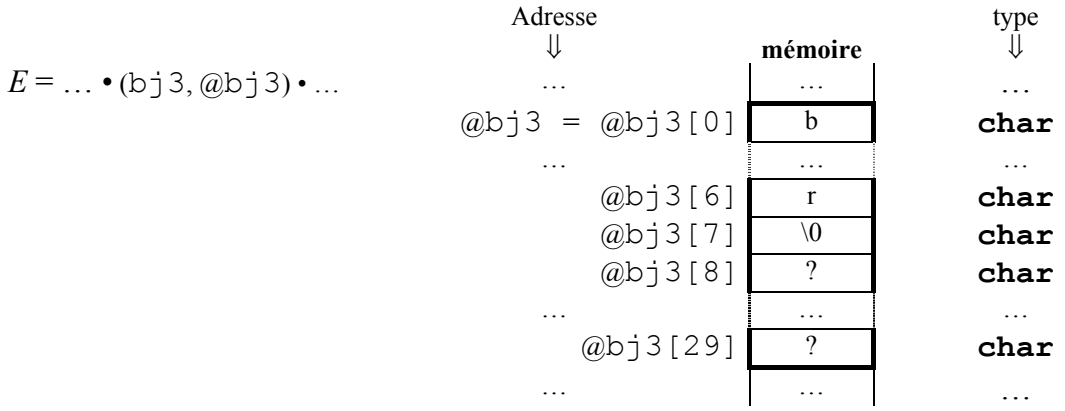
- La variable `bj1` est un vrai pointeur, initialisé comme pointant sur un tableau de caractères constants :



- La variable `bj2` est un vrai tableau, de taille 8, toutes les composantes étant initialisées explicitement :



- La variable `bj3` est un vrai tableau, de taille 30, dont seules les 8 premières composantes sont initialisées explicitement :



Pour la variable `bj1`, aucune écriture n'est autorisée, puisque la chaîne de caractères pointée est une constante. Pour les variables `bj2` ou `bj3`, les accès en écriture sont possibles, dans la limite de leurs tailles respectives.

11.2. LES STRUCTURES

Un type **structure** est un produit de types quelconques. L'accès aux composantes s'effectue néanmoins via un système de noms.

Exemple : la description d'un individu incluant son nom, son prénom, son âge et son poids est le produit de deux types textes (nom et prénom) et deux types réels positifs (âge et poids).

11.2.1. Déclaration d'un type structure

La déclaration d'un type structure de nom *T* est notée ainsi en C++ :

```
struct T ;
```

Comme les fonctions, il peut exister plusieurs déclarations d'un même type *T* au sein d'un texte C++.

11.2.2. Définition d'un type structure

La définition d'un type structure de nom *T* constitué de *n* composantes, chacune d'un type T_i, est notée ainsi en C++ :

```
struct T {
    T1 m1;
    …
    Tn mn;
};
```

Chaque m_i est le nom d'un **membre donnée** (ou **attribut**, ou **champs**) de la structure, i.e. de l'une des composantes du type produit. Nous verrons par la suite que ces noms servent à désigner les diverses composantes du produit. Ils remplacent la notion d'indice vue pour les tableaux.

Exemple : le type décrivant un individu, évoqué dans l'introduction de cette section :

```
struct unePersonne {
    char* sonNom;
    char* sonPrénom ;
    float sonAge ;
    float sonPoids ; };
```

Exemple : le type décrivant un nombre complexe, c'est à dire un nombre ayant deux composantes réelles, l'une dite partie réelle, l'autre partie imaginaire :

```
struct unComplexe {
    double re,    // la partie réelle
           im ;   // la partie imaginaire
};
```

Nous aurions pu représenter un nombre complexe par un tableau, puisqu'il s'agit d'un produit de types homogènes. Nous aurions alors écrit :

```
typedef double unComplexe[2] ;
```

Une telle définition de type peut aussi bien être écrite en dehors d'un bloc (définition globale) qu'à l'intérieur d'un bloc (définition locale). Les règles de portée de la définition ainsi écrite s'appliquent normalement.

11.2.3. Définition d'une variable structure

Soit *T* un type structure défini selon :

```
struct T {
    T1 m1;
    …
    Tn mn;
};
```

La définition d'une variable *nom* d'un type structure nommé *T* s'écrit :

```
T nom ; // variable non initialisée
```

ou

```
T nom = { e1, e2, …, em } ; // variable initialisée
```

La première forme définit la structure sans préciser comment seront initialisées les composantes ; en général, les composantes sont initialisées avec la valeur par défaut du type T_i.

La seconde forme définit comment sont initialisées les *m* premières composantes ($m \leq n$) ; chaque composante reçoit la valeur de l'expression e_i correspondante : la première composante reçoit la valeur de e_1, la seconde celle de e_2, etc. Si $m=n$, alors toutes les composantes sont explicitement initialisées. Sinon, les *n-m* composantes non initialisées explicitement reçoivent la valeur par défaut de leurs types respectifs.

Exemple : si un individu est représenté par :

```
struct unePersonne {
    char* sonNom;
    char* sonPrénom ;
    float sonAge ;
    float sonPoids ;
};
```

alors nous pourrions définir :

```
unePersonne lui  = { "Dupont", "Jean-Paul", 29, 83 } ;
unePersonne elle = { "Dupont", "Hélène", 28, 62 },
            inconnu ;
```

La première variable représente Jean-Paul Dupont, âgé de 29 ans, pesant 83kg ; la seconde Hélène Dupont, 28 ans, 62kg ; la dernière variable définit un individu dont aucune composante n'est explicitement initialisée.

Si la définition d'un type structure est suivie de définitions de variables de ce type, il est possible de fusionner ces définitions en une seule.

Exemple : l'exemple précédent peut s'écrire :

```
struct unePersonne {
    char* sonNom;
    char* sonPrénom ;
    float sonAge ;
    float sonPoids ;
} lui  = { "Dupont", "Jean-Paul", 29, 83 },
  elle = { "Dupont", "Hélène", 28, 62 },
  inconnu ;
```

Du point de vue de l'environnement, les composantes d'une structure sont toutes rangées dans des cases mémoire adjacentes, comme les tableaux. L'adresse associée à une structure est celle de sa première composante.

Exemple : la définition de la variable suivante de type `unComplexe` :

```
unComplexe i = { 0, 1 } ;
```

conduit à l'environnement:

$E = \ldots \bullet (\texttt{i}, \texttt{@i}) \bullet \ldots$

adresse ⇓	mémoire	type ⇓
…	…	…
@i = @i.re	0	**double**
@i.im	1	**double**
…	…	…

Une variable ou valeur de type structure est appelée **objet** en C++. Cette appellation est liée à l'aspect **programmation par objets** que permet C++, qui n'est pas au programme de ce cours.

11.2.4. Opérateurs d'accès

Si *o* est une variable structure de type *T* et *p* un pointeur sur une valeur structure de même type, alors l'accès à la composante (au membre) de nom *m* est noté :

o`.`*m*	accès au membre *m* pour un objet *o*
p`->`*m*	accès au membre *m* d'un objet via le pointeur *p*

Les opérateurs d'accès `.` et `->` sont prioritaires sur les opérateurs unaires à gauche comme `+`, `-`, `*` ou `&`. Ces deux opérateurs concrétisent la notion de projection abordée lors de la présentation générale sur les types produits.

Exemple :

expression non parenthésée	expression parenthésée équivalente
`-p->m + +*o.m`	`(-(p->m)) + (+(*(o.m)))`

Chaque membre d'une variable structure est une variable à part entière. Les égalités suivantes sont toujours satisfaites :

$(\&o)->m = o.m$ et $(*p).m = p->m$

Exemple : le programme suivant définit une date comme le produit de trois entiers. Chaque composante est demandée à l'usager via une fonction de saisie.

```
unsigned saisir(char* msg, unsigned min, unsigned max) {
    unsigned n;
    do { // la demande de saisie est répétée tant que la valeur saisie n'est pas
         cout << msg ;              // dans l'intervalle [min, max]
         cin >> n ;
    } while (n<min || n>max) {
    return n ;
} ;

struct uneDate {
    unsigned short jour,
                   mois ;
    unsigned int   année ;
} ;

main() {
    uneDate d ;
    d.jour = saisir(" jour : ", 1, 31) ;
    d.mois = saisir(" mois : ", 1, 12) ;
    d.année = saisir("année : ", 1900, 2100) ;
    cout << "Vous avez saisi la date suivante : "
         << d.jour << '/'
         << d.mois << '/'
         << d.année << '\n' ;
} ;
```

11.2.5. Opérations prédéfinies

Les opérations définies sur un membre d'une structure sont celles associées au type de ce membre. Par contre, une structure en tant que telle (i.e. considérée comme un tout) ne dispose que de deux opérations de base : l'affectation entre structures, et l'initialisation à partir de structures existantes.

Dans le cas de l'affectation d'une variable structure avec une valeur structure de même type, cette affectation se ramène à une affectation membre à membre.

Exemple : avec le type `uneDate` défini dans l'exemple précédent, si deux variables sont définies ainsi :

```
uneDate d1 = { 29, 2, 2000 }, d2 ;
```

alors l'affectation :

```
d2 = d1 ;
```

est équivalente à :

```
d2.jour = d1.jour ;
d2.mois = d1.mois ;
d2.année = d1.année ;
```

Dans le cas de l'initialisation d'une variable structure à partir d'une valeur structure de même type, cette initialisation se ramène à une initialisation membre à membre.

Exemple : avec le type `uneDate` défini dans l'exemple précédent, si l'on définit :

```
uneDate d1 = { 29, 2, 2000 };
```

alors la définition suivante est licite :

```
uneDate d2 = d1 ; // initialisation à partir d'une structure existante
```

et est équivalente à :

```
uneDate d2 = { d1.jour, d1.mois, d1.année } ;
```

Comme nous l'avons déjà signalé, attention de ne pas confondre affectation et initialisation : bien que l'écriture soit pratiquement identique (signe = exploité dans les deux cas), le code exécuté pour une affectation n'est pas le même que celui exécuté pour une initialisation, bien que, en général, leurs effets soient les mêmes.

11.2.6. Structures imbriquées

Comme l'indique la définition des structures, les membres d'une structure peuvent être de n'importe quel type, y compris un type tableau ou structure…

Exemple : en reprenant le type `uneDate`, il nous est possible de définir :

```
struct unePersonne {
    char    nom[10],
            prénom[20] ;
    uneDate naissance ;
} ;
```

Ainsi, nous pourrions définir :

```
UnePersonne JD = { "Dupont", "Jean", { 20, 10, 1956 }} ;
```

Noter l'imbrication des initialisations. Pour connaître l'année de naissance de Jean Dupont, il suffit d'écrire :

```
JD.naissance.année
```

Pour connaître la troisième lettre de son prénom (celle d'indice 2), il suffit d'écrire :

```
JD.prénom[2]
```

`JD` est en effet `unePersonne`, dont trois membres données sont accessibles : `nom`, `prénom` et `naissance`. Or la valeur du membre donnée `naissance` est elle-même décomposable en trois membres, ceux d'`uneDate`.

12. LES SOMMES DE TYPES

Si T_1, T_2, …, T_n sont n types quelconques, leur somme est notée $S = T_1 + T_2 + \ldots + T_n$. Toute valeur v de type S est associée (mise en correspondance) à une valeur unique ayant nécessairement pour type l'un des T_i.

Si v est une valeur d'un type somme $T_1 + T_2 + \ldots + T_n$, et qu'il existe un moyen de connaître de quel type T_i relève v, alors la valeur v_i qui lui est associée est calculée par une **sélection** σ_i :

$$\sigma_i : T_1 + T_2 + \ldots + T_n \rightarrow T_i \text{ , avec } \sigma_i(v) = v_i$$

Une fonction κ capable d'associer à une valeur x d'un type T une valeur v de type $T_1 + T_2 + \ldots + T_n$ est appelée un **constructeur** :

$$\kappa : T \rightarrow T_1 + T_2 + \ldots + T_n \text{ , avec } \kappa(x) = v$$

Tout langage fournit en standard les constructeurs de base κ_i tels que :

$$\kappa_i : T_i \rightarrow T_1 + T_2 + \ldots + T_n \text{ , avec } \kappa_i(v_i) = v \text{ et } \sigma_i(v) = \sigma_i(\kappa_i(v_i)) = v_i$$

12.1. LES ENUMERATIONS

Dans les langages fonctionnels courants (C++ n'est pas un langage fonctionnel), une **énumération** est définie comme une forme particulière de somme de types : chaque type figurant dans la somme ne possède qu'une seule valeur. A chacune de ces valeurs uniques correspond un constructeur, lequel permet d'obtenir une valeur du type somme.

Cette définition s'applique à la notion d'énumération telle qu'elle existe dans C++. Un tel type somme est aussi appelé **type énuméré**.

12.1.1. Déclaration d'un type énuméré

La déclaration d'un type énuméré précise le nom du nouveau type introduit. L'écriture d'une telle déclaration est :

```
enum nom;
```

où *nom* est le nom du type. Il peut exister plusieurs déclarations d'un même type énuméré au sein d'un texte C++.

12.1.2. Définition d'un type énuméré

La définition d'un type énuméré précise le nom du nouveau type introduit ainsi que la liste des noms des littéraux autorisés pour ce type. L'écriture d'une telle définition est :

```
enum nom { n1, n2, …, nk } ;
```

où *nom* est le nom du type et chaque n_i est le nom d'un littéral admissible pour le type.

Exemple : un type énumérant des noms de couleurs :

```
enum uneCouleur { rouge, vert, bleu, jaune, orange } ;
```

Dans cet exemple, faisons le lien avec la présentation générale des sommes de types en début de chapitre : le type `uneCouleur` est bien une somme de types ; en effet :

$$\texttt{uneCouleur} = C_{\texttt{rouge}}+C_{\texttt{vert}}+C_{\texttt{bleu}}+C_{\texttt{jaune}}+C_{\texttt{orange}}$$

Aucun des 5 types $C_{\dots}$ apparaissant dans cette somme n'est explicite en C++ (aucun n'a un nom C++) ; considérons le type $C_{\texttt{rouge}}$: il ne possède qu'une seule valeur, que nous notons $V_{\texttt{rouge}}$, elle aussi non explicite en C++ (elle n'a pas de nom). Nous pouvons affirmer que la valeur `rouge`, de type `uneCouleur`, est associée à la valeur $V_{\texttt{rouge}}$ de type $C_{\texttt{rouge}}$; `rouge` est donc bien associée à une valeur unique de l'un des types figurant dans la somme. Chaque nom de littéral d'un type énuméré fait donc office de constructeur de ce type.

Le nom d'un littéral ne peut pas être surchargé par plusieurs types énumérés distincts, i.e. un nom de littéral ne peut être introduit que par un seul type énuméré.

Exemple : les définitions de types ci-dessous posent problème : le nom `orange` est utilisé dans deux types énumérés distincts, ce que C++ n'autorise pas :

```
enum uneCouleur { rouge, vert, bleu, jaune, orange } ;
enum unFruit { banane, orange, pomme, raisin } ;
```

12.1.3. Définition d'une variable de type énuméré

La définition d'une variable ou d'une constante d'un type énuméré *T* suit le schéma classique :

```
T nom ; // pas d'initialisation explicite
```

ou

```
T nom = exp ; // initialisation explicite avec une expression simple
```

Exemple : en reprenant le type des couleurs précédent :

```
uneCouleur c1, c2 = rouge, c3 = c2 ;
const uneCouleur red = rouge, yellow = jaune ;
```

Le passage d'un paramètre de type énuméré ne pose pas de difficulté particulière.

Exemple : une fonction passant trois couleurs, chacune avec un mode de passage différent :

```
void fonc(uneCouleur c1, uneCouleur &c2, uneCouleur *c3);
```

12.1.4. Opérations prédéfinies

Si *n* est une valeur affectable d'un type énuméré *T*, *a* et *b* deux valeurs quelconques de ce même type *T*, alors les opérations suivantes sont définies sur le type *T* :

opération	**sémantique**
n = *a*	affectation
a © *b*	test exploitant l'opérateur de comparaison ©, où © est au choix == != < <= > >=

Si besoin, le compilateur peut convertir toute valeur d'un type énuméré en un nombre entier. Par défaut, le premier littéral est codé 0, le suivant (dans l'ordre de définition du type) est codé 1, le suivant 2, etc. Le passage inverse (d'un entier vers une valeur d'un type énuméré) n'est pas automatique ; il nécessite un opérateur explicite de conversion, et de plus s'avère risqué (aucun contrôle n'est fait à l'exécution pour vérifier qu'il existe bien un littéral dont le code correspond à l'entier converti).

Exemple : voici un bloc parfaitement valide :

```
{    enum uneCouleur { rouge, vert, bleu } ;
     uneCouleur c = bleu ;
     int i = c ; // i vaut 2 (code de la couleur)
     c = (uneCouleur)10 ; // demande de conversion explicite de l'entier
}                         // 10 en une couleur. Pas de contrôle fait !
```

D'une certaine façon, il est possible de considérer qu'un type énuméré en C++ n'est jamais plus qu'un type entier possédant des constantes dont les noms sont fixés une fois pour toute.

Exemple : la définition du type énuméré suivant :

```
enum uneCouleur { rouge, vert, bleu, jaune, orange } ;
```

pourrait s'écrire:

```
typedef short int uneCouleur;
const uneCouleur rouge=0, vert=rouge+1, bleu=vert+1,
                 jaune=bleu+1, orange=jaune+1 ;
```

Cette écriture correspond au codage effectif auquel procède un compilateur C++ ; néanmoins, le type énuméré `uneCouleur` est un vrai nouveau type, alors que la seconde écriture (avec le **`typedef`**) n'introduit pas un nouveau type, mais se contente de nommer différemment un type existant (en l'occurrence **`short int`**).

Exemple : presque tous les types primitifs en C++ sont des types énumérés, en tout cas virtuellement (en prenant quelques libertés sur les noms des littéraux et leur surcharge potentielle) :

```
enum void {};
enum bool { false, true };
enum char { '\0', '\1', …, 'A', 'B', …, 'a', 'b', … };
enum short { -128=-128, -127, -126, …, -1, 0, 1, …, 127 };
enum unsigned short { 0, 1, …, 254, 255 };
…
enum int { -32768=-32768, -32767, …, -1, 0, 1, …, 32767 };
```

12.2. LES UNIONS

Un type **union** est une somme de types quelconques.

Exemple : la description d'un passager qui peut être soit un homme (au sens être humain) soit un animal de compagnie est une somme des types *homme* et *animal de compagnie*. A tout instant, un passager ne peut être que de l'un de ces types et un seul.

12.2.1. Définition d'un type union

La définition d'un type union de nom *T* constitué par la somme de *n* composantes, chacune d'un type T_i, est notée ainsi en C++ (syntaxe pratiquement identique à la définition d'une structure) :

```
union T {
    T1 m1;
    …
    Tn mn;
};
```

Chaque m_i est le nom d'un **membre donnée** (ou **attribut**, ou **champs**) de l'union, i.e. de l'une des composantes potentiellement active du type somme.

Exemple : le type décrivant un passager, évoqué dans l'introduction de cette section :

```
struct unePersonne {
    char *sonNom, *sonPrénom ;
    float sonAge ;
};

struct unAnimal { char *sonNom, *saRace ; };

union unPassager {
    unePersonne laPersonne;
    unAnimal    lAnimal ;
};
```

Exemple : le type décrivant un point du plan qui peut être repéré soit par ses coordonnées cartésiennes (x, y), soit ses coordonnées polaires (ρ, θ) :

```
struct desCoordonneesCartesiennes { double x, y ; };

struct desCoordonneesPolaires { double rho, theta ; };

union unPoint {
    desCoordonneesCartesiennes cart;
    desCoordonneesPolaires     pol;
};
```

Une telle définition de type peut aussi bien être écrite en dehors d'un bloc (définition globale) qu'à l'intérieur d'un bloc (définition locale). Les règles de portée de la définition ainsi écrite s'appliquent normalement.

12.2.2. Définition d'une variable union

Soit *T* un type union défini selon :

```
union T {
    T1 m1;
    …
    Tn mn;
};
```

La définition d'une variable *nom* d'un type union *T* s'écrit :

```
T nom ; // variable non initialisée
```

ou

```
T nom = { e } ; // variable initialisée
```

La première forme initialise la variable avec la valeur par défaut du premier membre de l'union.

La seconde forme initialise la variable avec la valeur de l'expression *e* qui doit impérativement être du type du premier membre déclaré de l'union. Il n'y a pas moyen d'initialiser une variable union avec une valeur d'un des types des autres membres figurant de la définition de l'union, comme nous avions procédé avec les structures. Cette contrainte sera levée lorsque nous étudierons la notion de constructeur dans la section sur les classes.

Rappelons-nous que, contrairement aux structures, un seul membre d'une union est actif (exploitable) à un instant donné, les autres membres devenant alors obsolètes ; dans une structure, les membres ont tous une existence indépendante les uns des autres. Dans une union, activer un membre, c'est éliminer les autres.

Exemple : avec un tel type union :

```
union T { int a ; double b ; char c ;} ;
```

nous pourrions définir :

```
T x  = { 83 } ; // seul le premier membre a est initialisable
T y ;
```

La première variable a son membre `a` initialisé à 83 ; la seconde variable a son membre `a` initialisé par défaut.

Comme pour les structures, une définition d'un type union suivie de définitions de variables de ce type peut être fusionnée en une seule définition.

Exemple : l'exemple précédent peut s'écrire :

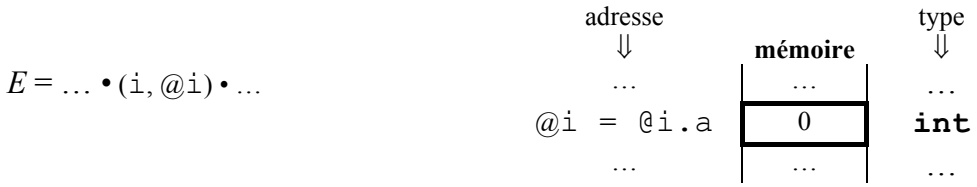
```
union T { int a ; double b ; char c ;} x = { 83 }, y;
```

Du point de vue de l'environnement, la valeur d'une union est rangée dans (au moins) une case mémoire dont le type est celui de l'un des membres (nous parlerons du **membre actif**) de l'union à un instant donné.

Exemple : en reprenant le type T ci-dessus, la définition de la variable suivante :

```
T i = { 0 } ;
```

conduit à l'environnement:

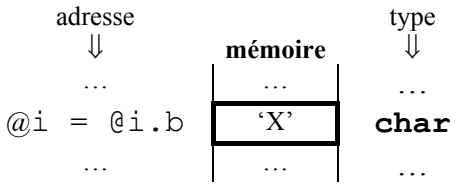


Si la définition de variable précédente est suivie d'une affectation sur un autre membre de l'union, comme par exemple (voir la section suivante pour les opérateurs d'accès à une union) :

```
i.c = 'X' ;
```

alors l'environnement devient :

$E = \ldots \bullet (\texttt{i}, @\texttt{i}) \bullet \ldots$



Notez bien que la valeur est bien stockée à la même adresse (@i), et que la case mémoire stocke maintenant une valeur d'un autre type (la case d'adresse @i portait une valeur de type **`int`**, laquelle a été remplacée par une valeur de type **`char`**).

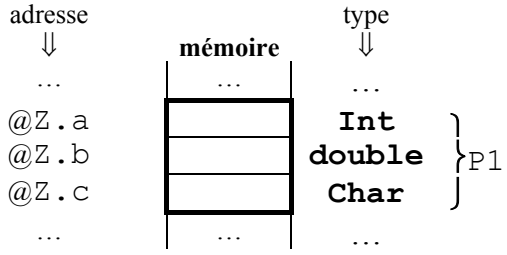
Cet exemple montre bien que, à la différence d'une structure qui associe au moins une case mémoire distincte à chacun de ses membres, une union n'associe des cases mémoires qu'à un membre à la fois (une seule pour l'exemple).

Si un membre d'une union est d'un type nécessitant l'allocation de plus d'une case mémoire, toute valeur du type union se verra allouer un espace mémoire égal au plus grand des espaces mémoire nécessaires au codage de chacun des membres de l'union.

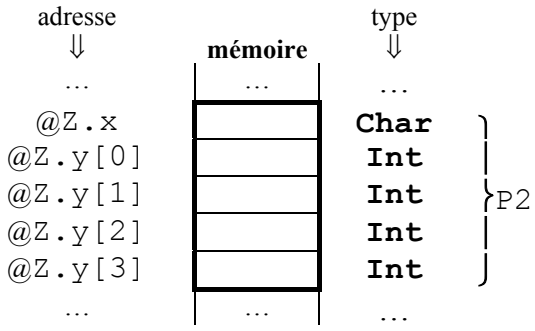
Exemple : supposons que soient définis les types suivants :

```
struct P1 { int a ; double b ; char c ;};
struct P2 { char x ; int y[4] ;} ;
union U { P1 u ; double v ; P2 w ;} ;
```

Le type `P1` implique pour chacune de ses valeurs l'allocation de 3 cases mémoires :



Le type `P2` implique pour chacune de ses valeurs l'allocation de 5 cases mémoires :

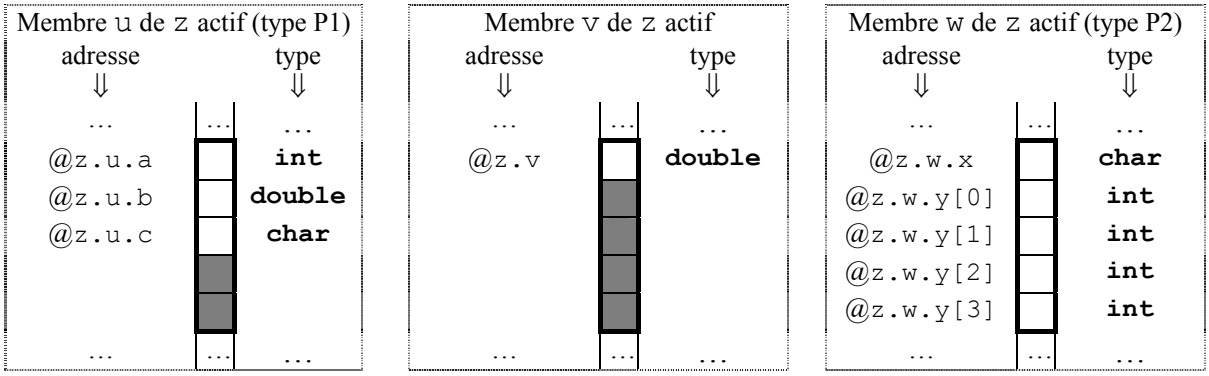


Le type `U` doit permettre de représenter des valeurs qui sont au choix de type `P1`, `P2` ou **`double`**. Une valeur de type `P1` (membre `u` de l'union) nécessite 3 cases mémoire, une valeur de type `P2` (membre `w`) 5, et 1 pour le type **`double`** (membre `v`).

Le membre le plus consommateur est donc `w` : le nombre de cases allouées à toute valeur de type `U` sera donc 5. Selon le membre actif, certaines cases ne seront tout simplement pas exploitées (mais tout de même réservées). Avec la définition suivante :

```
U z ;
```

voici tous les cas de figure possibles sur l'exploitation des cases mémoires associées à `z` :



Une variable ou valeur de type union est appelée **objet** en C++, comme les structures.

12.2.3. Opérateurs d'accès

Si *o* est une variable union de type *S* et *p* un pointeur sur une valeur union de même type, alors l'accès à la composante (au membre) de nom *m* est noté :

o`.`*m*	accès au membre *m* pour un objet *o*
p`->`*m*	accès au membre *m* d'un objet via le pointeur *p*

Les opérateurs d'accès `.` et `->` ont les mêmes priorités que celles étudiées pour les structures. Ces deux opérateurs concrétisent la notion de sélection abordée lors de la présentation générale sur les types sommes.

Noter qu'accéder à un membre d'une union, c'est, de fait, activer ce membre, et donc rendre obsolètes tous les autres. Une autre façon de décrire cette propriété est d'user de la notion de *point de vue* : accéder au membre *m* de type *T* d'un objet *o* d'un type union *S*, c'est adopter sur *o* un point de vue *T* : l'objet *o*, de type *S*, sera vu comme l'objet *o.m*, de type *T*. Autrement dit, un opérateur d'accès à un objet union est un opérateur qui change le point de vue sur l'objet.

Chaque membre d'une variable union est une variable à part entière.

12.2.4. Opérations prédéfinies

Les opérations définies sur un membre d'une union sont celles associées au type de ce membre. Par contre, une union en tant que telle (i.e. considérée comme un tout) ne dispose que de deux opérations de base : l'affectation entre unions, et l'initialisation à partir d'unions existantes (comme les structures).

Dans le cas de l'affectation d'une variable union avec une valeur union de même type, cette affectation se ramène à une copie du contenu des cases mémoires associées au membre de l'union à affecter.

Exemple : avec le type `T` défini dans l'exemple précédent, si deux variables sont définies ainsi :

```
T v1, v2 ;
v1.c = 'X' ;
```

alors l'affectation :

```
v2 = v1 ;
```

est équivalente à :

```
v2.c = v1.c ;
```

Dans le cas de l'initialisation d'une variable union à partir d'une valeur union de même type, cette initialisation se ramène à une copie du membre.

Exemple : avec le type `T` défini dans l'exemple précédent, si l'on définit :

```
T v1 ;
v1.c = 'X' ;
```

alors la définition suivante est licite :

```
T v2 = v1 ; // initialisation à partir d'une union existante
```

et est équivalente à :

```
T v2 ; v2.c = v1.c ;
```

12.2.5. Unions avec membre actif détectable

Une difficulté des unions en C++ est qu'il n'est pas possible, pour une valeur d'un type union donné, de savoir quel est son membre actif à un instant donné.

Exemple : considérons le programme suivant :

```
union T { int a ; double b ; char c ;} ;

main() {
    T v;
    {
        char choix ;
        cout << "Quel membre de V saisir [a, b ou c] : " ;
        cin >> choix ;
        if (choix=='a' || choix=='b' || choix=='c') {
            cout << "V." << choix << " = " ;
            switch(choix) {
                case 'a' : cin >> v.a ; break ;
                case 'b' : cin >> v.b ; break ;
                case 'c' : cin >> v.c ;
    }   }   } ;
    // point [1]
}
```

Au point `[1]`, il n'y a plus aucun moyen pour déterminer lequel des membres `a`, `b` ou `c` est actif. La seule variable nous permettant de retrouver cette information est la variable `choix` ; or le point `[1]` est hors de portée de cette variable.

Si le problème à traiter nécessite de connaître le membre actif d'une variable union, l'exemple précédent montre la voie à suivre : il nous faut impérativement une autre variable nous permettant de retrouver cette information.

La façon la plus élégante de procéder consiste à définir un type énuméré contenant autant de littéraux (constantes) que de membres définis dans l'union. Il suffit alors d'encapsuler le type union dans une structure, le membre ajouté par la structure servant justement à déterminer le membre actif dans l'union.

Exemple : la transformation du type union `T` de l'exemple précédent consiste à le redéfinir ainsi :

```
enum membreActifDeT { membreA, membreB, membreC } ;

union membreUnionDeT { int a ; double b ; char c ;} ;

struct T {
    membreActifDeT leMembreActif ;
    membreUnionDeT leMembreUnion ;
} ;

main() {
    T v;
    {
        char choix ;
        cout << "Quel membre de V saisir [a, b ou c] : " ;
        cin >> choix ;
        if (choix=='a' || choix=='b' || choix=='c') {
            cout << "V." << choix << " = " ;
            switch(choix) {
                case 'a' :
                    cin >> v.leMembreUnion.a ;
                    v.leMembreActif = membreA ;
                    break ;
                case 'b' :
                    cin >> v.leMembreUnion.b ;
                    v.leMembreActif = membreB ;
                    break ;
                case 'c' :
                    cin >> v.leMembreUnion.c ;
                    v.leMembreActif = membreC ;
                    break ;
    }   }   } ;
    // point [1]
}
```

Au point `[1]`, il suffit désormais de tester `v.leMembreActif`, et ainsi décider d'exploiter l'un des membres `v.leMembreUnion.a` à `v.leMembreUnion.c`.

Le langage C++ offre quelques possibilités de simplification pour ce genre de constructions :

- il est possible de définir un type *E* comme étant encapsulé dans un autre type *T*. Le type encapsulé est manipulable comme n'importe quel type, excepté que son nom sera noté *T*`::`*E* ; le préfixe *T*`::` doit précéder le nom des

littéraux si *E* est un type énuméré. Cette encapsulation permet de réutiliser le nom *E* dans d'autres types encapsulés.

- il est possible, en encapsulant un type union *E*, d'omettre son nom (le type union est alors dit **anonyme**), et de ne pas lui associer de membre dans *T* ; dans ce cas, les membres du type union deviennent des membres à part entière du type *T* qui l'encapsule.

Exemple : avec ces possibilités, notre définition de `T` devient :

```
struct T {
    enum membreActif { membreA, membreB, membreC } ;
        // type énuméré membreActif encapsulé
    membreActif leMembreActif ;
    union { int a ; double b ; char c ;} ;
        // type union anonyme encapsulé : a, b et c sont membres de T
} ;
```

La structure ne porte que deux membres :

- le membre `leMembreActif`
- au choix le membre `a`, le membre `b` ou le membre `c`

Voici comment cette structure pourrait être exploitée :

```
main() {
    T v;
    {
        char choix ;
        cout << "Quel membre de V saisir [a, b ou c] : " ;
        cin >> choix ;
        if (choix=='a' || choix=='b' || choix=='c') {
            cout << "V." << choix << " = " ;
            switch(choix) {
                case 'a' :
                    cin >> v.a ;
                    v.leMembreActif = T::membreA ;
                    break ;
                case 'b' :
                    cin >> v.b ;
                    v.leMembreActif = T::membreB ;
                    break ;
                case 'c' :
                    cin >> v.c ;
                    v.leMembreActif = T::membreC ;
                    break ;
    }   }   } ;
    // point [1]
}
```

Noter le préfixe `T::` devant tout littéral du type énuméré encapsulé, ainsi que l'accès direct aux membres de l'union à partir de la structure encapsulante.

13. LES TYPES RECURSIFS

Tous les types décrits jusqu'ici, qu'ils soient prédéfinis ou construits par l'usager, se caractérisent par le fait que toute valeur d'un de ces types nécessite toujours le même espace mémoire (nombre de cases mémoire) pour la coder.

Il arrive toutefois que, dans certains cas, nous souhaitions que la taille de l'espace mémoire associé à une valeur puisse varier ; prenons l'exemple d'un tableau qui, initialement défini comme comportant 20 éléments au maximum, devrait désormais mémoriser 30 éléments supplémentaires, portant l'effectif à 50. En l'état, nous sommes incapables de répondre à ce besoin (toute solution produite à partir de ce que nous connaissons ne peut aboutir qu'à la définition d'un tableau de taille plus grande dès le départ, mais dont seulement une partie des composantes seront exploitées ; les composantes restantes peuvent être exploitées au besoin, mais dans la limite rédhibitoire de la taille fixée au départ).

13.1. MOTIVATION : L'EXEMPLE DES LISTES

Supposons qu'il existe un type nommé `liste`, représentant une sorte de tableau d'entiers dont la taille serait variable (il est possible à tout instant de modifier le nombre d'éléments d'une liste).

Supposons que les fonctions suivantes soient définies (*L* est supposée être une liste et *e* un entier) :

`liste vide()`	`vide()` retourne la liste vide
`bool` `vide(liste)`	`vide(`*L*`)` retourne vrai si la liste est vide, faux sinon
`int` `tête(liste)`	`tête(`*L*`)` retourne le premier élément de la liste *L* ; la liste *L* est supposée être non vide (résultat indéfini sinon)
`liste reste(liste)`	`reste(`*L*`)` retourne une liste composée des mêmes éléments que *L* privée de sa tête ; la liste *L* est supposée être non vide (résultat indéfini sinon)
`liste ajout(`**`int`**`, liste)`	`ajout(`*e*`, `*L*`)` retourne une liste dans la tête est l'entier *e* et le reste la liste *L*

Adoptons les notations suivantes (attention, ces notations ne sont pas valides en C++) :

- la liste vide sera notée `[]`
- la liste `[`e_1, e_2, …, e_n`]` est composée de *n* éléments e_i.

Posons que, si une variable de type `liste` est définie sans être initialisée, elle sera de fait initialisée comme étant vide.

Exemple : soit L_1 = [2, 4, 6, 8] et L_2 = []

```
vide(L1)          = faux
vide(L2)          = vrai
tête(L1)          = 2
tête(L2)          = ?
reste(L1)         = [4, 6, 8]
ajout(1, L1)      = [1, 2, 4, 6, 8]
ajout(3, L2)      = [3]
```

Exemple : voici les instructions permettant de définir L_1 et L_2 ci-dessus

```
liste L1 ; // L1 vide si non initialisée
L1 = ajout(2, ajout(4, ajout( 6, ajout(8, vide() ) ) ) ) ;
liste L2 = vide() ; // initialisation explicite, pas vraiment utile…
```

Ces quelques exemples montrent que, au cours de l'exécution d'un programme, une même variable `liste` peut avoir un nombre d'éléments variable.

13.1.1. Une analyse du type liste

Examinons la notion de liste d'un point de vue des types ; une liste est :

- soit la liste vide
- soit composée d'un entier (sa tête) et d'une liste (son reste)

Nous avons déjà rencontré des constructions de types où les valeurs étaient soit d'un type, soit d'un autre : les sommes de type. En l'occurrence, nous pouvons déjà écrire que le type `liste` est la somme de deux types :

- le type de la liste vide ; soit `uneListeVide` le nom de ce type, et `laListeVide` la seule valeur possible pour ce type.
- un type qui possède deux composantes, l'une qui est un entier, l'autre une liste.

Pour ce dernier type, notons que les deux composantes existent en même temps (elles sont même complètement indépendantes). La construction de type adaptée à ce cas est le produit. Le type qui nous intéresse est le produit de deux types : le type entier et le type liste. Appelons `unChainon` ce type produit.

Nous pouvons tenter d'écrire en C++ les types définis jusqu'ici :

```
enum uneListeVide { laListeVide } ;

struct unChainon {
    int   laTête ;
    liste leReste ;
} ;
```

Le type énuméré `uneListeVide` nous permet de définir un nouveau type ne possédant qu'une seule valeur. La construction **`struct`** pour `unChainon` est la seule qui convienne pour représenter un produit de types différents.

Ces définitions supposent que le type `liste` existe ; nous savons qu'une liste est soit une liste vide, soit un chaînon. Autrement dit, en C++, nous devons écrire :

```
union liste {
    uneListeVide v ;   // v pour liste vide
    unChainon    nv ; // nv pour liste non vide
} ;
```

En fait, si l'on souhaite pouvoir connaître lequel des membres `v` ou `nv` est actif pour une valeur de ce type, il nous faut un type plus élaboré (cf. la section sur les types union à membre actif détectable) ; l'écriture finale en C++ ressemble donc plutôt à :

```
enum uneListeVide { laListeVide } ;

struct liste ;

struct unChainon {
    int laTête ;
    liste leReste ;
} ;

struct liste {
    enum quelChamp { cestV, cestNV } lequel ;
    union {
        uneListeVide v ;
        unChainon    nv ;
    }
} ;
```

Cette écriture respecte l'une des contraintes de C++ : à chaque fois qu'un nom est utilisé dans un programme, il doit avoir au moins été déclaré au préalable. C'est ce qui justifie la déclaration :

```
struct liste ;
```

juste avant la définition :

```
struct unChainon { …
```

laquelle définit un membre de type `liste`. Notons que ce type est **récursif** : dans sa définition, il introduit une composante qui est elle-même une liste (une liste peut être bâtie sur un chaînon, lui-même formé d'une liste).

13.1.2. Une définition des fonctions de base sur les listes

A supposer que ces types soient définissables, nous pouvons tenter de définir les fonctions indiquées en préambule :

```
liste vide() { liste L ;
    L.lequel = liste::cestV ; L.v = laListeVide ;
    return L ; } ;

bool vide(liste L) { return L.lequel == liste::cestV ; }

int tête(liste L) {
    if (vide(L)) return -1 ; // traitement arbitraire de l'erreur
    return L.nv.laTête ;
} ;
```

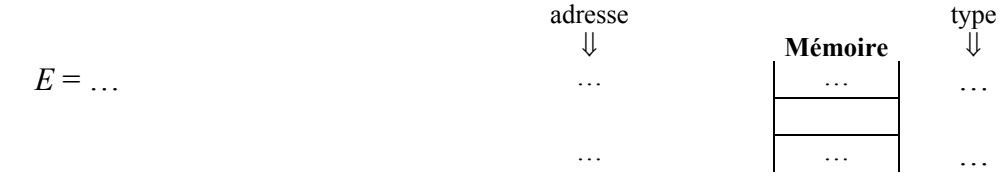
```
liste reste(liste L) {
    if (vide(L)) return L ; // traitement arbitraire de l'erreur
    return L.nv.leReste ;
} ;

liste ajout(int e, liste L) {
    liste R ;
    R.lequel = liste::cestNV ;
    R.nv.laTête = e ;
    R.nv.leReste = L ;
    return R ;
} ;
```

13.1.3. Pas si simple…

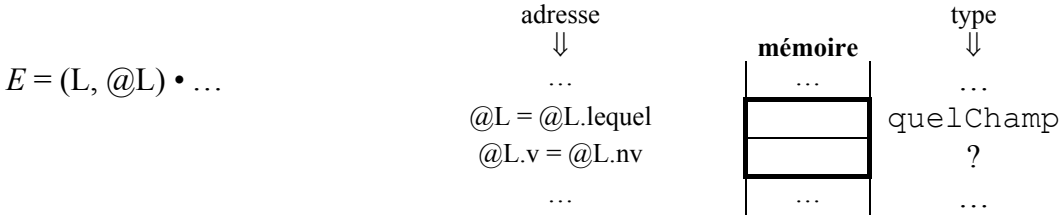
La définition du type `liste` ci-dessus ne semble guère poser de problème. Malheureusement, tout n'est pas si simple en C++ ; pour bien comprendre d'où provient la difficulté, il suffit de se poser la question suivante :

> *Quelles sont les conséquences sur l'environnement d'une définition de variable de type liste, i.e. comment est transformé l'environnement après cette instruction :*
>
> `liste L ;`

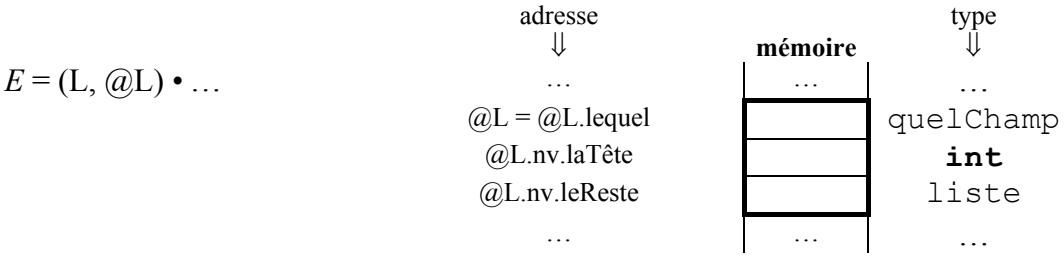
Avant exécution, l'environnement ressemble à :



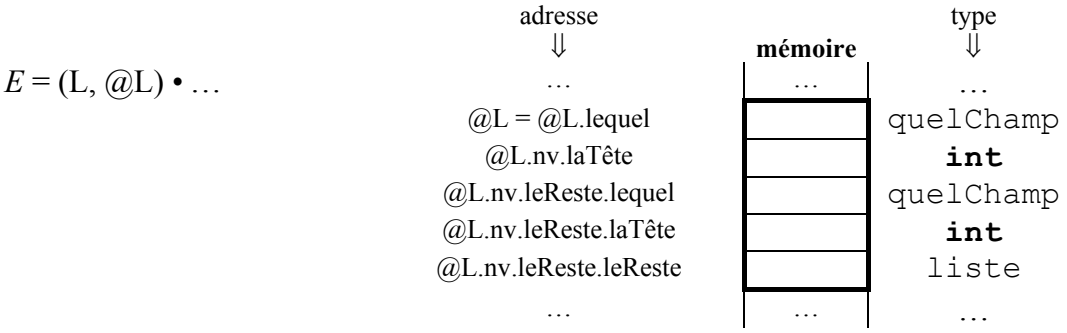
Après exécution, l'environnement est censé se transformer ainsi :



Comme nous l'avions noté dans notre section sur la construction union, le second membre de `L` conduit à allouer l'espace nécessaire au stockage de la valeur la plus consommatrice de mémoire. Ce membre est soit un équivalent entier (membre `v`, d'un type énuméré, ne consommant qu'une case mémoire), soit un chaînon (membre `nv`) ; c'est bien entendu ce dernier membre qui est le plus consommateur de mémoire : un chaînon est formé d'un entier (la tête) et d'une liste (le reste), c'est à dire pour l'allocation en cases mémoire :



Or ce dernier membre `L.nv.leReste` est lui-même une liste, auquel il va bien falloir réserver de l'espace mémoire. Le raisonnement précédent nous conduit donc à allouer cet espace :



Or ce dernier membre `L.nv.leReste.leReste` est lui-même une liste, … Le principe d'allocation des membres d'une union nous conduit à réserver une infinité de cases mémoire, ce qui pose un sérieux problème !

13.2. LES TYPES RECURSIFS EN C++

Le problème d'allocation évoqué dans la section précédente est applicable à tout type récursif en C++. Si *T* est un type récursif, le seul moyen offert par C++ pour contourner cette difficulté consiste à transformer tout membre de type *T* en un membre de type *T**. Le problème d'allocation infinie est résolu, puisqu'un pointeur, quelque soit le type visé, ne consomme qu'une seule case mémoire.

Ce passage obligatoire par les pointeurs nous permet de réaliser une économie ; en effet, il existe un pointeur particulier, le **pointeur nul** (sa valeur correspond en général à l'adresse 0), qui est garanti comme ne pointant jamais sur une quelconque valeur. Cette valeur particulière des pointeurs peut être exploitée pour représenter l'une des valeurs singulières (son type est non récursif) d'un type récursif.

Appliqué aux listes, le discours précédent nous permet d'écrire :

```
struct unChainon ;
typedef unChainon* liste ;

struct unChainon {
    int laTête ;
    liste leReste ;
} ;
```

Une liste est donc simplement un pointeur sur un chaînon. Si ce pointeur est nul, nous l'interprétons comme la liste vide. S'il est non nul, c'est donc que le pointeur pointe sur un chaînon, donc que la liste possède bien une tête et un reste, lequel est une liste, vide ou pas.

Les définitions ci-dessus nous amènent à réviser celles des fonctions de base sur les listes :

```
liste vide() { return 0 ; } ;

bool vide(liste L) { return L == 0 ; }
```

```
int tête(liste L) {
    if (vide(L)) return -1 ; // traitement arbitraire de l'erreur
    return L->laTête ;
} ;

liste reste(liste L) {
    if (vide(L)) return L ; // traitement arbitraire de l'erreur
    return L->leReste ;
} ;

liste ajout(int e, liste L) {
    unChainon R = { e, L };
    return &R ;
} ;
```

La définition de la fonction `ajout` pose un problème crucial : le résultat est un pointeur sur une variable `R` qui sera effacée de l'environnement (donc de la mémoire) une fois l'appel à `ajout` terminé. Le résultat sera donc un pointeur caduque : il pointera sur une case mémoire dans laquelle il n'y a plus aucune valeur… Cette situation était prévisible : l'allocation du nouveau chaînon est obtenue en définissant une variable du type idoine. Cette définition est valable jusqu'à ce que le bloc qui la portait se ferme. Sa fermeture provoque l'effacement de toutes les variables qu'il a lui-même introduites dans l'environnement, rendant caduque tout pointeur référençant l'une d'elles.

C'est la seconde difficulté introduite par les types récursifs ; pour résoudre ce problème de caducité des valeurs retournées, nous devrons nous appuyer sur un mécanisme capable de nous allouer de l'espace mémoire sans que celui ne soit perdu lorsque le bloc qui aura provoqué cette allocation se fermera. Ce mécanisme est l'**allocation dynamique**.

13.3. ALLOCATION DYNAMIQUE

L'allocation dynamique est un mécanisme qui permet d'allouer des cases mémoires au besoin, sans passer par des instructions de définition de variables. Le mécanisme d'allocation est indépendant du mécanisme qui gère les fermetures de blocs : toutes les allocations demandées par des définitions de variables dans un bloc sont automatiquement récupérées lors de la fermeture du bloc. Ce mécanisme procède à une **allocation** dite **automatique**.

Dans l'allocation dynamique, les demandes d'allocation sont provoquées par l'exécution d'un opérateur spécial : **`new`**. Le mécanisme d'allocation automatique ne s'occupe pas de récupérer les espaces alloués dynamiquement lorsque des blocs se ferment. Puisque c'est le programmeur qui décide du moment des allocations dynamiques, ce sera aussi à lui de savoir quand récupérer les espaces mémoire correspondants (nous parlerons de **désallocation**) ; c'est ce qui motive une seconde instruction spéciale : **`delete`**.

13.3.1. Opérateur d'allocation

Soit *T* un type. Toute demande d'allocation pour mémoriser une valeur de type *T* s'écrira :

```
new T
```

Cette expression retourne un pointeur sur l'espace nouvellement alloué. Si la demande d'allocation consiste à réserver de l'espace pour mémoriser un tableau de *n* valeurs de type *T*, alors cette écriture devient :

```
new T[n]
```

où *n* est une expression quelconque devant retourner un entier naturel. Le résultat est un pointeur sur la première composante d'un tableau de *n* composantes de type *T*.

Exemple : examinons les conséquences sur l'environnement de l'exécution en séquence des instructions suivantes :

```
int* p ;
{
    int a = 3;
    p = &a ;        // point [1]
    p = new int ;   // point [2]
} ;
*p = 4 ;            // point [3]
p = new int[2] ;
p[0] = 5 ;
p[1] = 6 ;          // point [4]
```

Au point `[1]`, l'environnement est défini par :

E = (a, @a) • ✸ • (p, @p) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@a	3	**int**
…		…
@p	@a	**int***
…	…	…

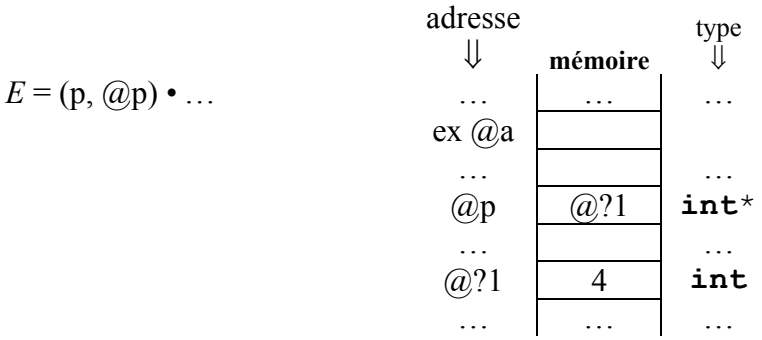
Au point `[2]`, l'environnement devient :

E = (a, @a) • ✸ • (p, @p) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@a	3	**int**
…		…
@p	@?1	**int***
…		…
@?1	?	**int**
…	…	…

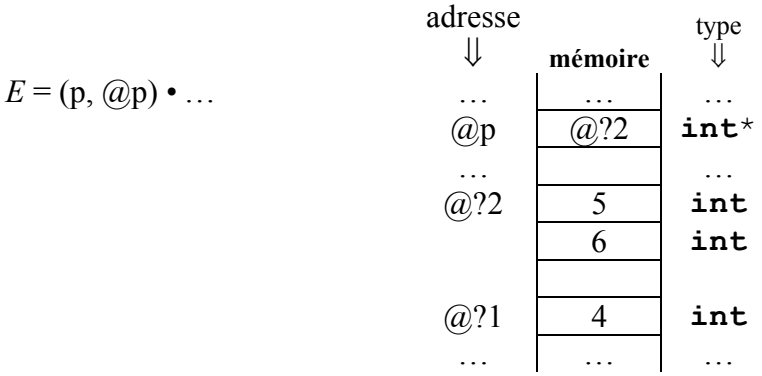
La demande d'allocation dynamique a en effet consisté à trouver une case mémoire pas encore exploitée permettant de coder une valeur entière (son adresse est @?1 sur le schéma, le ? indiquant simplement par convention qu'aucun nom de variable ne lui est associé, le numéro permettant de la distinguer des autres allocations dynamiques) ; la valeur entière associée est indéfinie.

Au point `[3]`, l'environnement devient :



La fermeture du bloc a provoqué l'effacement de la variable `a`, et donc une récupération automatique de l'espace mémoire associé ; sur le schéma, la case ex @`a` qui contenait la valeur de `a` est désormais réutilisable. L'affectation qui suit permet de modifier la valeur de l'entier alloué dynamiquement.

Au point `[4]`, l'environnement devient :



Le pointeur `p` est redirigé sur une autre zone d'entiers allouée dynamiquement (ici un tableau). Les affectations permettent de modifier les composantes du tableau associé. Noter toutefois que la case d'adresse @?1 n'a pas été récupérée, et reste donc marquée comme toujours exploitée. Rappelons-nous en effet que, dans le cas des allocations dynamiques, nous devons écrire explicitement des instructions pour récupérer ces espaces alloués dynamiquement, dès lors que nous sommes sûrs qu'ils ne seront plus effectivement exploités ; c'est l'objet de la section qui suit.

13.3.2. Opérateur de désallocation

Soient *T* un type et *p* un pointeur de type *T** ; la désallocation d'un espace mémoire alloué dynamiquement est provoquée ainsi :

- si *p* pointe sur un espace alloué dynamiquement par un **`new`** *T*, alors la valeur pointée sera désallouée (ou récupérée) par l'instruction :

  `delete` *`p ;`*
- si *p* pointe sur un espace alloué dynamiquement par un **`new`** *`T[n]`*, alors la valeur pointée sera désallouée par :

  `delete` `[]` *`p ;`*

Exemple : dans l'exemple précédent, si l'on souhaite libérer après usage les espaces mémoires alloués dynamiquement, la séquence d'instructions à écrire serait :

```
int* p ;
{
    int a = 3;
    p = &a ;          // point [1]
    p = new int ;     // point [2]
} ;
*p = 4 ;              // point [3]
delete p ;
p = new int[2] ;
p[0] = 5 ;
p[1] = 6 ;            // point [4]
delete [] p ;         // point [5]
```

Au point `[4]`, l'environnement deviendrait :

E = (p, @p) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@p	@?2	**int***
…		…
@?2	5	**int**
	6	**int**
ex @?1		
…	…	…

Noter que la première opération **`delete`** a bien libéré la case mémoire d'adresse @?1, permettant de la réutiliser.

Au point `[5]`, `p` pointe sur une zone qui ne contient plus rien : c'est donc une erreur que de continuer à exploiter la valeur pointée après ce point.

E = (p, @p) • …

adresse ⇓	mémoire	type ⇓
…	…	…
@p	@?2	**int***
…		…
@?2		
…	…	…

13.3.3. Remarques

Ordonner la désallocation d'une variable allouée automatiquement provoque des effets non prévisibles (de *aucun effet* au *blocage de l'ordinateur*).

Exemple : une séquence d'instructions fortement déconseillée

```
int a ; delete &a ;
```

Désallouer un tableau alloué dynamiquement par une instruction **`delete`** plutôt que **`delete`** `[]` ne provoque en général la libération que de la première composante du tableau ; les autres restent considérées comme toujours exploitées…

Exemple : une séquence d'instructions qui laissera 1000 cases mémoires orphelines, perdues pour toute exploitation ultérieure, puisque considérées comme étant toujours utilisées.

```
int* p = new int[1001] ; delete p ;
```

Lorsqu'un programme s'exécute, il dispose d'un espace mémoire découpé en deux parties :

- une première partie, appelée **pile d'exécution** (*stack* en anglais), sert aux allocations automatiques, contrôlées par les instructions de définitions de variables et les fermetures de blocs. Les variables globales (celles qui ne seront détruites qu'en fin de programme) peuvent être stockées à part (zone des **allocations statiques**).
- une seconde partie, appelée **tas** (*heap* en anglais), sert aux allocations dynamiques, contrôlées par l'exécution de **`new`** et **`delete`**.

13.4. RETOUR SUR LA REALISATION DES LISTES

Nos développements sur l'allocation dynamique se justifiaient par les problèmes rencontrés pour définir la fonction `ajout` du type liste :

```
liste ajout(int e, liste L) {
    unChainon R = { e, L }; return &R ; } ;
```

La suite nous a démontré l'incorrection de cette définition : le nouveau chaînon à créer est alloué automatiquement. Il doit en fait être alloué dynamiquement :

```
liste ajout(int e, liste L) {
    unChainon* R = new unChainon ;
    R->laTête = e ; R->leReste = L;
    return R ;
} ;
```

Le chaînon ainsi alloué survivra à la fermeture du bloc correspondant à l'appel de la fonction. Néanmoins, il nous faut fournir une fonction qui, si nous sommes certains que tous les chaînons d'une liste ne sont plus exploités, nous permette de les désallouer :

```
void libérer(liste L) {
    if (vide(L)) return ;
    libérer(reste(L)) ; // libère d'abord les chaînons du reste…
    delete L ;          // …puis libère le chaînon courant
} ;
```

Tout programme manipulant des listes devra donc appeler notre fonction `libérer` sur toute liste qui sera jugée obsolète, nous permettant de réutiliser l'espace mémoire que ses chaînons consommaient.

Cette première version de réalisation des listes et des fonctions associées est pratiquement achevée. Elle pose néanmoins d'autres problèmes, que nous traiterons dans une prochaine section (problème de listes pouvant partager des chaînons : si l'une de ces listes est libérée, les autres listes référenceront des chaînons qui n'existent plus…).

14. LA GENERICITE

La généricité est un puissant outil d'abstraction. Le concept de fonction permet de paramétrer un bloc d'instructions, via ces variables spéciales que sont ses paramètres. Tout appel de fonction précise quelles valeurs donner à ces paramètres avant d'exécuter le bloc.

Le paramétrage des fonctions énoncé ci-dessus montre que les seuls éléments qui peuvent changer d'un appel à l'autre pour une même fonction sont les valeurs liées aux paramètres de la fonction. La généricité nous permet désormais de paramétrer aussi le type des paramètres.

14.1. MOTIVATION

Pour illustrer le besoin de paramétrer les types des paramètres, prenons la fonction `min`, capable de retourner la plus petite de deux valeurs d'un même type. En C++, si nous souhaitons définir cette fonction pour les entiers standard (**`int`**), nous devons écrire :

```
int min(int a, int b) {
    if (a<b) return a ; else return b ; }
```

Si nous voulons en plus que cette fonction opère sur les nombres flottants (**`float`**), nous devons ajouter une nouvelle définition :

```
float min(float a, float b) {
    if (a<b) return a ; else return b ; }
```

Le nom `min` est surchargé (deux fonctions portent le même nom), et le compilateur se sert du type des expressions pour déterminer quel code exécuter, si nécessaire en opérant des conversions de type.

Exemple : examinons la séquence d'instructions ci-dessous :

```
int a=1, b=2, c ;
float x=3.0, y=4.0, z ;
c = min(a, b) ; // sélection de int min(int, int)
z = min(x, y) ; // sélection de float min(float, float)
z = min(a, y) ; // sélection de float min(float, float)
                //      car conversion de a en float
```

De façon générale, nous devons repérer dans un programme tous les appels à la fonction `min`, noter à chaque fois dans quel contexte l'appel est fait (type des arguments de l'appel), et finalement écrire autant de définitions qu'il y a de contextes d'appel à cette fonction ; chaque définition correspondant à un couple de valeurs d'un même type *T*. Le problème pour le concepteur est qu'il risque de se retrouver avec plusieurs fois le même code à écrire, au type des paramètres près ; pour notre fonction `min`, si *T* est un type, il y a de bonnes chances pour que le code pour ce type soit du style :

```
T min(T a, T b) { if (a<b) return a ; else return b ; }
```

Ce code doit être malheureusement répété autant de fois qu'il y a de types *T* distincts.

L'idée est de se dire : quelque soit le type *T*, la définition de notre fonction sera toujours la même ; pourquoi ne pas permettre de rendre variable aussi le type *T* ? Cette idée se traduit en C++ par la notion de **variable de type**, i.e. une variable qui sera liée non pas à une valeur mais à un type. Pour notre fonction `min`, la définition s'écrira alors :

```
template <class T>          // T est une variable de type
T min(T a, T b) {           // a et b sont des paramètres classiques
    if (a<b) return a ; else return b ; }
```

L'intérêt pour le concepteur est qu'il n'a plus qu'une seule définition à écrire (celle ci-dessus), qui plus est valable pour n'importe quel type *T* de son programme. Tout le travail du compilateur consiste alors à déterminer, pour chaque appel à la fonction `min`, à quel type il faut lier *T* pour définir quel code exécuter.

Ces paramètres d'un nouveau genre sont aussi appelés **paramètres de généricité**, et la définition est dite **générique**. Dans certains langages, en particulier la plupart des langages fonctionnels typés, une définition dont les paramètres de généricité sont tous des variables de type est dite **polymorphe**, et l'on évoque son **polymorphisme de type.**

Lorsqu'une variable classique est initialisée (donc liée) avec une valeur, nous parlons d'initialisation. Lorsqu'une variable de type est liée à un type, nous parlerons d'**instanciation** : la variable de type sera dite **instanciée** au type.

La généricité ne s'applique pas seulement aux fonctions ; elle s'applique aussi à d'autres constructions de type C++ : les structures et les unions.

14.2. GENERICITE DES FONCTIONS

14.2.1. Déclaration / définition d'une fonction générique

Soit *f* une fonction ayant *n* paramètres classiques, chacun d'un type T_i. Cette fonction sera générique si elle introduit en plus *k* paramètres de généricité, sa déclaration s'écrivant alors :

```
template <G1, G2, …, Gk> R f(P1, P2, …, Pn) ;
```

et sa définition :

```
template <G1, G2, …, Gk> R f(P1, P2, …, Pn) corps ;
```

où *corps* est le bloc associé à la fonction. L'écriture des paramètres P_i a déjà été étudiée (cf. la section sur les fonctions). Chacun des paramètres de généricité G_i est de la forme :

class n_i : n_i est le nom d'une variable de type

T_i n_i : n_i est le nom d'un paramètre d'un type existant T_i

Exemple : la fonction générique `max` calculant la plus grande de deux valeurs d'un même type :

```
template <class T>
T max(T a, T b) { if (a>b) return a ; else return b ; }
```

Exemple : la fonction générique `echange` capable d'échanger le contenu de deux variables d'un même type `T` :

```
template <class T>
void echange(T& a, T& b) { T tmp = a ; a = b ; b = tmp ; }
```

Exemple : la fonction générique `tracer` capable d'appliquer une fonction à une valeur tout en laissant une trace à l'écran après l'appel afin d'afficher le résultat :

```
template <class A, class B>
B tracer(B (*f)(A), A x, char* g="=>", char* d="\n") {
    B resultat = (*f)(x);
    cout << g << resultat << d;
    return resultat ;
}
```

Les noms des paramètres de généricité qui sont des variables de type doivent figurer obligatoirement dans la signature (paramètres classiques + type du résultat) de la fonction. Comme pour les paramètres classiques, il est possible d'associer à un paramètre de généricité une valeur par défaut ; dès lors, tous les paramètres de généricité figurant à sa droite doivent en définir une aussi.

Exemple : une définition incorrecte de fonction générique :

```
template <class T>
void fonc(int a) {   // le nom T doit figurer dans la signature
    T tmp ; cin >> tmp ;
    if (a%2==0) cout << tmp; else cout << -tmp;
}
```

La spécification des paramètres de généricité (après le mot-clé **`template`**) ne porte que sur la déclaration ou définition qui suit immédiatement.

Exemple : si les fonctions génériques `min` et `max` doivent être définies, écrire :

```
template <class T>
T min(T a, T b) { if (a<b) return a ; else return b ; }

template <class T>     // spécification de généricité à répéter
T max(T a, T b) { if (a>b) return a ; else return b ; }
```

14.2.2. Appel d'une fonction générique

L'appel à une fonction générique s'écrit de la même façon qu'un appel à une fonction classique : il n'y a rien à préciser quand à l'instanciation des paramètres de généricité (le compilateur règle la question automatiquement, si possible). Aucune conversion de type n'est appliquée aux arguments d'un appel à une fonction générique : la signature de l'appel doit impérativement pouvoir s'apparier telle quelle à la signature de la fonction appelée.

Exemple : variantes d'appels sur la fonction `min` :

```
template <class T>
T min(T a, T b) { if (a<b) return a ; else return b ; };
```

```
main() {
    int a=1, b=2 ;
    float x=3.0, y=4.0 ;
    cout << min(a,b)     // correct ; T instancié avec int
         << min(x,y)     // correct ; T instancié avec float
         << min('a','b') // correct ; T instancié avec char
         << min(a,y)     // incorrect ; aucune instanciation de T ne
         << endl;        //   permet de définir min(int,float)
};
```

Il existe des situations où le compilateur ne réussit pas seul à trouver une bonne instanciation ; dans ce cas, il est possible de forcer l'instanciation, en faisant suivre immédiatement, dans l'appel, le nom de la fonction par la liste des instanciations souhaitées : encadrées de < et >, et séparées par des virgules.

Exemple : forçage d'une instanciation :

```
template <class T, unsigned int n>
unsigned int truc(T a, T b) { return a==b ? n : 1 ; };

main() {
    int a=1, b=2 ;
    cout << truc<int, 3>(a,b) << endl;
};
```

L'appel force le compilateur à instancier `T` avec **`int`** et `n` avec 3.

14.3. GENERICITE DES TYPES

Tous les développements sur la généricité présentés pour les fonctions s'appliquent aux constructions de type structures et unions. Les types génériques obtenus sont aussi appelés **types paramétrés**.

Dans cette section, nous ne développons que la généricité des structures. Il suffit de remplacer le mot *structure* par *union* pour obtenir le discours équivalent sur la généricité des unions.

14.3.1. Déclaration et définition d'un type générique

La déclaration d'un type structure générique T s'écrit :

`template` <G_1, G_2, …, G_k> **`struct`** *T*;

où les G_i sont les paramètres de généricité, chacun de la forme :

`class` n_i : n_i est le nom d'une variable de type

T_i n_i : n_i est le nom d'un paramètre d'un type existant T_i

Comme les fonctions génériques, un paramètre de généricité peut avoir une valeur par défaut, laquelle contraint tous les paramètres à sa droite d'en définir une eux aussi. Comme toute déclaration, une telle clause peut apparaître plusieurs fois dans un même texte C++. La spécification de généricité (**`template`**) ne porte que sur la déclaration qui suit immédiatement.

Exemple : déclaration d'un produit, appelé paire, de deux types quelconques A et B :

```
template <class A, class B > struct unePaire;
```

La définition d'une structure générique ajoute le détail des membres qui la caractérisent :

```
template <G1, G2, …, Gk> struct T { T1 m1; … ; Tn mn; } ;
```

Exemple : définition d'une paire, produit de deux types quelconques A et B :

```
template <class A, class B >
struct unePaire { A premier; B second; };
```

Comme les fonctions, tous les paramètres de généricité d'un type générique doivent obligatoirement figurer dans sa définition.

14.3.2. Usage d'un nom de type générique

A l'inverse des fonctions, où l'appel d'une fonction générique ne devait pas nécessairement préciser l'instanciation des paramètres de généricité, l'usage d'un nom d'un type générique impose de préciser explicitement comment doivent être instanciés ses paramètres de généricité.

Si *T* est un type générique ayant *k* paramètres de généricité, toute exploitation du nom de ce type pour définir ou déclarer des entités (paramètres de fonctions, variables, constantes, membres données de structures ou d'unions, etc.) devra s'écrire ainsi :

```
T<e1, e2, …, ek>
```

où chaque e_i est une expression permettant de définir ce qui sera lié au paramètre de généricité correspondant. Si certains paramètres de généricité disposent de valeurs par défaut, il est bien entendu possible de spécifier moins d'arguments dans l'écriture ci-dessus, que le compilateur s'empressera de compléter avec les valeurs par défaut.

Exemple : un petit programme exploitant des paires d'éléments quelconques :

```
template <class A, class B >
struct unePaire { A premier; B second; };

template <class A, class B >
unePaire<B, A> retourner(unePaire<A, B> p) {
    // la fonction retourner prend en entrée une paire (x, y)
    // et retourne une paire (y, x)
    unePaire<B, A> r = { p.second, p.premier} ;
    return r ;
} ;

template <class A, class B >
void afficher(unePaire<A, B> p) {
    cout << '(' << p.premier << ',' << p.second << ")\n";
} ;

main() {
    unePaire<int, char> p1 = { 12, '?' };
    unePaire<double, int> p2 = { 1.0, 2 } ;
    afficher(p1) ;
    afficher(retourner(p1)) ;
    afficher(p2) ; }
```

Ce code montre que tout usage du nom du type `unePaire` précise l'instanciation des paramètres de généricité. Noter que cette instanciation peut-être réalisée avec des variables de type (cas des deux fonctions `retourner` et `afficher`).

Petit détail syntaxique : si l'instanciation d'une variable de type nécessite un type générique, il faut séparer les deux caractères > consécutifs par un espace (si ces deux caractères sont accolés, le compilateur considère qu'il s'agit de l'opérateur >>, utilisé notamment pour la saisie via `cin`).

Exemple : définition d'une paire dont la première composante est un **`char`** et la seconde une paire de (**`float`**, **`int`**) :

```
unePaire<char, unePaire<float, int> > p ;
          // noter l'espace séparant les deux > finaux
p.premier = 'a' ;
p.second.premier = 2.0 ;
p.second.second = 1 ;
```

15. LA GESTION D'ERREUR : LES EXCEPTIONS

15.1. MOTIVATION

Nous avons résolu quelques exercices dans lesquels nous étions amenés à gérer des situations anormales ; le code traitant ce type de situation était toutefois soit inexistant, soit très insuffisant.

Exemple : dans notre étude des listes, nous avions par exemple défini la fonction `reste` ainsi, sachant que cette fonction pose un problème si la liste à traiter est vide :

```
liste reste(liste L) {
    if (vide(L))
        return L ; // traitement arbitraire de l'erreur
    else
        return L->leReste ;
} ;
```

Cette fonction `reste` retourne un résultat de type `liste` ; quelle que soit la liste en entrée, la fonction doit toujours retourner un résultat… ce qui pose problème quand la fonction n'est pas définie pour certaines listes. En C++, nous sommes obligés de produire un résultat, même dans les cas indéfinis : une vraie quadrature du cercle ! La technique adoptée consiste ici à modifier la définition originale de la fonction `reste` afin qu'elle soit définie pour toute liste : le reste d'une liste vide est désormais une liste vide.

Exemple : soit à définir une fonction devant calculer le quotient de la division entière de deux entiers naturels *a* et *b* (positifs ou nuls). Le résultat est un entier naturel, mais n'est pas défini si *b* est nul.

```
unsigned int quotient(unsigned int a, unsigned int b) {
    if (b==0)
        … // quoi faire ici ?
    else
        return a/b ;
} ;
```

Une solution consiste à modifier le type du résultat pour retourner une valeur spéciale facilement reconnaissable dans les cas d'erreurs :

```
int quotient(unsigned int a, unsigned int b) {
    if (b==0)
        return -1 ;
    else
        return a/b ;
} ;
```

Il est facile de vérifier désormais que l'appel à la fonction s'est bien passé : si `quotient`(*a*, *b*) retourne un entier positif ou nul, c'est que le quotient est défini, et vaut justement cet entier. Si par contre le résultat vaut –1, c'est que le quotient est indéfini.

Tout concepteur de programme n'a pas le droit de faire l'hypothèse que chaque appel à l'une de ses fonctions correspond aux seuls cas où la fonction est définie : il se doit de prévoir tous les cas possibles, y compris donc ceux qui posent problème.

En l'absence de mécanisme propre de gestion des erreurs, les solutions les plus communément adoptées pour traiter des erreurs survenant dans une fonction se résument à :

- transformer la définition originale de la fonction (indéfinie pour quelques cas) en une définition où tous les cas conduisent à un résultat (stratégie adoptée dans le premier exemple évoqué ci-dessus)
- retourner une valeur remarquable (de même type que n'importe quel résultat) que l'appelant de la fonction pourra reconnaître et interpréter comme une erreur
- passer un paramètre de plus à la fonction qui recevra un code d'erreur.

La plupart de ces solutions obligent l'appelant de la fonction à faire un test systématiquement et explicitement avant (pour éviter d'appeler la fonction dans un cas qu'il sait problématique) ou après (pour vérifier que le résultat est significatif) chaque appel, ce qui alourdit le code de façon considérable. C'est pourquoi les concepteurs de C++ ont mis en place un mécanisme de gestion d'erreur décomposé en deux parties :

- Lorsqu'une fonction appelée rencontre un problème, elle déclenche une erreur (techniquement, nous dirons qu'elle **lève une exception**) ⇒ instruction **`throw`**.
- Lorsqu'une fonction (appelante) appelle une autre fonction (appelée), et que cette dernière peut potentiellement lever une exception, la fonction appelante peut mettre en place un **piège** (ou **bloc de capture**) à exception, lequel se traduira par du code à exécuter si une exception particulière était levée. Intérêt : aucun test d'erreur explicite n'est écrit ⇒ instruction **`try`**.

15.2. INSTRUCTION DE LEVEE D'UNE EXCEPTION

Si une portion de code est telle qu'un cas anormal est détecté, il suffit d'exécuter l'instruction :

```
throw e ;
```

où *e* est une expression ayant une valeur d'un type *T* quelconque portant toute l'information nécessaire pour corriger éventuellement ou commenter l'erreur.

15.3. INSTRUCTION DE CAPTURE D'EXCEPTIONS

Lorsqu'un bloc d'instructions *bloc* est susceptible de déclencher une exception, et que son concepteur souhaite explicitement gérer ces erreurs potentielles, il encapsulera ce bloc dans un piège à exceptions via l'instruction suivante :

```
try
    bloc
catch (T1 v1) bloc1
catch (T2 v2) bloc2
…
catch (Tn vn) blocn
catch (...) blocn+1 ;
```

où chaque bloc (*bloc* ou $bloc_i$) est encadré d'accolades ({ et }). Le bloc principal *bloc* est dit **piégé** (c'est son exécution qui déclenchera potentiellement une exception). Chaque clause :

```
catch (Ti vi) bloci
```

est dite **clause de capture** de l'exception T_i ; chaque $bloc_i$ est appelé **bloc de traitement de l'exception** T_i. La dernière clause spéciale :

```
catch (...) blocn+1
```

est optionnelle, toujours placée à la fin de la liste des clauses de capture, et appelée **clause de capture par défaut**. Son bloc associé est dit **bloc de traitement par défaut d'exception**.

Chaque bloc de traitement d'exception $bloc_i$ peut lui-même contenir l'instruction de levée d'exception **`throw`**, sous deux formes différentes :

```
throw e ;
```

où *e* est une expression d'un type *T* quelconque , ou encore :

```
throw ;
```

La section qui suit définit la sémantique des instructions liées à la gestion d'exception.

15.4. SEMANTIQUE DE LA GESTION DES EXCEPTIONS

Supposons que nous devions examiner les effets de l'exécution de l'instruction **`try`** suivante, l'environnement courant étant *E* :

```
{ // bloc englobant
    …
    try
        bloc  // bloc piégé
    catch (T1 v1) bloc1
    catch (T2 v2) bloc2
    …
    catch (Tn vn) blocn
    catch (...) blocn+1 ;
    iaprès ;
    …
}
```

Exécuter cette instruction, c'est exécuter son bloc piégé *bloc*. Deux situations se présentent :

- soit aucune exception n'est levée au cours de l'exécution des instructions qu'il contient, et l'exécution du bloc englobant se poursuit normalement avec l'instruction $i_{après}$ et suivantes.

- soit une exception est levée via l'exécution d'une instruction :

```
throw e ;
```

Pour cette dernière situation, où *e* est une expression de type *T* ayant une valeur *v*, les conséquences sont les suivantes :

a) l'exécution du bloc piégé est abandonnée ; il est donc refermé

b) chaque clause de capture est examinée, dans l'ordre où elles ont été définies.

Plusieurs alternatives sont possibles :

- soit une clause de capture piège une exception du même type *T* que celui de celle levée, i.e. il existe une clause :

```
catch (Ti vi) blocᵢ
```

  telle que $T_i = T$; dans ce cas, l'exécution de **`try`** se poursuit en exécutant le bloc de traitement d'exception $bloc_i$, dans l'environnement courant *E* enrichi de la variable de nom v_i, de type *T* et ayant la valeur *v*.

  A nouveau deux situations se présentent :

  - ♦ soit l'exécution du bloc de traitement $bloc_i$ ne lève aucune exception, alors l'instruction **`try`** se termine, l'exécution du bloc englobant se poursuivant normalement avec l'instruction $i_{après}$ et suivantes.
  - ♦ soit une exception est levée (exécution d'un **`throw`**), et dans ce cas… passer au paragraphe décrivant cette situation (levée d'exception dans un bloc de traitement d'exception), ci-dessous.

- soit aucune clause de capture ne piège une exception de type *T*, et la clause de capture par défaut est définie, i.e. il existe :

```
catch (...) blocₙ₊₁
```

  alors l'exécution de **`try`** se poursuit en exécutant le bloc de traitement d'exception $bloc_{n+1}$, dans l'environnement courant *E*, avec les deux mêmes issues possibles que celles signalées dans les clauses de capture classiques (cf. ci-dessus).

- soit aucune clause de capture ne piège une exception de type *T*, et la clause de capture par défaut n'existe pas : dans ce cas, l'exception est levé à nouveau, mais dans le bloc englobant cette fois. Autrement dit, si aucune clause de capture par défaut n'est définie, le compilateur considère qu'elle existe, avec la définition suivante :

```
catch (...) { throw ; }
```

Il nous reste à décrire la situation où l'exécution d'un bloc de traitement d'exception $bloc_i$ lève elle-même une exception. Cette situation est provoquée par une instruction **`throw`**, qui peut se présenter sous deux formes :

```
throw e ;
```

ou

```
throw ;
```

Dans ce dernier cas, le compilateur relève l'exception en cours de traitement. Dans les deux cas, l'effet est identique : c'est le bloc englobant qui, cette fois, est abandonné, l'exception pouvant être capturée si ce bloc englobant faisait lui-même partie d'un bloc piégé.

Si aucun piège à exceptions n'est posé par le concepteur, le compilateur défini un traitement d'erreur par défaut qui se traduit, en général, par un message à l'écran signalant que l'exception levée n'a pas été capturée, suivi de l'arrêt du programme.

15.4.1. Un exemple

Soit le programme suivant :

```
int main() {
    int x = 1, n ;
    cout << "n = " ; cin >> n ;
    try {  // bloc piégé appelé b1
        x = x+n ;
        if (n>0) {
            x = x+n ;
            try { // bloc piégé appelé b2
                if (n==2) throw -1 ;
                if (n==3) throw 1.0F ;
            }
            catch (int e) {
                cout << "b2.int : " << e << endl ;
                throw '?' ;}
            catch (char e) {
                cout << "b2.char : " << e << endl ;
                throw ; // équivalent ici à throw e
            } ;
            x = x+n ;
            if (n==1) throw n+2 ;
        }
    }
    catch (int e) {
        cout << "b1.int : " << e << endl ; }
    catch (float e) {
        cout << "b1.float : " << e << endl ; }
    catch (char e) {
        cout << "b1.char : " << e << endl ; } ;
    cout << "x = " << x << endl ;
}
```

Il s'agit de deviner la liste des messages affichés à l'écran lorsque l'utilisateur répond successivement à la question posée par 0, 1, 2, 3 ou 4.

- 1er cas : `n = 0`

  Le bloc piégé `b1` est exécuté sans lever d'exception. L'ajout de `n` à `x` opère, le bloc du **`if`** est ignoré (le test `n>0` est faux) ; le bloc piégé se termine, puis suit l'affichage :

  ```
  x = 1
  ```

- 2ème cas : `n = 1`

  Le bloc piégé `b1` lève une exception. L'ajout de `n` à `x` opère, le test `n>0` du **`if`** est vrai : exécution du bloc qui suit. A nouveau ajout de `n` à `x`. Exécution du bloc piégé `b2`, lequel ne lève aucune exception (exécution normale) : nous poursuivons donc l'exécution après le **`try`** de `b2`. Nouvel ajout de `n` à `x`. Le test `n==1` étant vérifié, levée explicite d'une exception avec une expression de type **`int`**, de valeur `n+2`, soit 3.

  Le bloc en cours d'exécution étant `b1`, les exceptions sont piégées. Recherche de la première clause de capture du genre **`catch (int …)`** : il en existe une (la première). Exécution du traitement d'erreur associé, avec définition temporaire d'une variable **`int e = 3`**. Ce bloc provoque l'affichage de :

  ```
  b1.int : 3
  ```

  L'exécution du traitement d'erreur se termine normalement (pas d'exception levée) : nous poursuivons l'exécution après le **`try`** du bloc `b1` ; suit donc l'affichage :

  ```
  x = 4
  ```

- 3ème cas : `n = 2`

  Exécution du bloc piégé `b1` : l'ajout de `n` à `x` opère, le test `n>0` du **`if`** est vrai : exécution du bloc qui suit. A nouveau ajout de `n` à `x`. Exécution du bloc piégé `b2`, lequel lève une exception avec la valeur –1 de type **`int`** : abandon du bloc `b2` et recherche d'une clause de capture du genre **`catch (int …)`** au niveau du piège à exceptions associé à `b2` ; nous en trouvons une (première clause) : exécution du traitement d'erreur correspondant ; tout d'abord affichage de :

  ```
  b2.int : -1
  ```

  Puis levée d'exception avec une valeur **`'?'`** de type **`char`**. Le traitement d'erreur associé à `b2` est donc abandonné. Le bloc englobant piégé est `b1` : abandon de l'exécution de `b1`, et recherche dans ses clauses de capture d'un **`catch (char …)`** ; la clause est trouvée : exécution du traitement d'erreur associé, lequel affiche :

  ```
  b1.char : '?'
  ```

  L'exécution du traitement d'erreur se termine normalement (pas d'exception levée) : nous poursuivons l'exécution après le **`try`** du bloc `b1` ; suit donc l'affichage :

  ```
  x = 5
  ```

- 4ème cas : `n = 3`

  Exécution du bloc piégé `b1` : l'ajout de `n` à `x` opère, le test `n>0` du **`if`** est vrai : exécution du bloc qui suit. A nouveau ajout de `n` à `x`. Exécution du bloc piégé `b2`, lequel lève une exception avec la valeur 1.0F de type **`float`** : abandon du bloc `b2` et recherche d'une clause de capture du genre **`catch (float …)`** au niveau du piège à exceptions associé à `b2` ; nous n'en trouvons pas ; le bloc piégé `b2` est donc définitivement abandonné, et

l'exception est re-levée dans le bloc englobant, soit `b1`. A nouveau recherche dans ses clauses de capture d'un **`catch`** (**`float`** …) ; la clause est trouvée (seconde clause) : exécution du traitement d'erreur associé, lequel affiche :

```
b1.float : 1
```

L'exécution du traitement d'erreur se termine normalement (pas d'exception levée) : nous poursuivons l'exécution après le **`try`** du bloc `b1` ; suit donc l'affichage :

```
x = 7
```

- 5ème cas : `n` = 4

  Exécution du bloc piégé `b1` : l'ajout de `n` à `x` opère, le test `n>0` du **`if`** est vrai : exécution du bloc qui suit. A nouveau ajout de `n` à `x`. Exécution du bloc piégé `b2`, lequel ne lève aucune exception. Poursuite de l'exécution après le **`try`** de `b2` : à nouveau ajout de `n` à `x`. Le test `n==1` qui suit échoue : le bloc piégé `b1` se termine normalement ; suit donc l'affichage :

  ```
  x = 13
  ```

16. LES CLASSES

Le concept de classe permet au langage C++ d'être incorporé à la panoplie des outils de la programmation orientée objets (POO) ; il donne à C++ son statut de langage idéal pour le génie logiciel : efficacité du code, protection des données (encapsulation), garantie de leur usage dans des contextes autorisés (typage fort), réutilisation du code existant (généricité, héritage).

L'étude des classes en C++ se découpe classiquement en trois volets :

- les classes sans lien d'héritage
- les classes avec lien d'héritage simple
- les classes avec lien d'héritage multiple

Dans notre cours, les seuls aspects des classes que nous étudions relèvent du premier volet : les classes sans héritage. Les deux autres volets ne sont pas au programme…

16.1. PROGRAMMATION ORIENTEE OBJET ET C++

16.1.1. Correspondance terminologique C++ / POO

Bien que la programmation objet ait introduit une terminologie aujourd'hui largement diffusée, les concepteurs de C++ ont malheureusement choisi de transformer les termes de la POO pour les adapter au monde de la programmation en C. Pour résumer :

POO	C++	Signification en POO
classe	classe	une classe (sorte de type)
instance	objet	une valeur d'une classe
propriété	membre	un attribut ou une méthode
attribut	membre donnée	une valeur associée à une instance
méthode	fonction membre	une fonction applicable à une instance
message	accès au membre d'un objet	application d'une méthode ou lecture / écriture d'un attribut
héritage	héritage	mécanisme de réutilisation - redéfinition
super-classe	classe de base	La classe dont une autre hérite
sous-classe	classe dérivée	La classe qui hérite d'une autre

16.1.2. Les types abstraits de données

La notion de type abstrait de donnée est une vieille idée en informatique (apogée dans les années 1970), qui est certainement pour beaucoup dans l'émergence de la notion de classe des langages objets. Un type abstrait de données est une construction

qui précise comment les valeurs de ce type peuvent être exploitées sans rien indiquer de leur codage en machine.

Exemple : le type abstrait représentant une liste pourrait se définir ainsi (pseudo-langage) :

```
type : Liste<a>
    (où a est une variable de type. Il s'agit donc d'un type générique – ou
    polymorphique)
opérations :
      vide : Liste<a>
     ajout : a * Liste<a> -> Liste<a>
      tête : Liste<a> -> a
     reste : Liste<a> -> Liste<a>
 est_vide? : Liste<a> -> bool
axiomes :
  (1)        tête( ajout(e, L) ) = e
  (2)      reste( ajout(e, L) ) = L
  (3)         est_vide?( vide ) = true
  (4)  est_vide?( ajout(e, L) ) = false
erreurs :
  (1) tête( vide )
  (2) reste( vide )
```

La partie **opérations** indiquent toutes les opérations autorisées sur une liste (le type de chacune des opérations précise le contexte de leur usage). La partie **axiomes** définit la sémantique des opérations, autrement dit les propriétés qui restent invariantes. La partie **erreurs** précise des cas d'usage qui, bien que typés correctement, sont sémantiquement erronés.

Dans la définition du type abstrait, rien n'est indiqué sur l'implantation possible de ce type en machine ; mais toute l'information suffisante à son usage (opérations et axiomes) est donnée : c'est la notion de type abstrait de données.

En C++, un type abstrait se concrétise, idéalement, sous forme d'une classe.

16.1.3. La notion de classe en C++

Une classe C++ est une généralisation des constructeurs de type **struct** et **union** ; une classe permet de définir à la fois :

- les **membres données** de la structure sur laquelle elle s'appuie
- des types locaux à la classe, appelés **types membres**
- les fonctions qui manipulent ces membres ; ces fonctions sont appelées des **fonctions membres**.

16.2. DEFINITION - DECLARATION D'UNE CLASSE

La définition ou déclaration d'une classe suit les mêmes règles syntaxiques que la définition - déclaration d'une **struct**. Ainsi, déclarer une classe de nom `C` s'écrit simplement :

```
class C ;
```

La définition d'une classe `C` ressemble à celle d'une structure

```
class C {
    // définitions/déclarations de membres
} ;
```

avec toutefois les différences suivantes :

- aucun membre donnée ne peut être initialisé lors de sa déclaration.
- les membres peuvent être des fonctions.
- certains membres peuvent être des types (définis localement à la classe).
- chaque déclaration/définition de membre peut être précédée d'un **mode de protection d'accès** :

`public` : le membre est dit **public**.
`private` : le membre est **privé**.

Il existe un troisième mode de protection (**`protected`** - lié à l'héritage), non étudié dans ce cours.

Dès qu'une spécification de protection est présente, elle reste valable pour tous les membres de la classe qui suivent, jusqu'à ce qu'une prochaine spécification de protection la change. Par défaut, dès l'ouverture de la définition d'une classe, la protection des membres qui suivent est **`private`**.

Exemple : une définition de classe :

```
class Liste { // le membre qui suit est privé

    struct chainon { // déclaration d'un type local privé
        int element;
        chainon *suivant } ;

    chainon *dernier; // déclaration du membre donnée

public: // spécification d'un mode de protection : tous les membres qui
        //         suivent sont publics, sauf indication contraire
    Liste(); // une fonction membre particulière, appelée constructeur
    ~Liste(); // une fonction membre particulière, appelée destructeur
public: // spécification d'un mode de protection, a priori inutile
    void ajout(int); // une fonction membre banale
    // peuvent suivre d'autres déclarations de membres, publics ou privés :
};
```

En C++, le mot clé **`class`** peut être remplacé par le mot clé **`struct`** ; dans ce cas, tous les membres définis après l'accolade ouvrante de la définition de la structure sont publics.

Exemple : ces deux définitions sont équivalentes :

```
class Personne {
public: // des membres données
   char *nom,
        *prenom
   int   age;
        // des fonctions membres
   void saluer();
private:
   // autres membres
```

```
struct Personne {

   char *nom,
        *prenom;
   int   age;

   void saluer();
private:
    // idem
```

16.2.1. L'encapsulation

Le fait de définir au sein d'une même entité (la classe) des membres (les membres données, les fonctions qui les manipulent, voire des types locaux), et d'autoriser le monde extérieur à ne pouvoir manipuler qu'une partie de ces membres est appelé **encapsulation**. Une classe permet de définir des objets qui peuvent être considérés comme des boites noires sur lesquelles il est possible d'agir via les fonctions membres.

16.3. DEFINITION - DECLARATION D'UN OBJET D'UNE CLASSE

La déclaration d'un objet dont le type est une classe C++ suit la même syntaxe que la déclaration d'une variable de type **`struct`** ou **`union`** : il suffit d'indiquer le nom de la classe, puis une liste de définition de variables ; par exemple :

```
class Liste { … } p1, *p2;
// définition de la classe Liste, suivie de la définition d'un objet de
// type Liste appelé p1, puis d'un pointeur p2 sur une Liste
Liste &p3 = p1, p4[10];
// définition d'une référence p3 sur une Liste, puis d'un tableau p4
// de 10 Liste
```

Il n'est pas possible d'initialiser un objet d'une classe comme une structure, c'est à dire faire suivre la définition du nom de la variable par une initialisation de structure :

```
class Personne {
 public:
   int   age;
   char  *nom, *prenom;};

Personne lm = { 60, "Dupuis", "Martin" }; // impossible
```

L'initialisation d'un objet passe nécessairement par l'usage de fonctions membres particulières, les **constructeurs** (cf. sections suivantes).

16.4. ACCES AUX MEMBRES D'UN OBJET

16.4.1. Exploitation des membres

L'accès aux membres d'un objet suit les mêmes règles que pour les constructions **`struct`** ou **`union`**, c'est à dire via les opérateurs d'accès **`.`** et **`->`** si les membres

accédés sont des membres données ou des fonctions membres. L'exploitation de types membres est par contre différente.

Accès à un membre donnée

Soit *o* un objet de classe *C* dont la classe définit un membre donnée *m* ; l'accès à ce membre pour cet objet s'écrit *o*`.`*m* (opérateur d'accès `.`). Si *p* est un pointeur sur un objet de classe *C*, alors l'accès s'écrit *p*`->`*m* (opérateur d'accès `->`).

Exemple : avec la classe `Liste` définie précédemment, si `L` est un objet de la classe `Liste`, l'accès à son membre donnée `dernier` s'écrira :

```
L.dernier
```

Cette écriture de l'accès est en fait une forme simplifiée : le nom complet du membre donnée *m* est en fait *C*`::`*m* (qui se lit : le membre *m* défini dans la classe *C*), ce qui devrait conduire à des écritures d'accès du style :

o`.`*C*`::`*m* ou *p*`->`*C*`::`*m*

Sachant toutefois que les variables *o* ou *p* ont été au moins déclarées au préalable (par une forme du genre *C o* ou *C* p*), l'indication explicite de la classe d'appartenance du membre est redondante, i.e. inutile.

Dans la littérature, y compris celle produite par le comité de normalisation, `::` est appelé **opérateur de résolution de portée**. Cette appellation d'opérateur est toutefois inadaptée : aucun résultat n'est calculé. Il ne s'agit que d'une notation utile au compilateur lorsqu'il analyse le texte C++.

Accès à une fonction membre

Si *o* est un objet dont la classe *C* définit une fonction membre *m*`(…)`, alors l'accès à ce membre pour cet objet s'écrit *o*`.`*m*`(…)` (et s'écrit *p*`->`*m*`(…)` si *p* est un pointeur sur un objet de cette même classe). Cet accès a pour conséquence l'appel à la fonction membre.

Exemple : si `L` est un objet de la classe `Liste`, l'accès à sa fonction membre `ajout` s'écrira :

```
L.ajout(1)
```

Comme dans la section sur l'accès aux membres données, cette écriture de l'accès est une forme simplifiée : le nom complet de la fonction membre *m* est en fait *C*`::`*m*, ce qui devrait conduire à des écritures d'accès du style :

o`.`*C*`::`*m*`(…)` ou *p*`->`*C*`::`*m*`(…)`

Accès à un type membre

Si *T* est un nom de type membre défini dans une classe *C*, toute exploitation du nom de ce type, ou une quelconque de ses composantes, devra s'écrire *C*`::`*T* (le type *T* défini dans la classe *C*). Si toutefois le nom de ce type est exploité dans une définition d'une fonction membre de *C*, alors l'écriture pourra se réduire à *T* (pas d'indication explicite de la classe encapsulant *T*).

Ainsi, s'il est loisible d'omettre la classe d'appartenance d'un membre donnée ou d'une fonction membre, cette omission est par contre non tolérée pour les types membres, excepté dans les déclarations ou définitions des fonctions membres de la classe d'appartenance.

Exemple : soit la définition suivante :

```
class A {
public:
    enum B {a, b, c} ;    // un type membre
};
```

Toute exploitation du type énuméré `B` doit préciser la classe encapsulant sa définition ; idem pour ses littéraux :

```
main() {
    A::B uneVariable;     // exploitation du nom du type
    uneVariable = A::a ; // exploitation d'un composant du type
} ;
```

Cette particularité résout un problème évoqué lors de l'étude des unions : il devient possible, via l'encapsulation, d'user d'un même nom pour désigner un littéral dans différents types énumérés.

Exemple : avec les définitions suivantes :

```
class unePlanète {
    enum lesNoms { Vénus, Terre, Mars, Jupiter } ;
};

class unDieuRomain {
    enum lesNoms { Mars, Jupiter, Vénus } ;
};

class uneBarreChocolatée {
    enum lesNoms { Mars, Lion, Twix } ;
} ;
```

Il n'y a plus d'ambiguïté dans l'exploitation du nom `Mars`, puisque nous devrons écrire au choix :

```
UnePlanète::Mars
UnDieuRomain::Mars
UneBarreChocolatée::Mars
```

L'encapsulation peut opérer à n'importe quel niveau : une classe peut encapsuler un type membre, qui est lui-même une classe, laquelle peut encapsuler un type membre, etc. Dans ce cas, la résolution de portée **`::`** est spécifiée autant de fois que nécessaire afin de préciser toutes les classes à traverser pour atteindre la définition du type membre.

Exemple : soit la définition suivante :

```
class A {
public:
    class B { // une classe membre de A
    public:
        enum C {a, b, c} ;    // un type membre de B
    };
};
```

Toute exploitation du type énuméré `C` ou d'un de ses composants doit préciser la classe encapsulant sa définition :

```
main() {
    A::B::C uneVariable;     // nom du type
    uneVariable = A::B::a ; // composant du type
} ;
```

16.4.2. Incidence du mode de protection d'accès

La section précédente nous indique comment accéder à un membre d'une classe ou d'un objet. Ces accès sont toutefois conditionnés par les **modes de protection d'accès** qui leur sont associés. Soit une classe *C* définissant un membre *m* :

- si *m* est privé dans *C* (mode **`private`**), seules les fonctions membres de la classe *C* peuvent accéder à *m*
- si *m* est public (mode **`public`**), le membre est accessible partout.

Exemple : soient les classes suivantes :

```
class A {
  public:  int m1 ;
  private: int m2 ;
           void f();
} ;
```

```
class B {
  public:   int m3 ;
  private:  int m4 ;
            void f();
} ;
```

Dans les fonctions suivantes ont été barrés les accès interdits (accès à des membres privés dans des fonctions non membres de la classe définissant le membre) :

```
void A::f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

```
void B::f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

```
void f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

16.4.3. Fonctions amies

Il existe parfois des situations où une fonction nécessite des accès à des membres privés, sans toutefois que le concepteur de la fonction souhaite en faire une fonction membre du type concerné. Le problème est alors de rendre accessibles à une fonction des membres qui, en principe, ne le lui sont pas.

Le concepteur d'une classe peut, lors de sa définition, préciser quelles fonctions, en plus des classiques fonctions membres de la classe, auront tout de même accès aux membres privés: ces fonctions sont appelées des **fonctions amies**.

Cette **déclaration d'amitié** est précisée lors de la définition de la classe. Une telle déclaration porte soit sur une fonction (classique ou membre d'une classe), soit sur une classe. Dans ce dernier cas, toutes les fonctions membres de la classe deviennent des amies de la classe en cours de définition. Plusieurs déclarations d'amitié peuvent être définies au sein d'une même classe. La syntaxe d'une telle déclaration est :

```
friend signature_de_fonction;
```

ou

```
friend nom; // nom d'une class, union, ou struct
```

Exemple : nous avons repris l'exemple précédent, mais en rajoutant des clauses d'amitié :

```
class A {
  public:  int m1 ;
  private: int m2 ;
           void f();
  friend B ;
} ;
```

```
class B {
  public:   int m3 ;
  private:  int m4 ;
            void f();
  friend void f() ;
} ;
```

Auront accès à tous les membres de `A` toutes les fonctions membres de `A` plus toutes les fonctions membres de `B` (clause **`friend`** `B`). Auront accès à tous les membres de la classe `B` toutes les fonctions membres de `B` plus la fonction **`void`** `f()` (clause **`friend`** associée dans `B`). Avec ces informations, les accès interdits dans les fonctions suivantes ont été barrés :

```
void A::f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

```
void B::f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

```
void f() {
   A x ;
   x.m1 ;
   x.m2 ;

   B y ;
   y.m3 ;
   y.m4 ;
} ;
```

Attention, en C++, l'amitié est une propriété qui ne se transmet pas (non transitive), et qui n'est pas symétrique :

- si *A* est ami de *B*, et *B* ami de *C*, alors *A* n'est pas ami de *C*, sauf déclaration explicite.
- si *A* est ami de *B*, alors *B* n'est pas ami de *A*, sauf déclaration explicite.

16.5. LES FONCTIONS MEMBRES

16.5.1. Définition / déclaration d'une fonction membre

Lors de la définition d'une classe, une fonction membre peut être soit simplement déclarée (indication de sa seule signature), auquel cas elle devra être définie plus loin dans le fichier source, soit elle est définie (indication de la signature et définition du corps de la fonction).

La définition d'une fonction membre en dehors de la définition de sa classe suit le même schéma que celui d'une définition de fonction ordinaire, excepté que le nom de

la fonction membre doit impérativement être précédé (préfixé) du nom de la classe suivi de `::` (voir la section sur l'accès aux membres).

Exemple : soit la classe suivante :

```
class Point {
    float x, y; // deux membres données privés
  public:
    // La première fonction membre placer est définie dans la classe
    void placer(float a, float b) {
        this->x = a; this->y = b; };
    // La seconde est simplement déclarée
    void translater(float, float);
};
```

La fonction membre `translater` est définie en dehors de la classe par :

```
void Point::translater(float dx, float dy) {
    this->x = this->x + dx; this->y = this->y + dy; };
```

16.5.2. La variable prédéfinie `this`

Dans une section précédente, nous avons signalé que l'accès à une fonction membre *f* d'une classe *C* via un objet *o* de classe *C* ou un pointeur *p* sur un objet de classe *C* (donc *p* est du type *C**) s'écrit :

$$o.f(\ldots) \text{ ou } p->f(\ldots)$$

L'accès ressemble donc à un appel de fonction classique, et le nombre d'arguments figurant dans l'appel doit en principe coïncider avec le nombre de paramètres spécifiés dans la fonction (moins contraignant si des valeurs par défaut ont été spécifiées).

Exemple : avec la classe `Point` définie précédemment, nous pourrions écrire :

```
main() {
    Point p ;
    float X, Y ;
    cout << "x = " ; cin >> X ;
    cout << "y = " ; cin >> Y ;
// accès à la fonction membre en respectant le nombre de paramètres
    p.placer(X, Y) ;
    p.translater(1, -1) ;
} ;
```

Le premier appel (accès à la fonction membre `placer`) signifie que nous forçons le point `p` à se placer en position (`X`, `Y`). Le second oblige le point à se déplacer d'un déplacement (Δ_x, Δ_y), où $\Delta_x = 1$ et $\Delta_y = -1$; le point devrait donc se trouver en ($X+\Delta_x$, $Y+\Delta_y$) après ce dernier appel.

A chaque fois qu'un appel à une fonction membre *f* opère, sont au moins introduits dans l'environnement tous les paramètres de la fonction (comportement normal lors d'un appel de fonction), mais aussi une autre variable, implicitement ajoutée par le compilateur à la définition de la fonction membre, telle que :

- son nom est **`this`** (c'est un mot clé de C++)
- son type est *C**, où *C* est le nom de la classe définissant *f*

- sa valeur est un pointeur sur l'objet qui a servi à l'appel (accès) de la fonction membre

Autrement dit, si nous faisons abstraction des protections d'accès posées sur les membres, nous pouvons considérer que :

- si *f* est une fonction membre d'une classe *C* ayant *n* paramètres p_i de types T_i et retournant un résultat de type *R*, définie par :

  R C::*f*(*T*$_1$ *p*$_1$, …, *T*$_n$ *p*$_n$) { … }
- si *o* est un objet de classe *C* accédant à *f*, soit (chaque e_i est une expression d'un type compatible avec T_i) :

  o.*f*(*e*$_1$, …, *e*$_n$)

alors tout se passe comme si le compilateur transformait notre code ainsi :

- à la fonction membre *f* est associée une fonction classique C_*f* définie par :

  R C_*f*(*C** **this**, *T*$_1$ *p*$_1$, …, *T*$_n$ *p*$_n$) { … } // *même corps que f*
- l'accès à la fonction membre via *o* est transformé en un appel classique :

  C_*f*(&*o*, *e*$_1$, …, *e*$_n$)

Comme la grande majorité des fonctions membres font accès à des membres via le pointeur **this**, le langage C++ autorise l'omission du texte **this**-> dans les fonctions membres.

Exemple : la fonction membre `translater` de la classe `Point` peut être simplifiée en :

```
void Point::translater(float dx, float dy) {
    x = x + dx; y = y + dy; };
```

16.5.3. Les constructeurs

Il existe deux catégories de fonctions membres particulières : les constructeurs et les destructeurs.

Un constructeur (de données) est une fonction membre qui est appelée lorsqu'un objet d'une classe est nouvellement créé (lors d'une définition, ou par un appel explicite de l'opérateur **new** d'allocation dynamique). Ses principales caractéristiques sont les suivantes :

- le nom d'un constructeur est toujours le nom de la classe à laquelle il est rattaché.
- aucune spécification du type du résultat ne doit être mentionnée
- une même classe peut posséder plusieurs constructeurs : seules leurs signatures permettront de les distinguer.

La déclaration / définition d'un constructeur suit en gros les mêmes règles que celles des fonctions membres.

Définition d'un constructeur

Si *C* est une classe définissant *n* membres données m_1, …, m_n, chaque membre étant d'un type T_i, un constructeur de cette classe se déclare en écrivant :

```
C(S1, …, Sk)
```

si la déclaration figure dans la définition de la classe (les S_j sont les types des paramètres du constructeurs ; il peut n'y en avoir aucun), ou :

```
C::C(S1, …, Sk)
```

si cette déclaration figure hors de la définition de la classe. Il est possible d'indiquer des valeurs par défaut, comme n'importe quelle fonction.

La définition d'un constructeur s'écrit :

```
C(S1 p1, …, Sk pk)
 : m1(a1), m2(a2), …, mn(an)  // initialisation des membres données
 { … }                        // corps
```

si la définition figure dans la définition de la classe (les a_j sont une liste d'expressions pouvant exploiter les paramètres des constructeurs ou tout nom dans la portée), ou encore :

```
C::C(S1 p1, …, Sk pk)
 : m1(a1), m2(a2), …, mn(an)  // initialisation des membres données
 { … }                        // corps
```

si cette définition figure hors de la définition de la classe.

La première partie *initialisation des membres données* (après le **:** mais avant l'ouverture du corps du constructeur) peut être omise. Si tel est le cas, le compilateur l'ajoute automatiquement ; ainsi :

```
C::C(S1 p1, …, Sk pk)
{ … }                         // corps
```

est automatiquement transformé en (voir ci-dessous la section sur l'initialisation des objets, en particulier les constructeurs vides) :

```
C::C(S1 p1, …, Sk pk)
 : m1(), m2(), …, mn()       // initialisation des membres données
 { … }                       // corps
```

Initialisation des objets

Si *C* est une classe et qu'un objet *o* de classe *C* est défini, cette définition s'écrit ainsi :

```
C o = C::C(e1, …, ek) ;
```

Cette définition précise explicitement quel constructeur doit servir à initialiser l'objet nouvellement introduit.

Un certain nombre de simplifications d'écriture sont toutefois autorisées :

Forme initiale	Forme simplifiée
`C o = C::C(e1, …, ek) ;`	`C o = C(e1, …, ek) ;`
`C o = C(e1, …, ek) ;`	`C o(e1, …, ek) ;`
`C o = C(e1) ;`	`C o = e1 ;`
`C o = C() ;`	`C o;`

La dernière simplification montre que si aucune initialisation explicite ne figure dans la définition, c'est le constructeur *C*`()` qui sera exploité pour l'initialisation ; ce constructeur est appelé **constructeur vide** ou **constructeur par défaut**.

L'ordre des opérations intervenant lors d'une définition d'objet est le suivant :

1. allocation de l'espace mémoire nécessaire à l'objet à définir
2. mise en place de l'appel du corps du constructeur : introduction dans l'environnement de toutes les variables p_i paramètres du constructeur, initialisées avec les valeurs des expressions e_i correspondantes. Comme toute fonction membre, un paramètre additionnel figure dans cette liste : la variable **`this`**, laquelle aura pour valeur le pointeur sur l'objet en cours de définition.
3. initialisation de chaque membre m_j de l'objet en cours de définition en exploitant le constructeur du type T_j du membre, auquel sont passés les arguments a_j
4. exécution du corps du constructeur

Exemple : le programme suivant :

```
class A { public: int z ; A(int) ; } ;
A::A(int v)
 : z(v)
 { cout << "A::A(int)" << endl ;};

class B { public: int x ; A y ; B(int, int) ;};
B::B(int v, int w)
 : x(v), y(w)
 { cout << "B::B(int, int)" << endl ;};

main() {
    B o = B(1, 2) ;
} ;
```

provoque l'affiche des messages suivant à l'écran :

```
A::A(int)
B::B(int, int)
```

En effet, l'initialisation de `o` implique d'exécuter le constructeur `B(int, int)`. Nous commençons par l'initialisation des membres données, qui ici impliquent de ranger 1 dans le membre donnée `x` de `o`, et l'initialisation du membre `y` de `o`, via le constructeur `A(int)`. Cette initialisation provoque l'initialisation du membre donnée `z` de `o.y`, avec pour résultat d'y ranger l'entier 2. Le corps du constructeur `A(int)` s'exécute, provoquant le premier message, et achevant l'initialisation du membre `y` de `o`. Tous les membres de `o` étant initialisés, le corps du constructeur `B(int, int)` est exécuté, d'où le second message.

16.5.4. Les destructeurs

Un **destructeur** (de données) est une fonction membre qui est appelée lorsqu'un objet d'une classe est désalloué (lors d'une fermeture d'un bloc portant la définition de cet objet, ou par un appel explicite de l'opérateur de désallocation **`delete`** si l'objet a été alloué dynamiquement).

Le nom d'un destructeur est toujours le nom de la classe à laquelle il est rattaché, précédé du caractère ~ (tilde) ; comme les constructeurs, aucune spécification de type du résultat ne doit figurer dans sa signature. Une classe ne possède au mieux qu'un seul destructeur. La définition d'un destructeur suit les mêmes règles que celles des fonctions membres.

Séquence des opérations

L'ordre des opérations intervenant lors d'une destruction d'objet est exactement symétrique à celui des constructeurs, soit :

1. mise en place de l'appel du corps du destructeur : introduction dans l'environnement de la variable **`this`**, laquelle aura pour valeur le pointeur sur l'objet à détruire.
2. exécution du corps du destructeur
3. destruction de chaque membre m_j de l'objet à détruire en exploitant le destructeur du type T_j du membre
4. désallocation de l'espace mémoire associé à l'objet à détruire

Exemple : le programme suivant :

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

class B { public: A x ; ~B() ;};
B::B() : x() { cout << "B::B()" << endl ;};
B::~B() { cout << "B::~B()" << endl ;};

main() {
    B o;
} ;
```

provoque l'affiche des messages suivant à l'écran :

```
A::A()
B::B()
B::~B()
A::~A()
```

En effet, l'initialisation de `o` implique d'exécuter le constructeur `B()`. Nous commençons par l'initialisation des membres données, qui ici se limite à l'initialisation du membre `x` de `o`, via le constructeur `A()`. Cette initialisation provoque l'exécution du corps du constructeur `A()`, provoquant le premier message, et achevant l'initialisation du membre `x` de `o`. Tous les membres de `o` étant initialisés, le corps du constructeur `B()` est exécuté, d'où le second message.

Lorsque le bloc de `main` se ferme, l'objet `o` est détruit. Le corps du destructeur `~B()` est d'abord exécuté, puis la destruction de tous les membres de `o`, ici le seul membre `x`, provoque l'exécution du destructeur `~A()`, d'où les messages.

16.5.5. Les fonctions membres définies par défaut

Lorsqu'une classe est définie, le compilateur peut être amené à définir automatiquement certaines fonctions membres s'il le nécessite. Dans cette section, nous supposons que *C* est une classe définissant *n* membres données m_1, …, m_n, chaque membre donnée m_i étant d'un type T_i.

Le constructeur vide

Si aucun constructeur n'est défini pour la classe *C*, le compilateur associe automatiquement à *C* un constructeur vide, défini par :

```
C::C()
 : m1(), m2(), …, mn()  // initialisation des membres données
 {}                     // corps vide
```

L'initialisation des membres données s'appuie sur les constructeurs vides de leurs types associés.

Dès lors qu'au moins un constructeur a été explicitement défini par le concepteur de la classe, le compilateur n'ajoute plus cette définition par défaut du constructeur vide.

Le constructeur de recopie

Lorsqu'une fonction possède un paramètre de type *C* (notre classe), impliquant donc un passage par valeur, un constructeur est chargé de recopier la valeur passée au paramètre : le **constructeur de recopie** (d'où son nom). La signature de ce constructeur est toujours :

```
C::C(const C&) ;
```

Exemple : examinons le code suivant :

```
class A { public: int a ; A(int) ; void f() ; } ;
A::A(int v) : a(v) {} ;
void A::f() { cout << a << endl ; } ;
void fonction(A p) { p.f() ; } ;
main() { A o(2) ; fonction(o) ; } ;
```

Nous savons que l'appel à `fonction` dans `main` est équivalent à exécuter un sous-bloc du style (cf. la section sur les appels de fonctions) :

```
{
    A p = o ;    // introduction du paramètre et initialisation
    { p.f() ; } // corps de la fonction
} ;
```

La première ligne est en fait une forme simplifiée, laquelle devrait s'écrire :

```
A p = A::A(o) ; // introduction du paramètre et initialisation
```

Cette forme montre que l'initialisation du paramètre s'appuie sur un constructeur auquel est passé un objet A ; or aucun constructeur de ce type n'est explicitement défini dans la classe. Pourtant, ce programme est parfaitement compilé, sans erreur ; le compilateur exploite en fait le constructeur de recopie qu'il aura lui-même défini !

Quelle que soit une classe *C* définie, il existera toujours pour cette classe un constructeur de recopie :

- soit ce constructeur est explicitement défini par le concepteur de la classe
- soit ce constructeur est automatiquement ajouté par le compilateur, avec la définition suivante :

```
C::C(const C& o)
 // initialisation des membres données
 : m1(o.m1), m2(o.m2), …, mn(o.mn)
 {}                              // corps vide
```

Cette définition par défaut montre que le constructeur de recopie se contente de recopier un à un tous les membres de l'objet à copier ; on parle de **copie membre à membre**.

Si le concepteur conçoit une classe dont l'un des membres données exploite l'allocation dynamique, il doit **impérativement** définir un constructeur de recopie s'il souhaite que son code fonctionne correctement.

Le destructeur par défaut

Si aucun destructeur n'est défini dans une classe *C*, le compilateur ajoute automatiquement une définition par défaut de ce destructeur, limitée à sa plus simple expression :

```
C::~C() {} // corps vide
```

L'opérateur d'affectation

Bien que cet opérateur ne devrait pas être étudié dans ce cours, nous le mentionnons ici car il est étroitement associé au constructeur de recopie ; il joue un rôle capital dans les opérations d'affectation entre objets exploitant des valeurs allouées dynamiquement. Pour toute classe *C* définie, il est toujours possible d'écrire une affectation d'une valeur de type *C* dans une variable de même type *C*.

```
class C { … } ;
C v1 ;
C v2 ;
v1 = v2 ;
```

Cette affectation est totalement contrôlée par la fonction membre suivante :

```
C& C::operator=(const C&) ;
```

Deux situations se présentent :

- soit cette fonction membre est explicitement définie par le concepteur de la classe
- soit elle est automatiquement ajoutée par le compilateur, avec la définition suivante :

```
C& C::operator=(const C& o) {
     m1 = o.m1 ;
     m2 = o.m2 ;
     …
     mn = o.mn ;
     return *this ; // retourne toujours l'objet affecté lui-même
} ;
```

Cette définition par défaut ressemble fortement à celle du constructeur de recopie : tous les membres de l'objet à affecter sont affectés un à un; on parle d'**affectation membre à membre**.

Comme le constructeur de recopie, toute classe dont l'un des membres données exploite l'allocation dynamique doit **impérativement** définir un opérateur d'affectation, sous peine d'erreurs à l'exécution dans certains cas. Ces difficultés ne sont toutefois pas au programme du cours IPA.

16.6. RETOUR AUX CONSTRUCTIONS DE TYPE STRUCT ET UNION

Les constructions de type que sont **`struct`** et **`union`** ont été étendues en C++ afin d'offrir les mêmes possibilités que la construction de type **`class`** :

- protection des membres (par défaut, le mode est **`public`**)
- définition possible de fonctions membres (avec constructeurs et destructeur)
- définition possible de types membres