C,
un premier
langage

Jacques Le Maitre

Département d'informatique
UFR de Sciences et Techniques
Université de Toulon et du Var

Table des matières

1

Introduction

1.1 C, son origine et son importance

Le langage C a été créé au début des années 70 par des chercheurs du laboratoire de la compagnie Bell aux USA. Il a été conçu, à l'origine, pour être le langage de programmation du système d'exploitation UNIX.

C est un langage très largement diffusé, en tant que tel, mais aussi comme noyau des langages C++, un langage de programmation objet et Java, le langage de programmation du Web .

1.2 Exemple de programme C

Le programme suivant est écrit en C. Il demande à son utilisateur d'entrer deux nombres *x* et *y*, puis calcule le plus grand, *max*, de ces deux nombres et l'affiche.

```
(1)  #include <stdio.h>
(2)  /*
(3)   * Maximum de deux nombres entiers
(4)   */
(5)  main()
(6)    {
(7)    int x, y, max;
(8)    printf("x ? ");
(9)    scanf("%d", &x);
(10)   printf("y ? ");
(11)   scanf("%d", &y);
(12)   if (x >= y)
(13)     max = x;
(14)   else
(15)     max = y;
(16)   printf("max = %d", max);
(17)   }
```

Ce programme est construit de la façon suivante ;

- A la ligne 1 on indique que les fonctions `printf` et `scanf` des lignes 8, 9, 10, 11 et 16 ont le type déclaré dans le fichier « `stdio.h` ». Ces fonctions permettent respectivement, d'afficher ou de saisir des valeurs.
- Les lignes 2 à 4 contiennent un commentaire qui, ici, indique ce que calcule le programme.

- Les lignes 7 à 17 contiennent la définition de la fonction `main`. Un programme C est constitué d'une suite de définitions de fonctions. Parmi elles, la fonction `main` qui doit toujours être présente et qui est exécutée la première. Le corps de cette fonction est le bloc qui commence à la ligne 6 et se termine à la ligne 17.
- La ligne 7 contient la déclaration de trois variables `x`, `y` et `max` de type `int` (nombre entier). A la suite de cette déclaration trois cases désignées par `x`, `y` et `max` sont alloués dans la mémoire du programme. Elles contiendront chacune un nombre entier.
- La ligne 8 contient une instruction dont l'exécution (en abrégé, nous dirons : « une instruction qui ») affiche un message indiquant que le programme attend la saisie de la valeur qui sera affectée à la case mémoire désignée par `x`. L'instruction de la ligne 9 saisit cette valeur. Les instructions des lignes 10 et 11 agissent de même pour la variable `y`.
- L'instruction `if` qui commence à la ligne 12 et se termine à la ligne 15 compare les deux valeurs saisies. Si la valeur affectée à la case mémoire désignée par (en abrégé, nous dirons : « la valeur de ») `x` est supérieure à celle de `y`, alors (ligne 13) la valeur de `x` est enregistrée dans la case mémoire désignée par (en abrégé nous dirons : « est affectée à ») `max`. Sinon (ligne 14), c'est la valeur de `y` qui est affectée à `max` (ligne 15).

1.3 Réalisation d'un programme C

La réalisation d'un programme C se déroule en trois étapes :

1. Saisir le texte du programme, appelé **texte source**, sous un éditeur de texte et le sauvegarder dans un fichier, appelé **fichier source**.
2. Soumettre le fichier source à un **compilateur** C qui traduit le programme en code machine et produit un **fichier exécutable**. Sous UNIX cela peut être réalisé par la commande `cc` qui produira un fichier exécutable `a.out`.
3. Pour exécuter le programme sous UNIX, taper la commande `a.out`.

1.4 Livres sur C

Nous conseillons tout particulièrement les deux livres suivants :

- B. Kernighan et D. Ritchie, *Le langage C, norme ANSI*, Dunod ;
- H. Garetta, *C* : *Langage, bibliothèque, applications*, InterEditions.

Le premier est celui des inventeurs de C. Le second, que nous avons largement utilisé pour rédiger ce cours, est le fruit de nombreuses années d'enseignement de la programmation à l'Université de la Méditerranée.

2

Données

Un programme C manipule des **données**. Une donnée a un **type** et une **valeur**, elle peut avoir un **nom** et peut être **constante** ou **modifiable**. La valeur d'une donnée modifiable peut changer au cours de l'exécution d'un programme alors que celle d'une donnée constante ne le peut pas.

On peut voir une donnée modifiable comme une case de la mémoire accessible à partir du nom de la donnée et qui contient la valeur de cette donnée.

Supposons que l'on veuille écrire un programme qui calcule le volume d'une sphère dont le rayon est donné par l'utilisateur du programme. Le nombre π sera représenté comme une donnée constante. Le rayon et le volume d'une sphère seront représentées comme des données modifiables. On considérera donc qu'il y a dans la mémoire du programme une case nommée *rayon* dans laquelle sera enregistrée le rayon de la sphère dont le volume doit être calculé et une case nommée *volume* dans laquelle le volume de la sphère sera enregistré une fois qu'il aura été calculé.

Une valeur peut être :

- un **nombre entier** ou un **nombre flottant**,
- une valeur composée : une **structure** ou un **tableau**,
- un **pointeur** (ou une **adresse**).

Les valeurs sont classées par **types**. Toute valeur a un et un seul type.

Il n'existe pas de type spécifique pour les booléens, les caractères et les chaînes de caractères. Le booléen « faux » est représenté par le nombre 0 et le booléen « vrai » par toute valeur non nulle. Un caractère est codé par un nombre entier (en ASCII, en général). Une chaîne de caractères est représentée comme un tableau de caractères (c.-à-d. comme un tableau de nombres entiers).

Dans ce chapitre, nous n'étudierons que les types numériques. Les structures et les tableaux seront étudiés au chapitre 5 et les pointeurs au chapitre 9.

2.1 Types numériques

Le tableau suivant présente les types numériques du langage C.

	Nom du type	Taille (la plus classique)	Ensemble des valeurs
Entiers	`char`	8 bits	-128 à 127
	`unsigned char`	8 bits	0 à 255
	`short`	16 bits	-32 768 à 32 767
	`unsigned short`	16 bits	0 à 65 535
	`long`	32 bits	-2 147 483 648 à 2 147 483 647
	`unsigned long`	32 bits	0 à 4 294 967 296
	`int`	Ce type rassemble les entiers qui peuvent être implantés de la façon la plus efficace sur la machine qui exécute le programme. Il est le plus souvent équivalent au type `long`.	
Flottants	`float`	32 bits	$-1,7.10^{38}$ à $-0,29.10^{-38}$ $0,29.10^{-38}$ à $1,7.10^{38}$ 7 chiffres décimaux signicatifs
	`double`	64 bits	$-0,9.10^{308}$ à $-0,56.10^{-308}$ $0,56.10^{-308}$ à $0,9.10^{308}$ 15 chiffres décimaux signicatifs
	`long double`	Ce type rassemble les flottants de grande précision.	

Deux remarques importantes :

- La taille associée à un type est dépendante du compilateur. Les tailles indiquées dans le tableau ci-dessus sont les plus usuelles.
- Lorsque qu'une variable a pour valeur un entier positif il y a intérêt à utiliser un type `unsigned` afin de disposer de la plus grande plage possible pour une même occupation mémoire.

2.2 Constantes littérales

Les nombres positifs, les caractères et les chaînes de caractères ont une représentation explicite, que nous appellerons **constante littérale**.

Les nombres négatifs n'ont pas à strictement parler de représentation littérale. Ils sont obtenus en appliquant l'opérateur unaire - à leur valeur absolue (voir ci-dessous §3.3.1).

2.2.1 Nombres entiers

Une constante littérale entière est soit `0`, soit une suite de chiffres dont le premier est différent de `0`. Par exemple :

```
0    1515
```

Une constante littérale entière est la représentation d'une valeur du plus petit des trois types `int`, `long` ou `unsigned long` qui contient cette valeur. Par exemple, `141946` est de type `long` et `3456789012` est de type `unsigned long`. On peut changer ce type en ajoutant le

suffixe `u` ou `U` pour `unsigned` et `l` ou `L` pour `long`. Par exemple, `1515L` est de type `long` et `1515UL` de type `unsigned long`.

2.2.2 Nombres flottants

Une constante littérale flottante est formée :

- d'une **partie entière** : une suite de chiffres,
- suivie d'un point,
- suivi d'une **partie fractionnaire** : une suite de chiffres,
- suivie d'un **exposant** : la lettre `e` ou `E`, suivie du signe `+` ou `-`, suivi d'une suite de chiffres.

On peut omettre :

- la partie entière ou la partie fractionnaire, mais pas les deux,
- le point ou l'exposant, mais pas les deux,

Par exemple, les constantes littérales flottantes suivantes :

```
2731.5e-1   .15    2e3
```

représentent respectivement les nombres :

273,15 0,15 2000

Une constante littérale flottante est la représentation d'une valeur de type `double`. On peut changer ce type en ajoutant le suffixe `f` ou `F` pour `float` et `l` ou `L` pour `long double`. Par exemple, `.15F` est de type `float` et `2731.5e-1L` est de type `long double`.

2.2.3 Caractères

Une constante littérale caractère a l'une des deux formes suivantes.

1. Un caractère imprimable entre apostrophes. Par exemple :

   ```
   'A'    '1'
   ```

2. Un caractère précédé du caractère `\` (« anti-slash ») dit « caractère d'échappement » :

`\n`	nouvelle ligne
`\t`	tabulation
`\b`	retour arrière
`\r`	retour chariot
`\f`	saut de page
`\a`	signal sonore
`\`*c*	caractère *c* (où *c* est différent de `n`, `t`, `b`, `r`, `f` ou `a`)

   Par exemple, le caractère « guillemet » peut se représenter `\"`.

Une constante littérale caractère est la représentation d'une valeur de type `char`. Par exemple, si les caractères sont codés en ASCII et que le code ASCII de la lettre A est 65, la constante littérale `'A'` représente le nombre entier `65` de type `char`.

2.2.4 Chaîne de caractères

Une constante littérale chaîne de caractères est formée par la suite de ses caractères, placée entre guillemets. Par exemple :

```
"Bonjour !"   "Je vous dis : \"Bonjour !\""
```

Notons l'utilisation du caractère d'échappement `\` qui permet de distinguer les guillemets appartenant à la chaîne de caractères de ceux qui servent à la délimiter.

Une constante littérale chaîne de caractères est la représentation d'une valeur de type « pointeur vers un caractère ».

2.3 Déclaration de variables

Une **variable** est une donnée modifiable définie par son nom : un **identificateur** et par le type des valeurs qu'elle peut prendre.

Un identificateur est une suite caractères dont chacun peut être une lettre, un chiffre ou le caractère _ et dont le premier caractère est une lettre. Par exemple :

```
x    y1   ma_variable
```

Les variables manipulées par un programme doivent être déclarées avant leur apparition dans le texte source du programme. Nous ne traiterons pour le moment que de la déclaration des variables numériques. La déclaration des variables de type structure, tableau et pointeur seront étudiées dans les chapitres suivants.

Une déclaration de variables numériques a la forme suivante :

T *ident*$_1$ `=` *exp*$_1$`, …,` *ident*$_n$ `=` *exp*$_n$ `;`

où :

- *T* est l'un des types numériques définis au §2.1,
- *ident*$_1$, …, *ident*$_n$ sont des identificateurs,
- *exp*$_1$, …, *exp*$_n$ sont des expressions.

Cette déclaration définit *n* variables de type *T*, nommées *ident*$_1$, …, *ident*$_n$, et qui ont pour valeur initiales les valeurs des expressions *exp*$_1$, …, *exp*$_n$.

L'expression des valeurs initiales dans une déclaration de variables ainsi que les valeurs initiales attribuées aux variables non initialisées dans leur déclaration, dépendent de la visibilité de cette variable, concept que nous étudierons au chapitre 8. Dans ce cours, nous adopterons une règle stricte en imposant que l'expression d'une valeur initiale soit une **expression constante** (c.-à-d. une expression sans variables) et en considérant qu'une variable non initialisée dans sa déclaration a une valeur initiale indéterminée.

Par exemple, les déclarations :

```
int i, j, k;
float longueur = 10.5, largeur = 6.3;
char une_lettre;
```

définissent respectivement :

- trois variables nommées `i`, `j` et `k` de type `int`, sans valeur initiale ;
- deux variables nommées `longueur` et `largeur` de type `float` et de valeurs initiales respectives 10,5 et 5,3 ;
- une variable nommée `une_lettre` de type `char` sans valeur initiale.

On peut « fixer » la valeur d'une variable en préfixant sa déclaration par le mot-clé `const`. Cette variable se comporte alors comme une constante : sa valeur, obligatoirement fournie dans la déclaration, ne pourra pas être modifiée au cours de l'exécution du programme.

Par exemple, la déclaration :

```
const float pi = 3.14;
```

déclare une donnée constante, de nom pi, de type float et de valeur 3.14.

2.4 Environnement et état

Par la suite, nous appellerons :

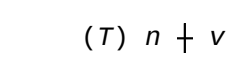
- *environnement*, l'ensemble des données visibles en un point d'un programme ;
- *état*, l'ensemble des valeurs des données à un instant de l'exécution d'un programme.

Il est important de noter que l'environnement est de nature statique alors que l'état est de nature dynamique. L'environnement dans lequel sera exécuté une instruction peut être déterminé par la lecture du texte source de ce programme. Par contre, l'état d'un programme dépend de l'historique de son exécution.

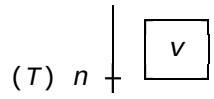
Considérons, par exemple, le programme du §1.2. L'environnement établi par la déclaration de la ligne 7 est formé des variables `x`, `y` et `max`. Si les valeurs attribuées à `x` et `y` (lignes 9 et 11) sont 32 et 55, l'état du programme après l'exécution de l'instruction de la ligne 12 est `x` = 32, `y` = 55 et `max` = 55.

Dans la suite de ce cours nous visualiserons graphiquement un environnement et un état de la façon suivante :

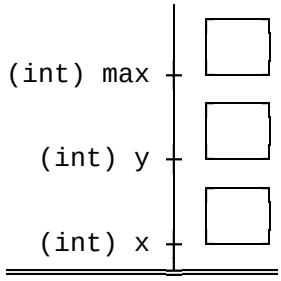
- une donnée constante de nom *n*, de type *T* et de valeur *v* sera représentée par :



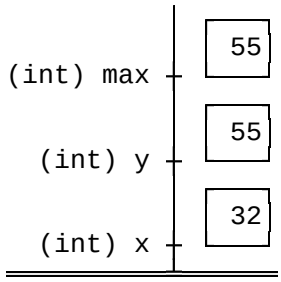
- une donnée modifiable de nom *n*, de type *T* et de valeur *v* sera représentée par :



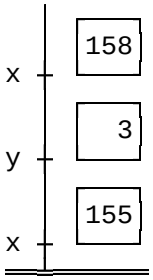
- l'environnement en un point d'un programme sera obtenu en empilant les représentations graphiques des données déclarées depuis le début de ce programme jusqu'à ce point. Par exemple, l'environnement dans lequel s'exécute le programme du §1.2 est le suivant :



- la représentation graphique de l'état d'un programme à un instant donné est obtenu en remplissant les cases mémoires associées à chaque donnée modifiable par leurs valeurs à cet instant. L'état du programme du §1.2 après l'exécution de l'instruction de la ligne 12 est le suivant :



Nous verrons que dans un même programme plusieurs variables de même nom peuvent être déclarées. La règle est que la dernière déclaration cache les autres. La valeur d'une donnée de nom *n* est donc obtenue en recherchant la valeur de la première donnée de nom *n*, depuis le sommet de la pile. Par exemple, dans l'état suivant :



la valeur de la variable x est 158 et non 155.

3

Expressions

3.1 Qu'est-ce qu'une expression ?

Une expression est une phrase formée à l'aide de constantes littérales, de noms de variables et d'opérateurs. Par exemple, dans un environnement où la variable `x` est visible, les phrases suivantes :

```
12
x
(x + 12) - 3
(x > 4) && (x > 10)
```

sont des expressions. Les deux premières sont des expressions simples et les deux dernières sont des expressions composées.

En C, toute expression a obligatoirement :

- un type,
- une valeur,

et éventuellement :

- une **adresse** qui est celle de la case mémoire qui contient la valeur de cette expression,
- un **effet de bord** qui change l'état du programme.

On distingue deux catégories d'expression :

- les **valeurs gauches** qui sont des noms de donnée modifiable. Une valeur gauche est donc associée à une case de la mémoire et a une adresse.
- **valeur droite**, qui sont des expressions qui ne sont pas associées à une case de la mémoire et qui donc qui n'ont pas d'adresse. Une valeur gauche peut être vue comme une valeur droite.

Si *e* est une expression nous noterons :

- *type*(*e*), le type de *e*,
- *val*(*e*) la valeur de *e*.

Une expression peut être mise entre parenthèses. Si *exp* est une expression, alors `(`*exp*`)` est une expression de même type et de même valeur que *exp*.

3.2 Expressions atomiques

- Si *c* est une constante littérale, alors *c* est une expression telle que :

- *type*(*c*) = type de la valeur représentée par *c*,
- *val*(*c*) = valeur représentée par *c*.

- Si *v* est un nom de variable, alors *v* est une expression telle que :
  - *type*(*v*) = type déclaré pour *v*,
  - *val*(*v*) = valeur de *v* dans l'environnement courant.

Un nom de variable est une valeur gauche, puisqu'il est le nom d'une donnée modifiable.

3.3 Expressions composées

Les expressions composées, le sont à l'aide d'opérateurs. Dans ce chapitre nous étudierons :

- les opérateurs arithmétiques,
- les opérateurs booléens,
- l'affectation,
- le changement de type.

Les opérateurs de manipulation des valeurs composées (structures et tableaux) seront étudiés au chapitre 5 et ceux de manipulation des pointeurs seront étudiés au chapitre 9.

3.3.1 Moins unaire

L'opérateur – calcule l'opposé d'un nombre.

Si *exp* est une expression de type numérique, alors :

`-`*exp*

est une expression telle que :

- *type*(`-`*exp*) = *type*(*exp*),
- *val*(`-`*exp*) = -*val*(*exp*),

3.3.2 Opérateurs arithmétiques binaires

Les opérateurs `+ - * /` calculent respectivement la somme, la différence, le produit et le quotient de deux nombres. L'opérateur `%` calcule le reste de la division de deux nombres entiers.

Si exp_1 et exp_2 sont des expressions de type numérique, alors

exp_1 `+` exp_2
exp_1 `-` exp_2
exp_1 `*` exp_2
exp_1 `/` exp_2
exp_1 `%` exp_2

sont des expressions telles que :

- *type*(exp_1 *op* exp_2) = *T*, où *T* est le plus grand des types `int`, *type*(exp_1) et *type*(exp_2) (voir ci-dessous §3.4.2).
- *val*(exp_1 *op* exp_2) = v_1 *op* v_2, où v_1 et v_2 sont le résultat des conversions de *val*(exp_1) et de *val*(exp_2) en *T*.

Par exemple :

val(((4.0 + 2.25) * 2.0)) = 12.5

val(15 / 2) = 7

val(15 % 2) = 1

3.3.3 Comparateurs

Les opérateurs == != < <= > >= testent respectivement l'égalité, la différence, l'infériorité, l'infériorité ou l'égalité, la supériorité et la supériorité ou l'égalité de deux nombres.

Si exp_1 et exp_2 sont des expressions de type numérique, alors :

exp_1 == exp_2
exp_1 != exp_2
exp_1 < exp_2
exp_1 <= exp_2
exp_1 > exp_2
exp_1 >= exp_2

sont des expressions telles que :

- *type*(exp_1 *op* exp_2) = int (ou *op* est l'un des comparateurs).
- *val*(exp_1 *op* exp_2) = v_1 *op* v_2, où v_1 et v_2 sont le résultat des conversions de *val*(exp_1) et de *val*(exp_2) dans le plus grand des types int, *type*(exp_1) et *type*(exp_2) (voir ci-dessous §3.4.2.1).

Par exemple :

val(1 == 9) = 0

3.3.4 Négation

L'opérateur ! calcule la négation d'un booléen.

Si *exp* est une expression, alors :

!*exp*

est une expression telle que :

- *type*(*exp*) = int.
- *val*(!*exp*) = 0 si *val*(*exp*) = 0, 1 sinon.

3.3.5 Conjonction et disjonction.

Les opérateurs || et && calculent respectivement la conjonction et la disjonction de deux booléens.

Si exp_1 et exp_2 sont des expressions, alors :

exp_1 || exp_2
exp_1 && exp_2

sont des expressions telles que :

- *type*(exp_1 *op* exp_2) = int.

- *val*(*exp*$_1$ `||` *exp*$_2$) =

  si *val*(*exp*$_1$) ≠ `0` alors
  1
  sinon
  si *val*(*exp*$_2$) ≠ `0` alors
  1
  sinon
  0
  fin-si
  fin-si

- *val*(*exp*$_1$ `&&` *exp*$_2$) =

  si *val*(*exp*$_1$) = 0 alors
  0
  sinon
  si *val*(*exp*$_2$) ≠ `0` alors
  1
  sinon
  0
  fin-si
  fin-si

- Le deuxième opérande n'est pas évalué si ce n'est pas nécessaire, c.-à-d. si le premier opérande a la valeur 1 dans le cas de l'opérateur `||` ou la valeur 0 dans le cas de l'opérateur `&&`.

Par exemple :

val(`!((1 || 0) && (0 || 1))`) = `0`

3.3.6 Affectation

L'opérateur = affecte une valeur à une donnée modifiable. Plus exactement, il enregistre cette valeur dans la case mémoire attachée à cette donnée.

Si *exp*$_1$ est une expression qui est le nom d'une donnée modifiable (une valeur gauche) et *exp*$_2$ est une expression expressions, alors :

exp$_1$ `=` *exp*$_2$

est une expression telle que :

- *type*(*exp*$_1$ `=` *exp*$_2$) = *type*(*exp*$_1$).
- *val*(*exp*$_1$ `=` *exp*$_2$) = *val*(*exp*$_2$) convertie en *type*(*exp*$_1$) (voir ci-dessous §3.4.2.2).
- Il y a un effet de bord : *val*(*exp*$_2$) est stockée dans la case mémoire désignée par *exp*$_1$.

Par exemple, si `x` et `y` sont des variables telles que *type*(`x`) = *type*(`y`) = `int` et *val*(`y`) = 5, l'expression :

```
(x = y) > 2
```

a la valeur 5 et a pour effet de bord d'affecter la valeur 5 à `x`.

Attention ! il ne faut pas confondre l'opérateur d'affectation = avec l'opérateur d'égalité ==.

Signalons enfin deux abréviations classiques. Si *ident* est un nom de variable numérique, alors :

```
ident++ ≡ ident = ident + 1
ident-- ≡ ident = ident - 1
```

3.4 Conversion de type

On distingue :

- les **conversions explicites** réalisées par l'opérateur de "cast",
- les **conversions implicites** réalisées avant l'application d'opérateurs dont les opérandes ne sont pas du type attendu.

3.4.1 Conversion explicite : opérateur de "cast"

Si *T* est un nom de type et *exp* est une expression dont la valeur est convertible en *T*, alors :

```
(T) exp
```

est une expression telle que :

- *type*((*T*) *exp*) = *T*.
- *val*((*T*) *exp*) = *val*(*exp*) convertie en *T*.

3.4.1.1 Conversion entier - entier

Les types entiers sont ordonnés de la façon suivante :

```
char < unsigned char < short < unsigned short < long < unsigned long
```

Soit deux types entiers T_1 et T_2 et une valeur *v* de type T_1. La conversion de *v* en une valeur de type T_2 est réalisée :

- sans perte, si $T_1 \leq T_2$ ou si $T_1 > T_2$ et que *v* est inférieure ou égale à la plus grande valeur représentable en T_2.
- avec troncature si $T_1 > T_2$ et que *v* est supérieure à la plus grande valeur représentable en T_2.

3.4.1.2 Conversion entier - flottant et flottant - entier

- La conversion d'une valeur *v* de type entier en une valeur de type flottant produit le flottant le plus proche de *v*.
- La conversion d'une valeur *v* de type flottant en une valeur de type entier produit la partie entière de *v*.

3.4.2 Conversion implicite

3.4.2.1 Opérations arithmétiques et opérations de comparaison

Si *op* est un opérateur arithmétique, l'expression :

exp_1 *op* exp_2

est évaluée de la façon suivante :

1. Les opérandes sont évalués.
2. Si l'un des deux opérandes est de type `char`, `unsigned char`, `short` ou `unsigned short` il est tout d'abord converti en `int`, si c'est possible ou en `unsigned int` sinon. C'est la **promotion entière**.
3. Les types restants sont ordonnés de la façon suivante :

   `int < unsigned int < long < unsigned long < float < double`

   Si l'un des opérandes est de type T_1 et l'autre de type T_2 et que $T_1 < T_2$ alors l'opérande de type T_1 est converti en T_2, puis l'opérateur est appliqué fournissant une valeur de type T_2.

3.4.2.2 Affectation

L'expression :

exp_1 `=` exp_2

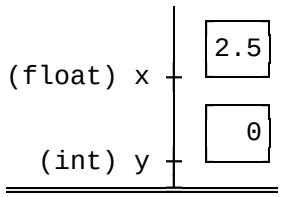
où exp_1 est une expression de type T_1 et exp_2 est une expression de type T_2, est évaluée de la façon suivante :

1. L'expression exp_2 est évaluée, puis sa valeur est convertie en T_1 si $T_2 \neq T_1$.
2. Cette valeur est affectée à la case mémoire désignée par exp_1.

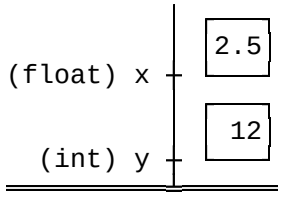
Par exemple, l'évaluation de l'expression :

```
y = 5.0 + 3 * x
```

dans l'état :



produit la valeur `12` de type `int` et le nouvel état :



3.5 Priorité et associativité des opérateurs

La priorité des opérateurs est utilisée lors de l'évaluation d'expressions dans lesquelles l'ordre des opérations n'a pas été explicitement indiquée par des paires de parenthèses.

Le tableau suivant indique la priorité et l'associativité des opérateurs qui ont été ou seront étudiés dans ce cours.

Opérateurs	Priorité	Associativité
`() [] .`	15	à droite
`! -`$_{\text{unaire}}$ `&`$_{\text{unaire}}$ `*`$_{\text{unaire}}$`(`T`)`	14	à gauche
`/ %`	13	à gauche
`+ -`$_{\text{binaire}}$	12	à gauche
`< <= > >=`	10	à gauche
`== !=`	9	à gauche
`&&`	5	à gauche
`||`	4	à gauche
`=`	2	à droite

Si op_1 et op_2 sont des opérateurs infixes, on a :

- $x\ op_1\ y\ op_2\ z \equiv (x\ op_1\ y)\ op_2\ z$, si la priorité de op_1 est supérieure à la priorité de op_2 ou si la priorité de op_1 est égale à la priorité de op_2 et que op_1 est associatif à gauche.
- $x\ op_1\ y\ op_2\ z \equiv x\ op_1\ (y\ op_2\ z)$, sinon.

Si op_1 est un opérateur préfixe et op_2 est un opérateur infixe, on a :

- $op_1\ x\ op_2\ y \equiv (op_1\ x)\ op_2\ y$, si la priorité de op_1 est supérieure à la priorité de op_2 ou bien si la priorité de op_1 est égale à la priorité de op_2 et que op_1 est associatif à gauche.
- $op_1\ x\ op_2\ y \equiv op_1\ (x\ op_2\ y)$, sinon.

Par exemple :

- `6 + 3 * 4` ≡ `6 + (3 * 4)`, car la priorité de `*` est supérieure à celle de `+`.
- `6 * 3 + 4` ≡ `(6 * 3) + 4`, pour la même raison.
- `!a && b || x > 3`
  ≡ `(!a) && b || x > 3`, car la priorité de `!` est supérieure à celle de `&&`,
  ≡ `((!a) && b) || x > 3`, car la priorité de `&&` est supérieure à celle de `||`,
  ≡ `(((!a) && b) || (x > 3))`, car la priorité de `>` est supérieure à celle de `||`.
- `3 + 4 - 5` ≡ `(3 + 4) - 5`, car `+` et `-` ont la même priorité et `+` est associatif à gauche.
- `x = y = 5` ≡ `x = (y = 5)`, car `=` est associatif à droite.

3.6 Ordre d'évaluation des opérandes

L'ordre d'évaluation des opérandes n'est pas spécifié par la norme ANSI excepté pour les opérateurs `||` et `&&` (voir ci-dessus §3.3.5). Il faut en tenir compte lorsque ces opérandes contiennent des opérateurs à effet de bord, car l'ordre dans lequel se produiront ces effets est indéterminé.

4

Instructions

4.1 Qu'est-ce qu'une instruction

Une instruction est un ordre donné à l'ordinateur de réaliser une suite d'actions dont chacune a pour effet de modifier le déroulement du programme, son environnement ou son état.

Les instructions offertes par C sont les suivantes :

- instruction vide,
- instruction « expression »,
- bloc,
- conditionnelle,
- itération,
- choix multiple,
- interruption d'une instruction de contrôle,
- retour de fonction.

4.2 Présentation des instructions

Chaque instruction sera définie par sa syntaxe et par l'action que déclenche son exécution.

4.2.1 Instruction vide

L'instruction :

```
;
```

est une instruction vide qui ne réalise aucune action.

4.2.2 Instruction "expression"

Si *exp* est une expression, alors

```
exp;
```

est une instruction qui réalise l'effet de bord de l'expression *exp*. Une telle instruction n'a donc de sens que si cet effet de bord n'est pas nul.

Par exemple :

```
12 + 4;
```

est une instruction qui ne déclenche aucune action. Elle est équivalente à une instruction vide. Par contre, l'instruction :

```
x = 12 + 4;
```

en déclenche une : affecter 16 à `x`.

4.2.3 Bloc d'instructions

Si $decl_1$, …, $decl_m$ sont des déclarations de variables et $inst_1$, …, $inst_n$ sont des instructions, alors :

```
{
decl1
…
declm
inst1
…
instn
}
```

est une instruction appelé **bloc d'instructions** ou plus simplement **bloc**.

Un bloc est composé d'une suite, éventuellement vide, de déclarations, suivie d'une suite d'instructions qui peuvent elles-mêmes être des blocs d'instructions.

Les variables déclarées dans un bloc sont des **variables locales** à ce bloc. Elles sont visibles dans ce bloc et dans tous les blocs englobés qui ne les redéfinissent pas. La valeur initiale d'une variable locale non initialisée dans sa déclaration, est indéterminée.

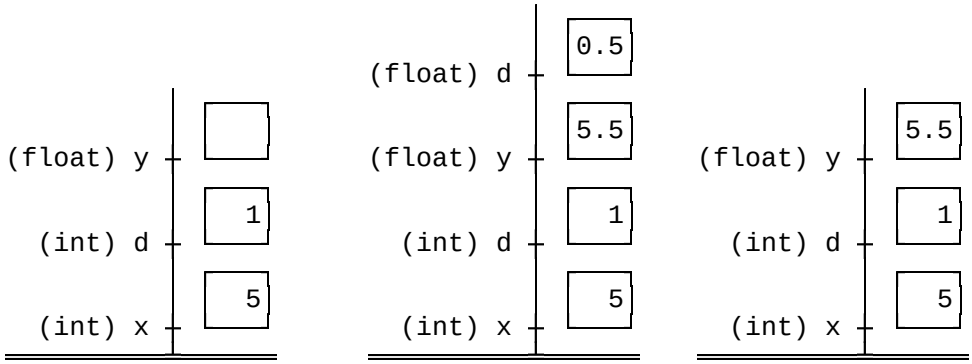
L'exécution d'un bloc d'instructions se déroule de la façon suivante :

1. Les données déclarées dans le bloc sont empilées au sommet de l'environnement dans l'ordre de leur déclaration.
2. Les instructions sont exécutées séquentiellement jusqu'à la sortie du bloc déclenchée :
   - soit parce que la fin du bloc a été atteinte,
   - soit par l'exécution de l'instruction `break` ou `return` (voir ci-dessous §4.2.7 et §4.2.8).
3. A la sortie du bloc, les variables locales à ce bloc sont enlevées de la pile. La sortie du bloc de la fonction `main` provoque l'arrêt du programme.

Considérons par exemple le bloc suivant :

```
{
int x = 5, d = 1;
float y;
   {
   float d = 0.5;
   y = x + d;
   }
…
}
```

Les déclarations de `x` et `y` dans le bloc externe sont visibles dans le bloc interne. La déclaration de `d` dans le bloc interne masque celle du bloc externe : `d` est un entier dans le bloc externe et un flottant dans le bloc interne. Notons que la valeur initiale de `y` est indéterminée. L'évolution de l'état du programme lors de l'exécution du bloc interne est le suivant :



Immédiatement avant l'exécution du bloc interne

Immédiatement avant la sortie du bloc interne.

Immédiatement après la sortie du bloc interne

4.2.4 Instruction conditionnelle

Une **instruction conditionnelle** permet de choisir l'instruction à exécuter parmi deux possibles, en fonction de la valeur d'une expression booléenne de choix.

Si *exp* est une expression et *inst*, $inst_1$ et $inst_2$ sont des instructions alors

```
if (exp)
   inst
```

```
if (exp)
   inst1
else
   inst2
```

sont des instructions conditionnelles.

La première réalise l'action :

si val(*exp*) ≠ `0` alors
   exécuter l'instruction *inst*
fin-si

et la seconde l'action :

si *val*(*exp*) ≠ `0` alors
   exécuter l'instruction $inst_1$
sinon
   exécuter l'instruction $inst_2$
fin-si

Par exemple :

```
if (mois >= 4 && mois < 10)
   saison = "chaude"
else
   saison = "froide"
```

Lorsque des instructions conditionnelles sont imbriquées, la règle suivante s'applique : chaque clause `else` se rapporte au dernier `if` ayant une condition suivie d'exactement une instruction. En application de cette règle, dans l'instruction :

```
if (cond1)
   if (cond2)
      inst21
   else
      inst22
```

la clause `else` est celle du `if` interne. Cette règle a pour conséquence qu'un `if` sans clause `else` imbriqué dans la clause `then` d'un `if` avec clause `else`, devra être inséré dans un bloc, pour éviter toute ambiguité. Dans l'instruction :

```
if (cond1)
   {
   if (cond2)
      inst21
   }
else
   inst12
```

la clause `else` se rapporte bien au `if` externe, en application de la règle d'imbrication des instructions conditionnelles. On aurait pu aussi ajouter au `if` interne une clause `else` vide :

```
if (cond1)
   if (cond2)
      inst21
   else
      ;
else
   inst12
```

4.2.5 Itération

Une instruction d'itération, permet de répéter l'exécution d'une instruction tant qu'une condition est vérifiée. Par exemple, ajouter 2 à la valeur d'une variable tant que cette valeur est inférieure à 100.

Trois instructions d'itération sont disponibles : `while`, `do…while` et `for`.

4.2.5.1 while

Si *exp* est une expression et *inst* est une instruction alors :

```
while (exp)
   inst
```

est une instruction d'**itération** (on dit aussi une **boucle**) dans laquelle *exp* est le **test de continuation** et *inst*, le **corps** de la boucle, est une instruction à répéter.

L'action réalisée est la suivante :

boucle :
   si $val(exp) \neq 0$ alors
      exécuter *inst*
      aller-à boucle
   fin-si

Le corps de la boucle est exécuté tant que le test de continuation est vrai. Il peut donc ne jamais être exécuté.

Une instruction `while` doit être précédée d'une initialisation des variables dont dépend le test de continuation et dont les valeurs évolueront à chaque exécution du corps de la boucle.

Par exemple, l'instruction suivante calcule la somme `s` des entiers de 1 à 9.

```
{
int s, i;
s = 0;
i = 1;
while (i <= 9)
   {
   s = s + i;
   i = i + 1;
   }
}
```

4.2.5.2 do…while

Si *inst* est une instruction et *exp* est une expression alors :

```
do
   inst
while (exp);
```

est une instruction d'itération. Comme dans l'instruction `while`, *inst* est le corps de la boucle et *exp* est le test de continuation.

L'action réalisée est la suivante :

boucle :
   exécuter *inst*
   si $val(exp) \neq 0$ alors
      aller-à boucle
   fin-si

Le corps de la boucle est exécuté jusqu'à ce que le test de continuation soit faux. Il est donc exécuté au moins une fois.

De même qu'une instruction `while`, une instruction `do…while` doit être précédée d'une initialisation des variables dont dépend le test de continuation et dont les valeurs évolueront à chaque exécution du corps de la boucle.

Par exemple, l'instruction suivante calcule le plus petit entier *n* tel que la somme des entiers de 1 à *n* soit supérieure à 50.

```
{
int s, n;
s = 0;
n = 0;
do
   {
   n = n + 1;
   s = s + n;
   }
while(s <= 50);
}
```

4.2.5.3 for

Si exp_1, exp_2, exp_3 sont des expressions et *inst* est une instruction, alors

```
for (exp1; exp2; exp3)
   inst
```

est une instruction d'itération équivalente, par définition, à :

```
exp1;
while (exp2)
   {
   inst
   exp3;
   }
```

Cette instruction intègre donc l'initialisation (exp_1) et l'évolution (exp_2) de la variable dont dépend le test de continuation (exp_3).

Par exemple, l'instruction suivante calcule la somme *s* des 10 premiers entiers.

```
{
int s, n;
s = 0;
for (n = 1; n <= 10; n = n + 1)
   s = s + n;
}
```

Les expressions exp_1 et exp_3 peuvent être absentes. Le test de continuation (exp_2) peut lui-aussi être absent, on considère alors qu'il est toujours vrai. Par exemple, l'instruction :

```
for (;;)
   inst
```

est une boucle infinie.

4.2.6 Choix multiple

Une instruction de **choix multiple** permet de choisir une instruction à réaliser parmi un ensemble d'instructions possibles, en fonction de la valeur d'une expression de choix.

Une instruction de choix multiple a la forme suivante :

```
switch (exp-choix)
   {
   inst-switch1
   …
   inst-switchn
   }
```

où :

- *exp-choix* est une expression.
- Le bloc constitue le **corps** de l'instruction.
- Chaque *inst-switch* a l'une des trois formes suivantes :

  `case` *exp-cas*`:` *inst-switch*

  `default:` *inst-switch*

  inst

  où `case` *exp-cas*`:` et `default:` sont des étiquettes de cas, *exp-cas* est une expression constante et *inst* est une instruction.

L'action réalisée par une instruction `switch` est la suivante :

si dans le corps il existe une étiquette `case` *exp-cas*`:` telle que *val*(*exp-cas*) = *val*(*exp-choix*) alors
    se brancher à l'instruction qui suit immédiatement cette étiquette.
sinon
    si dans le corps il existe une étiquette `default:` alors
        se brancher à l'instruction qui suit immédiatement cette étiquette.
    sinon
        ne rien faire.
    fin-si
fin-si

Une fois qu'un branchement a été effectué, les instructions sont exécutées en séquence jusqu'à la dernière instruction du corps. Pour éviter d'enchaîner l'exécution de deux cas exclusifs consécutifs, il faut terminer le premier par une instruction `break` (voir ci-dessous §4.2.7) qui a pour effet d'abandonner l'exécution de l'instruction.

Par exemple, l'instruction suivante calcule le nom du jour (`jour`) correspondant à un numéro de jour (`num_jour`) dans la semaine :

```
switch(num_jour)
   {
   case 1:
      jour = "lundi";
      break;
   case 2:
      jour = "mardi";
      break;
   …
   case 7:
      jour = "dimanche";
      break;
   default:
      printf("Il n'y a que 7 jours dans la semaine !");
   }
```

Il est possible de définir des cas à plusieurs étiquettes. Par exemple, l'instruction suivante détermine si un jour de la semaine est un jour d'activité ou de repos :

```
switch(num_jour)
   {
   case 6:
   case 7:
      activite = "repos";
      break;
   default:
      activite = "travail";
}
```

4.2.7 Rupture de séquence

L'instruction :

```
break;
```

provoque l'abandon de l'instruction en cours. Elle ne peut apparaître que dans le corps d'une instruction `while`, `do`, `for` ou `switch`.

4.2.8 Retour d'une fonction

Si *exp* est une instruction alors

`return` *exp*`;`

est une instruction qui provoque, comme nous le verrons au chapitre 6, l'abandon de l'exécution du bloc qui constitue le corps d'une fonction et retourne la valeur *val*(*exp*) de l'application de cette fonction.

5

Valeurs composées

5.1 Qu'est-ce qu'une valeur composée ?

Une valeur composée est une agrégation de données dont les valeurs peuvent être simples (des nombres) ou composées.

En C on distingue :

- Les **structures** qui sont composées de données de différents types.
- Les **tableaux** qui sont composées de données de même type.

5.2 Structures

Une structure est une suite contiguë de données de différents types qui constituent les **champs** de cette structure.

5.2.1 Déclaration d'une structure

Si *ident* est un identificateur et $decl_1$, …, $decl_n$ sont des déclarations de variables, alors

```
struct ident
   {
   decl1;
   …
   declm;
   }
```

est la déclaration d'une structure de nom *ident* composée des variables déclarées par $decl_1$, …, $decl_n$.

Par exemple, la structure déclarée par :

```
struct point
   {
   char nom;
   float x, y;
   };
```

a :

- le nom `point`,
- les champs `nom`, `x` et `y`, de types respectifs `char`, `float` et `float`.

Si *ident* est le nom d'une structure, $ident_1$, …, $ident_n$ sont des noms de variable, exp_{11},…, exp_{1k}, …, exp_{n1},…, exp_{nk} sont des expressions constantes, alors

`struct` *ident* *ident*$_1$ `=` {*exp*$_{11}$`,…,` *exp*$_{1k}$`}, …,` *ident*$_n$ `=` {*exp*$_{n1}$`,…,` *exp*$_{nk}$`};`

est la déclaration de *n* variables de type `struct` *ident* dont les champs sont initialisés respectivement avec les valeurs (*val*(*exp*$_{11}$),…, *val*(*exp*$_{1k}$)), …, (*val*(*exp*$_{n1}$),…, *val*(*exp*$_{nk}$)).

Par exemple :

```
struct point p1 = {'P', 1.5, 2.5}, p2 = {'Q', -1.5, -2.5};
```

déclare deux variables `p1` et `p2` de type `struct point` qui ont pour valeur les points *P* et *Q* de coordonnées respectives (1,5 ; 2,5) et (-1,5 ; -2,5).

L'initialisation est facultative ou peut être incomplète. Dans ce dernier cas nous considérerons que les champs non initialisés ont une valeur initiale indéterminée.

On peut regrouper la déclaration d'une structure avec celle des variables ayant cette structure pour type. Par exemple :

```
struct point
  {
  char nom;
  float x, y;
  } p1, p2;
```

On peut aussi ne pas nommer une structure. Par exemple :

```
struct
  {
  float x, y;
  } p1, p2;
```

Les structures peuvent être imbriquées. Par exemple, les dates de vacances scolaires peuvent être représentées par les deux structures suivantes :

```
struct date
  {
  int jour, mois, annee;
  }

struct vacances
  {
  char periode;
  char zone;
  struct date debut;
  struct date fin;
  } v;
```

La période est codée : 1 pour les vacances de la Toussaint, 2 pour celles de Noël, 3 pour celles d'hiver, 4 pour celles de printemps et 5 pour celles d'été.

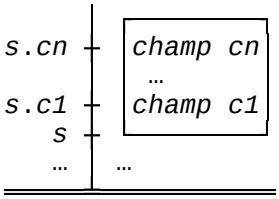
Une valeur possible de la variable `v` pourrait être le quadruplet :

`(3, 'A', (3, 2, 2001), (18, 2, 2001))`

qui représente les dates des vacances d'hiver de l'année scolaire 2000-2001 pour la zone A.

Si une structure est la valeur d'une donnée de nom *s* alors chaque champ *c* de cette structure est une donnée de nom *s*`.`*c*. Par exemple, `v.date.jour` est le nom de la donnée dont la valeur est le `jour` de la `date` des vacances `v`.

L'environnement ajouté par la déclaration d'une structure de champs c_1, …, c_n qui est la valeur d'une donnée de nom *s* est donc suivant :



5.2.2 Valeurs d'un champ

L'extraction de la valeur d'un champ d'une structure est réalisée par l'opérateur . (point).

Si *exp* est une expression qui est le nom d'une structure (une valeur gauche) et *ident* est un nom de champ de cette structure, alors :

exp.*ident*

est une expression telle que :

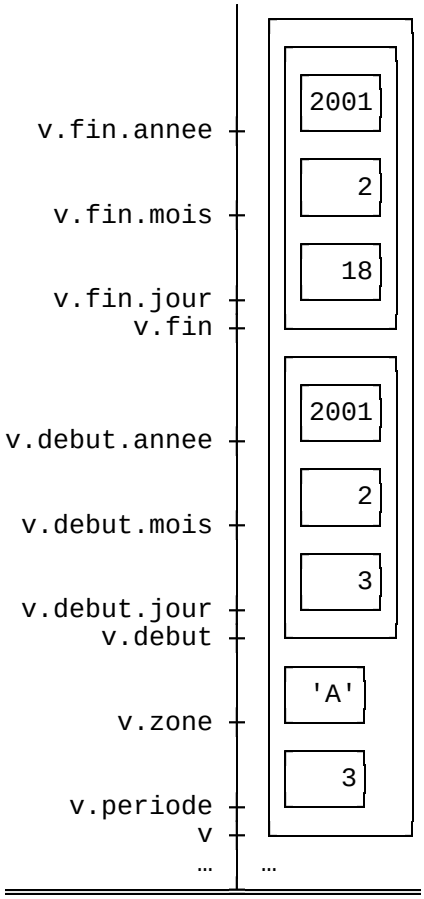
- *type*(*exp*.*ident*) = type du champ *ident* de la structure *val*(*exp*),
- *val*(*exp*.*ident*) = valeur du champ *ident* de la structure *val*(*exp*),

L'expression *exp*.*ident* est une valeur gauche si elle est le nom d'une donnée modifiable.

Considérons, par exemple, la structure déclarée par :

```
struct vacances v = {2, 'A', {3, 2, 2001}, {18, 2, 2001}}
```

Les noms désignant les cases mémoire associées à chacun des champs apparaissent très clairement sur le dessin de l'environnement ajouté par cette déclaration.



On a :

val(`v.zone`) = `'A'`
val(`v.debut.mois`) = `2`

5.2.3 Affectation d'une valeur à un champ

L'affectation d'une valeur à un champ est réalisée par l'opérateur = (voir ci-dessus §3.3.6). Par exemple :

```
struct vacances v;

v.periode = 3;
v.zone = 'A';
v.debut.jour = 3;
v.debut.mois = 2;
v.debut.annee = 2001;
v.fin.jour = 18;
v.fin.mois = 2;
v.fin.annee = 2001;
```

5.3 Tableaux

En C, un tableau est une suite contiguë de données de même type qui constituent les **éléments** du tableau.

5.3.1 Déclaration d'un tableau

Si *T* est un type, *ident* est un identificateur, *n* est un entier littéral, exp_0, …, exp_{n-1} sont des expressions constantes de type *T*, alors :

T ident`[n] = {`exp_0`, …, `exp_{n-1}`};`

est la déclaration d'un tableau, de nom *ident*, composé de *n* éléments de type *T*. Ces éléments sont numérotés de 0 à *n* - 1 (et non de 1 à *n*, **attention !**). Ils sont initialisés respectivement avec les valeurs *val*(exp_0), …, *val*(exp_{n-1}).

En réalité l'identificateur *ident* n'est pas le nom du tableau mais celui d'une constante dont la valeur est l'adresse du premier élément (rang 0) de ce tableau. L'expression *ident* n'est donc pas une valeur gauche. On ne pourra donc pas affecter un tableau à un autre. En C, les tableaux ne sont pas des valeurs à part entière, comme le sont les nombres et les structures.

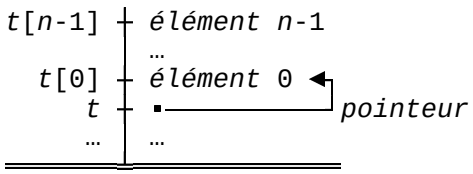
L'initialisation est facultative ou peut être incomplète. Dans ce dernier cas nous considérerons que les cases non initialisées ont une valeur initiale indéterminée.

Par exemple, la hauteur moyenne, mesurée en nombre de millimètres, de pluie tombée chaque mois en un lieu donné pourrait être enregistrée dans le tableau suivant :

```
int pluie[12];
```

Si un tableau est la valeur d'une donnée de nom *t* alors chaque élément *i* de ce tableau est une donnée de nom *t*`[`*i*`]`.

L'environnement ajouté par la déclaration d'un tableau à *n* éléments qui est la valeur d'une donnée de nom *t*, est le suivant :



5.3.2 Valeur d'un élément

L'extraction de la valeur d'un élément d'un tableau est réalisé par l'opérateur `[]`.

Si exp_1 est une expression de type tableau de type *T* et exp_2 est une expression de type entier, alors

exp_1`[`exp_2`]`

est une expression telle que :

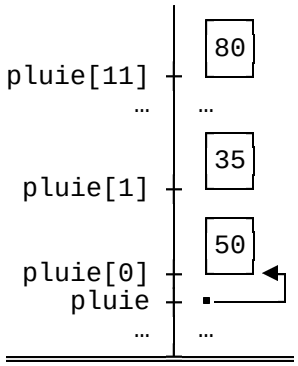
- *type*(exp_1`[`exp_2`]`) = *T*,
- *val*(exp_1`[`exp_2`]`) = valeur de l'élément de rang *val*(exp_2) du tableau *val*(exp_1),

L'expression exp_1`[`exp_2`]` est une valeur gauche si elle est le nom d'une donnée modifiable.

Considérons, par exemple, le tableau déclaré par :

```
int pluie[12] = {50, 35, …, 80}
```

Les noms des cases mémoire associées à chacun des éléments apparaissent très clairement sur le dessin de l'environnement ajouté par cette déclaration.



On a :

val(`pluie[1]`) = `35`

5.3.3 Affectation d'une valeur à un élément

L'affectation d'une valeur à un élément d'un tableau est réalisée par l'opérateur = (voir ci-dessus §3.3.6). Par exemple :

```
int pluie[12]

pluie[0] = 50;
pluie[1] = 35;

…
pluie[11] = 80;
```

5.3.4 Tableaux multidimensionnels

Les éléments d'un tableau peuvent être eux-mêmes des tableaux. La déclaration d'un tableau multidimensionnel est une généralisation de celle d'un tableau mono-dimensionnel.

Si *T* est un type, *ident* est un identificateur et n_1, … , n_k sont des entiers littéraux, alors

T ident`[`n_1`][`n_2`]…[`n_k`]`

est la déclaration d'un tableau de nom *ident*, composé de n_1 tableaux de dimension `[`n_2`]…[`n_k`]` et ainsi de suite.

L'élément de coordonnées (i_1, …, i_k) est la valeur de l'expression :

ident`[`i_1`]…[`i_k`]`

Pour l'initialisation on énumère les valeurs des éléments en faisant varier d'abord l'indice le plus profond.

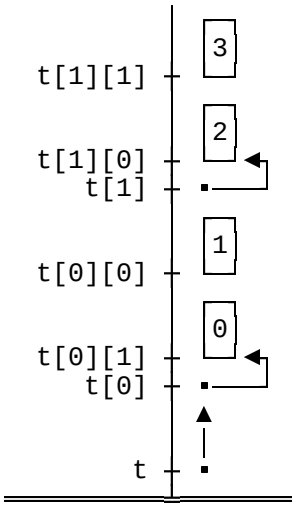
Par exemple, le tableau d'entiers suivant :

0	1
2	3
4	5
6	7
8	9

pourra être déclaré :

```
int t[5][2] = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}}
```

L'environnement produit par cette déclaration est le suivant :



On a :

val(`t[1]`) = adresse du sous-tableau de rang 1 de `t` (2e ligne de `t`),
val(`t[1][0]`) = `2`

6

Fonctions

6.1 Qu'est-ce qu'une fonction

Tout langage de programmation offre le moyen de découper un programme en unités indépendantes qui peuvent s'appeler les unes les autres.

En C, ces unités sont les **fonctions**.

Rappelons qu'en mathématiques une fonction est une correspondance entre un ensemble de départ et un ensemble d'arrivée.

En C, une fonction peut avoir un ou plusieurs arguments de type T_1, …, T_n et retourne une valeur de type T. Une fonction est donc une correspondance :

$$ext(T_1) \times \ldots \times ext(T_n) \rightarrow ext(T)$$

où × est le produit cartésien et pour tout type t, $ext(t)$ désigne l'ensemble des valeurs de type t.

6.2 Définition d'une fonction

La définition d'une fonction a la syntaxe suivante :

T ident (*decl*$_1$, …, *decl*$_n$)
bloc

où :

- La première ligne est appelée **en-tête** de la fonction.
- Les *decl*$_i$ (pour i de 1 à *n*) sont les déclarations des **arguments formels** de la fonction, c'est à dire les variables auxquelles seront affectées les **arguments effectifs** de la fonction au moment de son appel.
- *T* est le type d'arrivée de la fonction.
- *bloc* est un bloc d'instructions, appelé **corps** de la fonction, dont l'exécution calcule la valeur de la fonction ainsi que ses effets de bord.

Dans le corps de la fonction on doit trouver une ou plusieurs instructions :

```
return exp;
```

dont l'exécution a pour effet de renvoyer à la fonction appelante la valeur *val*(*exp*).

Par exemple, la fonction qui calcule la moyenne de deux entiers peut être définie de la façon suivante :

```
float moyenne(int x1, int x2)
  {
  float m;
  m = (x1 + x2) / 2.0;
  return m
  };
```

Autre exemple, la fonction qui calcule le plus grand de deux entiers (ou l'un des deux, s'ils sont égaux) peut être définie de la façon suivante :

```
int max(int x, int y)
  {
  if (x >= y)
    return x;
  then
    return y;
  }
```

6.3 Appel d'une fonction

L'appel d'une fonction est une expression ayant la forme suivante :

ident(*exp*$_1$,…, *exp*$_n$)

où l'identificateur *ident* est le nom de la fonction, un identificateur et *exp*$_1$, …, *exp*$_n$ sont des expressions dont les valeurs constituent les **arguments effectifs** de la fonction.

Si *f* est le nom d'une fonction dont les arguments formels sont les variables x_1, …, x_n, de type T_1, …, T_n et dont le corps est le bloc *b*, alors la valeur de l'expression *ident*(*exp*$_1$,…, *exp*$_n$) est calculée de la façon suivante :

1. Les valeurs des arguments effectifs sont calculées :

   $v_1 = val(exp_1) \ldots v_n = val(exp_n)$

2. Le bloc :

   ```
   {
   T1 x1
   …
   Tn xn
   x1 = v1;
   …
   xn = vn;
   b
   }
   ```

   est exécuté.

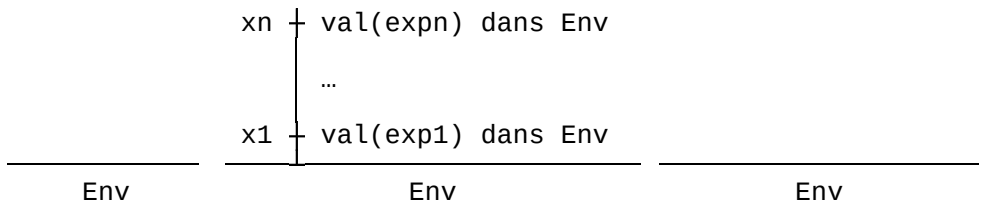
3. La valeur retournée est celle produite par la première instruction `return` *exp*`;` exécutée. On a donc :

   $val(f(exp_1, \ldots, exp_n)) = val(exp)$

L'environnement produit par un appel de fonction :

f(*exp*$_1$, …, *exp*$_n$)

est le suivant :

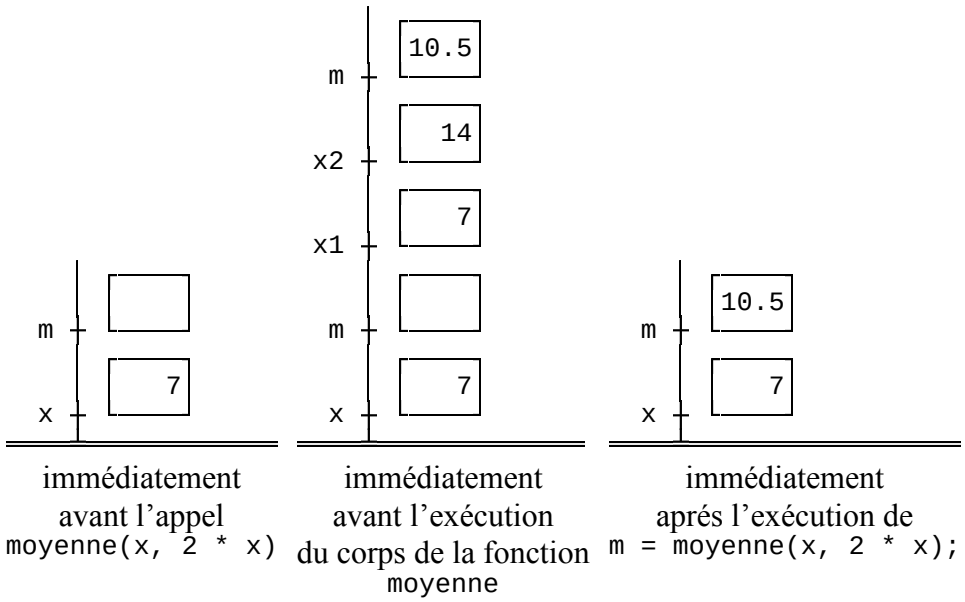


immédiatement avant l'appel de la fonction *f*	immédiatement avant l'exécution du corps de la fonction *f*	immédiatement aprés l'exécution du corps de la fonction *f*

Par exemple, l'appel de la fonction `moyenne` définie ci-dessus au §6.2 :

```
{
int x;
float m;
x = 7;
m = moyenne(x, 2 * x);
}
```

L'évolution de l'environnement est le suivant :



immédiatement avant l'appel `moyenne(x, 2 * x)`	immédiatement avant l'exécution du corps de la fonction `moyenne`	immédiatement aprés l'exécution de `m = moyenne(x, 2 * x);`

On constate que la valeur des arguments effectifs (ici 7 et 14) est copiée dans les cases mémoire associées aux arguments formels : on dit que les arguments sont **passés par valeur**.

6.4 Cas particuliers

Une fonction peut ne pas avoir de valeur de retour. Elle est alors déclarée :

`void` *ident*`(`$decl_1$`, …,` $decl_n$`)`

où `void` est le type vide.

Par exemple :

```
void ecrire_entier(int n)
  {
  printf("%d\n", n);
  }
```

Une fonction peut ne pas avoir d'arguments Elle est alors déclarée :

`void` *ident*`(v)`

où `void` est le type vide.

Par exemple, la fonction suivante lit un entier sur l'unité d'entrée :

```
int lire_entier(void)
  {
  int n;
  scanf("%d", n);
  return n;
}
```

Les deux fonctions précédentes peuvent être combinées pour afficher l'entier lu sur l'unité d'entrée :

```
ecrire_entier(lire_entier());
```

Les fonctions de lecture et d'écriture `scanf` et `printf` sont présentées au chapitre suivant.

La fonction `main` a pour type d'arrivée `int` et peut avoir des arguments. Lorsque cette fonction n'a pas d'arguments et ne renvoie pas de valeur il est traditionnel d'écrire son entête :

```
main()
```

C'est ce que nous ferons dans ce cours.

7

Entrées-sorties

En C les opérations d'entrées-sorties (saisie au clavier, affichage à l'écran, lecture et écriture dans un fichier, etc.) sont réalisées par des fonctions prédéfinies. Ces fonctions traitent les données qu'elles lisent ou écrivent comme des **flots** ou des **fichiers** d'octets qui représentent soit des caractères, soit des mots de la mémoire. Dans le premier cas les fichiers sont dits **fichiers textuels** et dans le second ils sont dits **fichiers binaires**. Par exemple, le texte source d'un programme C est enregistré dans un fichier textuel et le résultat de sa compilation dans un fichier binaire.

Ces fichiers sont associés aux unités d'entrée-sortie qui peuvent être un clavier, un écran, une imprimante, un disque, etc. Il existe trois unités standards d'entrée-sortie : l'**unité standard d'entrée** qui est habituellement le clavier du poste de travail, l'**unité standard de sortie** qui est habituellement l'écran du poste de travail et l'**unité standard de messages d'erreur** qui est habituellement, elle-même, l'écran du poste de travail.

Dans ce cours nous n'étudierons que deux des fonctions d'entrée-sortie : les fonctions `scanf` et `printf` qui permettent de lire ou d'écrire des données textuelles avec format sur les unités d'entrée et de sortie standards.

Le code machine des fonctions `scanf` et `printf` est contenu dans le module de la bibliothèque standard de C, qui contient l'ensemble des fonctions prédéfinies utilisable pour réaliser les opérations d'entrée-sortie. Ce code machine est inséré dans le fichier exécutable du programme lors de la phase d'**édition de liens**, qui suit la phase de compilation.

Puisque ces fonctions ne sont pas définies dans le programme lui-même, le compilateur doit connaître le type des arguments et de la valeur de chacune de ces fonctions, afin de pouvoir vérifier que le programme est correctement typé. La bibliothèque standard contient donc aussi pour chaque module un **fichier d'en-têtes** qui contient les déclarations des fonctions du module, c'est à dire leurs en-têtes. Pour le module des fonctions d'entrée-sortie, le fichier d'en-tête est « stdio.h ». Ce fichier doit être inclus dans le fichier source de tout programme qui utilise au moins une fonction prédéfinie d'entrée-sortie. Ceci est réalisé en plaçant au début du programme la directive :

```
#include <stdio.h>
```

qui indique au **pré-processeur** mis en œuvre avant le compilateur d'inclure, à cet endroit, dans le texte source du programme, le fichier d'en-tête « stdio.h ».

7.1 Lecture de données : la fonction `scanf`

La fonction `scanf` lit des données sur l'unité d'entrée standard selon un format spécifié en argument. Cette fonction constitue un véritable analyseur lexical, mais elle est assez difficile à

manipuler. Nous l'utiliserons sous la forme simplifiée suivante, qui permet de lire une valeur, selon un format spécifié et de l'enregistrer dans une case mémoire :

`scanf(`*spécification-de-format*`, &`*nom-de-donnée-modifiable*`)`

L'expression `&`*nom-de-donnée-modifiable* indique le nom de la case mémoire dans laquelle sera rangée la donnée lue. Plus exactement, comme nous le verrons au chapitre 9, elle a pour valeur l'adresse de cette case mémoire.

La spécification de format indique le type de la donnée à saisir. Elle est composée du caractère `%` suivi d'une lettre qui indique le type de la donnée à lire :

`d` pour un entier de type `int`,

`ld` devant un entier de type `long` (dans le cas où le type int est équivalent au type `short`),

`u` pour un entier de type `unsigned int`,

`lu` devant un entier de type `unsigned long` (dans le cas où le type int est équivalent au type `short`),

`c` pour un caractère (entier de type `char`),

`s` pour une chaîne de caractères terminée par un caractère d'espacement (blanc, tabulation, fin de ligne) qui n'en fait pas partie ; un '\0' est rajoutée automatiquement à la fin de la chaîne (voir ci-dessous chapitre 10).

`f` pour un flottant de type `float`,

`lf` pour un flottant de type `double`,

`Lf` pour un flottant de type `long double`.

Par exemple, l'exécution du bloc suivant permet de lire les valeurs de deux variables `longueur` et `largeur` de type `float`.

```
{
float longueur, largeur;
scanf(%f, &longueur);
scanf(%f, &largeur);
}
```

7.2 Ecriture de données : la fonction `printf`

La fonction `printf` écrit des données sur l'unité d'entrée standard selon un format spécifié en argument. L'affichage de données est réalisé par la fonction `printf`, dont l'appel a la forme suivante :

`printf(`*format*`, `exp_1`, …, `exp_n`)`

Le format est une chaîne de caractères dans laquelle sont inclus autant de spécifications de format qu'il y a d'expressions. Une spécification de format est préfixée par le caractère `%` suivie d'un ou plusieurs éléments que nous préciserons ci-dessous.

Les valeurs à afficher sont celles des expressions exp_1, …, exp_n.

L'affichage se déroule de la façon suivante :

1. Le caractère courant est le premier du format et l'expression courante est la première de la liste des expressions (exp_1, …, exp_n).
2. Si le caractère courant est différent de `%`, il est affiché. Si la fin du format n'a pas été atteinte, on passe au caractère suivant dans le format et l'on retourne en 2.

3. Si le caractère courant est %, la valeur de l'expression courante est affichée conformément à la spécification de format. Si la fin du format ou la fin de la liste des expressions n'a pas été atteinte, on passe au caractère qui suit la spécification dans le format et à l'expression suivante. Puis l'on retourne en 2.

Dans ce cours, nous utiliserons des spécifications de format simplifiées, composées :

- Obligatoirement du caractère %.
- Facultativement, d'un point (.) suivi d'un nombre qui indique le nombre de chiffres après la virgule pour l'affichage d'un flottant (6 par défaut).
- Obligatoirement un caractère de conversion suivant :

`d`	pour afficher un entier de type `int`,
`ld`	pour afficher un entier de type `long ou unsigned long` dans le cas où le type `int` est équivalent au type `short`,
`c`	pour afficher un caractère,
`s`	pour afficher une chaîne de caractères,
`f`	pour afficher un flottant en virgule fixe sans exposant de type `double`,
`Lf`	pour afficher un flottant en virgule fixe sans exposant de type `long double`,
`e`	pour afficher un flottant en virgule fixe avec exposant de type `double`,
`Le`	pour afficher un flottant en virgule flottante avec exposant de type `long double`.

Par exemple, l'exécution du bloc suivant :

```
{
int x, y;
printf("Entrez le premier nombre ?\n");
scanf("%d", &x);
printf("Entrez le second nombre ?\n");
scanf("%d", &y);
printf("La moyenne de %d et de %d est %f.", x, y, (x + y) / 2.0);
}
```

déclenche le dialogue suivant :

```
Entrez le premier nombre ?
12↵
Entrez le second nombre ?
3↵
La moyenne de 12 et de 3 est 7.500000.
```

L'écriture des cinq zéros dans `7.500000` provient du fait que par défaut, le nombre de chiffres après la virgule est 6. Ce nombre peut être changé en l'indiquant dans la spécification de format. Pour savoir comment le faire, consultez un manuel complet de C : un des deux livres cités dans l'introduction, par exemple.

8

Structure d'un programme C

Un programme C est contenu dans un ou plusieurs **fichiers sources**.

Un fichier source contient une suite d'éléments qui peuvent être :

- des directives pour le préprocesseur comme :

  ```
  #include <stdio.h>
  ```

  qui charge le fichier d'en-têtes qui contient la déclaration des fonctions d'entrée-sortie telles que `printf` ou `scanf` comme nous l'avons expliqué ci-dessus au chapitre 7.
- des déclarations de type,
- des déclarations de variables externes, c'est à dire des variables qui sont définies dans d'autres fichiers sources,
- des définitions de variables,
- des définitions de fonctions.

Dans ce cours nous n'étudierons que les programmes contenus dans un seul fichier source, sans déclarations externes et sans déclarations de type.

8.1 Structure d'un fichier source

Nous adopterons la structure simplifiée suivante.

dir_1
…
dir_m — Directives d'inclusion de fichiers en-tête.

v_1
…
…
v_m — Déclarations des variables globales :
les variables v_1, …, v_m sont visibles dans toutes les fonctions du programme.

f_1
…
…
f_n — Définitions des fonctions :
les fonctions f_1, …, f_i sont visibles dans le corps des fonctions f_i, …, f_n.

main — L'exécution d'un programme C démarre par l'appel de la fonction `main` :
les variables v_1, …, v_m et les fonctions f_1, …, f_n sont visibles dans son corps.

8.2 Visibilité des variables et des fonctions

Les règles de visibilité sont celles qui permettent d'associer un nom à sa déclaration. En C, une variable ou une fonction ne peut apparaître dans une expression que si elle a été préablement déclarée.

Dans un programme C, toutes les fonctions ont un nom différent. Par contre, il peut y avoir des variables différentes mais de même nom.

Les variables déclarées au niveau du fichier source et donc en dehors de tout bloc sont dites **globales**. Elles sont visibles depuis tout point du fichier source compris entre leur déclaration et la fin du fichier source, sauf si elles sont masquées par une variable locale ou un argument formel.

Les variables déclarées dans un bloc sont dites **locales** à ce bloc. Une variable locale est visible dans le bloc où elle a été déclarée ainsi que dans tout bloc inclus sauf si elle est masquée par une variable de même nom. Une variable locale masque les variables globales ou les variables locales de même nom déclarées dans un bloc englobant. D'où la règle que nous avons adoptée ci-dessus au §2.4, pour accéder à la valeur d'une variable dans un environnement et qui consiste à parcourir la pile depuis son sommet.

Les fonctions sont visibles depuis tout point du fichier source compris entre leur en-tête ou leur déclaration et la fin du fichier source.

Les arguments formels d'une fonction sont considérés comme des variables locales au bloc qui constitue le corps de la fonction. Elles sont donc visibles dans tout le corps de cette fonction sauf si elles sont masquées par des variables de même nom.

8.3 Commentaires

Il est possible de placer dans le texte source d'un programme des **commentaires**.

Un commentaire commence par `/*` et se termine par `*/`. Les commentaires peuvent s'étendre sur plusieurs lignes mais ne peuvent pas être imbriqués.

Par exemple, les textes suivants sont des commenatires :

```
/* Ceci est un commentaire. */
/*
 * Ceci est un commentaire
 * un peu plus long.
 */
```

La disposition du texte d'un programme est libre. Des blancs, des tabulations et des retours à la ligne peuvent être placés à tout endroit où cela ne coupe pas un identificateur.

Il est impératif, pour la lisibilité d'un programme de l'indenter et de le commenter.

Plusieurs styles d'indentations existent, dont celui adopté pour présenter les programmes de ce cours. Il faut en choisir un et s'y tenir tout au long d'un programme.

Les commentaires doivent être à la fois concis et précis et apporter des informations utiles. Inutile, par exemple de faire précéder une instruction d'itération par le commentaire :

```
/* Ceci est une boucle */
```

8.4 Mots réservés

Certains mots sont réservés. C'est le cas notamment des mots-clés qui apparaissent dans les instructions (if, `else`, …) ou des noms de type (`int`, `float`, …).

8.5 Exemple

Il s'agit d'écrire un programme qui calcule la somme des éléments d'un tableau de nombres flottants, qui a une dimension de 5 lignes par 2 colonnes.

Le programme est composé de la façon suivante :

- le tableau est représenté par la variable globale `tab`,
- la fonction `somme_ligne` calcule la somme des éléments d'une ligne dont le rang est donné,
- la fonction principale, `main`, calcule la somme des lignes.

Voici le programme :

```
/*
 * Somme des éléments d'un tableau d'entiers
 * tab : tableau
 * nbl : nombre de lignes
 * nbc : nombre de colonnes
 * somme_ligne(i) : somme des éléments de la ligne i
 */

#include <stdio.h>

float tab[2][5] = {{0.9, 1.8, 2.7, 3.6, 5.5},
                   {5.4, 6.3, 7.2, 8.1, 9.0}};
int nbl = 2, nbc = 5;

float somme_ligne(int i)
  {
  float s;
  int j;
  s = 0;
  for (j = 0; j <= nbc – 1; j = j + 1)
    s = s + tab[i][j];
  return s;
  }

void main(void)
  {
  float s ;
  int i ;
  s = 0 ;
  for (i = 0; i <= nbl – 1; i = i + 1)
    s = s + somme_ligne(i);
  printf("somme = %f\n", s);
  }
```

9

Pointeurs

9.1 Qu'est-ce que un pointeur ?

Un **pointeur** est l'adresse d'une donnée.

En C, les pointeurs sont des valeurs au même titre que les nombres ou les structures. Il existe donc des constantes et des variables de type pointeur, ou plus exactement de type « pointeur vers une donnée de type *T* », qu'en abrégé nous appellerons « pointeur vers un *T* ». Il existe aussi des opérateurs spécifiques pour manipuler les pointeurs.

Les pointeurs permettent l'accès aux éléments d'un tableau ainsi que le partage de valeurs.

9.2 Déclaration d'un pointeur

La déclaration d'un pointeur a la forme suivante :

T `*`*ident*

Par exemple :

```
int *p
char *p
struct personne
   {
   char[20] nom ;
   int age ;
   } *p
```

déclarent successivement :

- une variable `p` de type « pointeur vers une donnée de type `int` »,
- une variable `p` de type « pointeur vers une donnée de type `char` »,
- une variable `p` de type « pointeur vers une donnée de type `struct personne` ».

On peut déclarer simultanément des variables de type *T* et des variables de type pointeur vers une donnée de type *T*. Par exemple :

```
int n, *p;
```

déclare une variable `n` de type `int` et un pointeur `p` vers une donnée de type `int`.

9.3 Manipulation d'un pointeur

9.3.1 Adresse d'une donnée modifiable

L'adresse d'une donnée est obtenue par l'opérateur `&`.

Si *exp* est le nom d'une donnée modifiable (une valeur gauche), alors :

`&`*exp*

est une expression telle que :

- *type*(`&`*exp*) = pointeur vers un *type*(*exp*),
- *val*(`&`*exp*) = adresse de la donnée désignée par *exp*,

Attention ! `&exp` n'est pas une valeur gauche. Par exemple, si `x` est une variable de type `int`, on ne pourra pas écrire :

```
&x = 12
```

en pensant affecter la valeur `12` à `x`. Il suffit évidemment d'écrire :

```
x = 12
```

Par exemple, l'instruction suivante :

```
{
int *p, v;
p = &v;
}
```

affecte l'adresse de la variable `v` à la variable `p`. Remarquons que les types déclarés sont corrects : `v` est de type `int` et `p` est un pointeur vers un `int`.

9.3.2 Indirection

L'opérateur `*` permet d'extraire la valeur d'une donnée à partir d'un pointeur vers cette donnée, c'est à dire de son adresse. On dit que l'on réalise une **indirection**.

Si *exp* est une expression de type « pointeur vers une donnée de type *T* », alors :

`*`*exp*

est une expression telle que :

- *type*(`*`*exp*) = *T*,
- *val*(`*`*exp*) = valeur enregistrée dans la case désignée par *exp*.

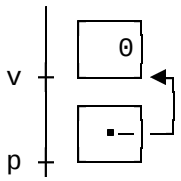
L'expression `*`*exp* est une valeur gauche : elle est le nom de la donnée modifiable dont *val*(*exp*) est l'adresse.

Considérons, par exemple, l'instruction :

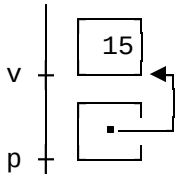
```
{
int *p, v;  (1)
v = 0;      (2)
p = &v;     (3)
*p = 15;    (4)
}
```

L'évolution de l'état du programme au cours de l'exécution de cette instruction est le suivant :

- après l'exécution de l'instruction (3) :



- après l'exécution de l'instruction (4) :



9.3.3 Accès au champ d'une structure

L'opérateur `->` permet d'extraire directement la valeur du champ d'une structure depuis un pointeur vers cette structure.

Si *exp* est une expression qui a pour valeur un pointeur vers une structure, et *ident* est le nom d'un champ de cette structure, alors l'expression :

exp`->`*ident*

est équivalente à :

`(*`*exp*`)->`*ident*

Par exemple, si `p` et `pp` sont des variables déclarées par :

```
struct personne
   {
   char nom[20] ;
   int age ;
   } p = {"Dupont", 36}, *pp;
```

et que l'on exécute l'instruction suivante :

```
pp = &p;
```

on aura :

val`(p.age) = `*val*`(pp->age) = 36`

9.4 Pointeurs et tableaux

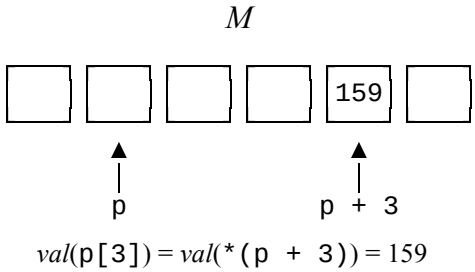
Par définition des pointeurs en C, si *M* est une zone de la mémoire contenant une suite de données de même type, si exp_1 une expression dont la valeur est un pointeur vers le i^e élément de *M* et si exp_2 une expression entière de valeur *j*, alors :

- $val(exp_1 + exp_2)$ = pointeur vers l'élément de rang $i + j$ dans *M*,
- $val(exp_1 - exp_2)$ = pointeur vers l'élément de rang $i - j$ dans *M*.

On a de plus :

exp_1[exp_2] ≡ *(exp_1 + exp_2)

Ces propriétés des pointeurs sont illustrées par la figure suivante :



val(p[3]) = *val*(*(p + 3)) = 159

Ces propriétés des pointeurs peuvent être appliquées aux tableaux, puisqu'en C (voir §5.3 ci-dessus), un tableau est une suite contiguë d'éléments de même type et a pour nom une constante dont la valeur est l'adresse du premier élément du tableau (élément de rang 0). On a donc les équivalences suivantes :

t ≡ &*t*[0]

t[*i*] ≡ *(*t* + *i*)

Par exemple, si `tab` est le nom d'un tableau déclaré par :

```
int tab[3] = {15, 8, 23};
```

on a :

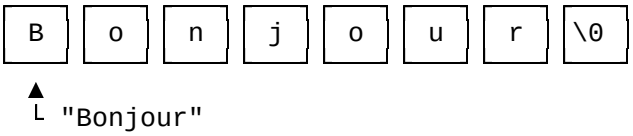
val(tab[1]) = *val*(*(tab + 1)) = 8.

9.5 Pointeurs et chaînes de caractères

En C, une chaîne de caractères est représentée par un tableau dont les éléments sont de type `char`. Une chaîne de caractères est donc manipulée comme un tableau.

Une constante littérale « chaîne de caractères » est un tableau qui a autant d'éléments que la chaîne de caractères plus un, le dernier, qui a la valeur 0 et qui marque la fin de la chaîne. La valeur d'une constante littérale est un pointeur vers le premier élément (rang 0) de ce tableau.

Par exemple :



Attention ! il y a une différence entre un nom de tableau et un pointeur vers son premier élément. Par exemple, si `p` est un pointeur vers un `char`, déclaré par :

```
char *p;
```

on peut écrire :

```
p = "Bonjour";
```

car `p` est bien une valeur gauche. On a enregistré le pointeur vers le premier élément du tableau `"Bonjour"` dans la case mémoire désignée par `p`. Par contre, si `t` est un tableau de caractères déclaré par :

```
char t[8]
```

on ne peut pas écrire :

```
t = "Bonjour";
```

car `t` étant une constante, n'est pas une valeur gauche.

9.6 Partage de valeur

Les pointeurs permettent le partage de valeurs comme nous allons le montrer sur l'exemple suivant.

On considère trois personnes p_1, p_2 et p_3 telles que p_1 est le père de p_2 et de p_3. On suppose que le nom de p_1 est « Jean », que celui de p_2 est « Yves » et que celui de p_3 est « Marie ».

En C, on pourra représenter ces trois personnes par trois variables `p1`, `p2` et `p3` ayant pour valeur une structure à deux champs : `nom` et `pere`, où `nom` est une chaîne de caractères (un pointeur vers un `char`) et `pere` est un pointeur vers la variable représentant le père. Pour la personne p_1 dont le père est inconnu, ce pointeur sera le pointeur nul désigné par `NULL`.

L'exécution de l'instruction suivante affecte son nom et son père à chaque personne puis imprime le nom du père de p_2 puis celui de p_3.

```
{
struct personne
   {
   char *nom;
   struct personne *pere;
   } p1, p2, p3
p1.nom = "Jean";
p1.pere = NULL;
p2.nom = "Yves";
p2.pere = &p1;
p3.nom = "Marie";
p3.pere = &p1;
printf("%s\n", p2->nom);
printf("%s\n", p3->nom);
}
```

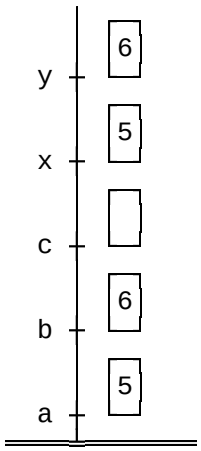
9.7 Passage d'arguments par adresse

Rappelons qu'en C les arguments d'une fonction sont transmis par valeur ce qui signifie que la valeur d'un argument est recopiée dans l'environnement dans lequel est évaluée la fonction.

Considérons, par exemple, le programme suivant :

```
int somme (int x, int y)
  {
  return (x + y);
  }
main()
  {
  int a, b, c;
  a = 5;
  b = 6;
  c = somme(a, b);
  }
```

L'état du programme immédiatement après l'appel de la fonction `somme` par la fonction `main` est le suivant :



On voit bien que les valeurs de `a` et de `b` ont été recopiées au sommet de la pile dans les cases mémoire liées à `x` et `y`.

Le passage d'un argument par valeur souffre de deux défauts :

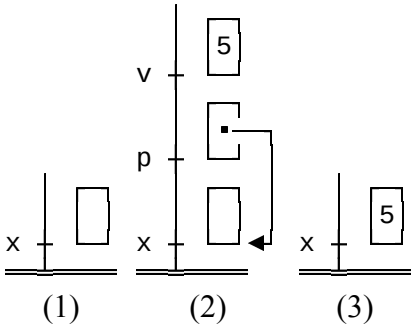
- la fonction ne peut pas modifier la valeur de cet argument puisqu'elle n'a accès qu'à sa copie,
- si l'argument est une structure ou un tableau volumineux, sa copie peut être coûteuse.

En ce cas la solution est de passer un pointeur vers l'argument (c.-à-d. son adresse) au lieu de l'argument lui-même. On dit que l'argument est passé **par adresse**.

On peut illustrer le mécanisme du passage d'argument par adresse en visualisant l'exécution du petit programme suivant qui appelle une fonction `init` pour affecter une valeur à une variable `x`.

```
int init(int *p, int v)
  {
  *p = v;
  }
main()
  {
  int x;
  init(&x, 5);
  }
```

La figure suivante montre l'état de ce programme immédiatement avant l'appel de la fonction `init` (1), immédiatement après (2) et enfin après le retour au programme principal (3).



Voyons un autre exemple. Considérons une horloge qui indique les heures, les minutes et les secondes. Cette horloge peut être représentée par la structure suivante :

```
struct horloge
   {
   int heures;
   int minutes;
   int secondes;
   }
```

Pour faire avancer une telle horloge, on peut définir une fonction `avancer_horloge`, dont l'unique argument est un pointeur vers l'horloge à faire avancer :

```
void avancer_horloge(struct horloge *h)
   {
   if (h->secondes < 59)
      h->secondes = h->secondes + 1;
   else
      {
      h->secondes = 0;
      if (h->minutes < 59)
         h->minutes = h->minutes + 1;
      else
         {
         h->minutes = 0;
         h->heures = h->heures + 1;
         }
      }
   }
```

Supposons maintenant qu'une variable globale `mon_horloge` soit déclarée par :

```
struct horloge mon_horloge
```

Pour faire avancer cette horloge il suffira d'appeler la fonction `avancer_horloge` par l'instruction suivante :

```
avancer_horloge(&mon_horloge)
```

On remarquera que lorsqu'un argument effectif est un nom de tableau, ce n'est pas le tableau qui est passé en argument mais un pointeur vers son premier élément, par définition d'un nom de tableau (voir §5.3.1 ci-dessus). Les tableaux sont donc passés par adresse en C.

Dans la déclaration d'un argument formel de type tableau, la valeur de la première dimension pourra être omise mais pas les suivantes.

Le programme suivant, par exemple, calcule la somme des éléments d'un tableau.

```
#include <stdio.h>
int mon_tableau[12] = {1, 2, 3, 4, 5, 6, 6, 5, 4, 3, 2, 1};
int somme(int tab[], int n)
   {
   int i, s;
   s = 0;
   for (i = 0; i <= n - 1; i = i + 1)
      {
      s = s + tab[i];
      }
   return s;
   }
main()
   {
   printf("Somme = %d\n", somme(mon_tableau, 12));
   getchar();
   }
```

L'exécution de ce programme produit l'affichage suivant :

```
somme = 42
```

10

Chaînes de caractères

10.1 Représentation d'une chaîne de caractères

Une chaîne de caractères est une suite de plusieurs caractères.

On note :

`""` une chaîne de caractères vide,
"$c_0c_1\ldots c_{n-1}$" une chaîne de n caractères c_0, c_1, …, c_{n-1}.

Rappelons (voir ci-dessus 2.2.3) que les caractères spéciaux tels que la tabulation ou le retour à la ligne ont une écriture spécifique caractérisée par le préfixe `\` (anti-slash). Par exemple, `\t` pour la tabulation, `\n` pour le retour à la ligne et `\"` pour le guillemet.

La longueur d'une chaîne de caractères est son nombre de caractères. Une chaîne de caractères de longueur nulle est appelée chaîne de caractères de caractères vide.

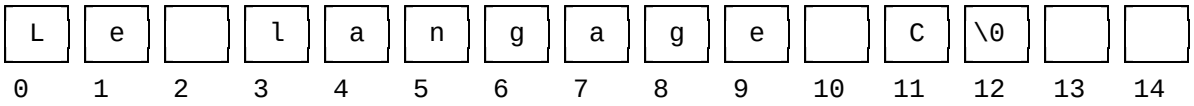
Attention ! à la différence entre un caractère et une chaîne de caractères :

`'A'` est le caractère *A*,
`"A"` est la chaîne composée du seul caractère *A*.

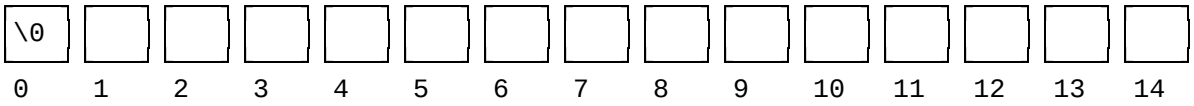
En C une chaîne de caractères est contenue dans un tableau de caractères. Les caractères sont rangés consécutivement à partir de la case 0. Le caractère `\0` (de code 0) est placé immédiatement après le dernier caractère de la chaîne pour en marquer la fin.

Un tableau de dimension n permet donc de représenter des chaînes de caractères de longueur maximum n - 1. Par exemple :

- La représentation de la chaîne de caractères `"Le langage C"` dans un tableau de longueur 15 est la suivante :



- La représentation d'une chaîne de caractères vide dans le même tableau est la suivante :



10.2 Déclaration et initialisation

Si *t* est un identificateur, *n* est un entier positif et *k* est un entier tel que $0 \leq k \leq n - 2$, la déclaration :

```
char t[n] = {c0, c2, …, ck, \0}
```

définit une variable *t* de type tableau de caractères de longueur *n*, pouvant contenir une chaîne de caractères de longueur maximale $n - 1$, initialisée à "$c_0c_2\ldots c_k$".

Pour faciliter la lecture de l'initialisation, l'écriture abrégée suivante est autorisée :

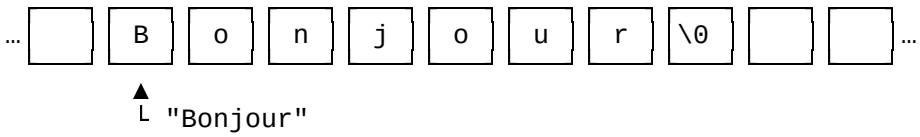
```
char t[n] = "c1c2…ck"
```

Par exemple :

```
char titre[15] = "Le langage C"
```

10.3 Constante littérale chaîne de caractères

Une constante littérale chaîne de caractères a pour valeur un tableau stocké dans la zone des constantes, c'est à dire l'adresse du premier caractère de ce tableau. Par exemple :



Si `t` est un tableau de caractères, l'affectation :

```
t = "Bonjour"
```

est interdite car on ne peut pas affecter un tableau à un tableau.

10.4 Affichage d'une chaîne de caractères

Si *cc* est une chaîne de caractères, l'instruction :

```
printf("%s", cc);
```

l'affiche.

10.5 Manipulation d'une chaîne de caractères

Pour illustrer la manipulation des chaînes de caractères en C, nous allons programmer les quatre opérations suivantes :

- calcul de la longueur d'une chaîne de caractères,
- test de égalité de deux chaînes de caractères,

- concaténation de deux chaînes de caractères,
- extraction d'une sous-chaîne de caractères.

Attention ! C ne contrôle pas que la marque de fin de chaîne ait été placée, ni que la longueur maximum du tableau dans lequel est rangée une chaîne de caractères est dépassée. Il faut donc, si on l'estime nécessaire, prévoir ces contrôles dans les programmes manipulant des chaînes de caractères.

10.5.1 Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères contenue dans un tableau *s* est calculée par la fonction `longueur` dont la définition est la suivante :

```
int longueur(char s[])
  {
  /*
   * La longueur d'une chaîne est égale à la position
   * de la marque de fin.
   * En effet si la longueur d'une chaîne est l elle se terminera
   * au rang l - 1 puisque les caractères sont numérotés à partir de 0.
   * La marque de fin se trouve donc bien à la position l.
   */
  int i;
  i = 0;
  while (s[i] != 0)
  i = i + 1;
  return i;
  }
```

10.5.2 Egalité de deux chaînes de caractères

L'égalité de deux chaînes de caractères s_1 et s_2 est calculée par la fonction `egal` dont la définition est la suivante :

```
unsigned char egal(char s1[], char s2[])
  {
  /*
   * Deux chaînes sont égales
   * si elles ont la même longueur
   * et si leurs caractères de même rang sont égaux.
   * On cherche à partir de 0 le premier rang i pour lequel :
   * - soit s1[i] != s2[i] ce qui se produit :
   *   1) si on a atteint la fin de s1 et pas celle de s2,
   *   2) si on a atteint la fin de s2 et pas celle de s1,
   *   3) si le i-ème caractère de s1 n'est pas égal
   *      au i-ème caractère de s2
   *   Dans ces trois cas les chaînes s1 et s2 ne sont pas égales.
   * - soit s1[i] == '\0', en ce cas les chaînes s1 et s2
   *   sont égales si s2[i] == '\0' (chaînes de même longueur).
   */
   int i;
   i = 0;
   while ((s1[i] == s2[i]) && (s1[i] != '\0'))
    i = i + 1;
   return (s1[i] == s2[i]);
   }
```

10.5.3 Concaténation de deux chaînes de caractères

La concaténation de deux chaînes de caractères `s1` et `s2` est réalisée par la fonction `concatener` dont la définition est la suivante.

```
void concatener(char s1[], char s2[])
  {
  /*
   * On lit la chaîne s1 jusqu'à la marque de fin (rang i)
   * puis on recopie la chaîne s2, marque de fin comprise,
   * dans s1 à partir du rang i.
   */
  int i, j;
  i = 0;
  while (s1[i] != 0)
    i = i + 1;
  j = 0;
  do
    {
    s1[i] = s2[j];
    i = i + 1;
    j = j + 1;
    }
  while (s2[j] != 0);
    s1[i] = 0;
  return;
  }
```

L'argument `s1` qui est modifié par cette opération ne peut pas être une constante littérale chaîne de caractères.

10.5.4 Extraction d'une sous-chaîne de caractères

L'extraction dans une chaîne de caractères `s1` de la sous-chaîne de caractères `s2` débutant au rang `i` et de longueur `n` est réalisée par la fonction `extraire`. Cette fonction retourne un entier égal à `0` si l'opération a été réalisée et 1 si elle n'est pas réalisable, c'est-à-dire si `i` ou `n` sont négatifs ou bien si `i` + `n` –1 est supérieur ou égal à la longueur de `s1`.

```
int extraire(char s[], int i, int n, char ss[])
  {
  int k;
  if ((i < 0) || (n < 0))
    /*
     * Paramètres négatifs :
     * on retourne 1 (extraction non réalisable).
     */
    return 1;
  /*
   * On avance jusqu'au rang i pour tester si la chaîne
   * ne se termine pas avant.
   */
  k = 0;
  while ((s[k] != 0) && (k < i))
    {
    k = k + 1;
    }
  if (k < i)
    /*
     * s se termine avant le rang i :
     * on retourne 1.
     */
    return 1;
  /*
   * On écrit dans ss les caractères de rang
   * i, i + 1, ..., i + n - 1 de s.
   */
  k = 0;
  while ((s[i + k] != 0) && (k <= n - 1))
    {
    ss[k] = s[i + k];
    k = k + 1;
    }
  if (k == n)
    /*
     * Extraction réussie :
     * on écrit la marque de fin de chaîne dans ss.
     */
     {
    ss[k] = '\0';
    return 0;
    }
  else
    /*
     * Il n'y a pas n caractères à extraire :
     * on retourne 1.
     */
    return 1;
  }
```

L'argument `s2` qui est modifié par cette opération ne peut pas être une constante littérale chaîne de caractères.

10.5.5 Exemple

Le programme C suivant illustre l'utilisation de ces quatre fonctions.

```
#include <stdio.h>
int longueur(char s[])
…
unsigned char egal(char s1[], char s2[])
…
void concatener(char s1[], char s2[])
…
void extraire(char s[], int i, int n, char ss[])
…
main()
   {
   char ma_chaine[50] = "";
   /*
    * Longueur
    */
   printf("%d\n", longueur(""));
   printf("%d\n", longueur("Bonjour"));

   /* Egalite
    */
   printf("%d\n", egal("jour", "journal"));
   printf("%d\n", egal("jour", "nuit"));
   printf("%d\n", egal("jour", "jour"));

   /*
    * Concatenation
    */
   concatener(ma_chaine, "Le langage C et ");
   concatener(ma_chaine, "le langage C++");
   printf("%s\n", ma_chaine);

   /*
    * Extraction.
    */
   err = extraire("Le langage C", 14, 7, ma_chaine);
   if (err == 0)
     printf("%s\n", ma_chaine);
   else
     printf("Extraction impossible !\n");
   err = extraire("Le langage C", 3, 7, ma_chaine);
   if (err == 0)
     printf("%s\n", ma_chaine);
   else
     printf("Extraction impossible !\n");
   err = extraire("Le langage C", 3, 12, ma_chaine);
   if (err == 0)
     printf("%s\n", ma_chaine);
   else
     printf("Extraction impossible !\n");
   }
```

Ce programme réalise successivement :

- le calcul de la longueur de la chaîne de caractères `""` ;
- le calcul de la longueur de la chaîne de caractères `"Bonjour"` ;
- le test d'égalité des chaînes de caractères `"jour"` et `"journal"`, `"jour"` et `"nuit"`, `"jour"` et `"jour"` (seules les deux dernières sont égales) ;
- la concaténation des chaînes de caractères `""` , `"Le langage C et "` et `"le langage C++"` ;

- l'extraction dans la chaîne de caractères `"Le langage C"` de la sous-chaîne de caractères commençant au rang 14 et de longueur 3 (qui est impossible), de la sous-chaîne de caractères commençant au rang 3 et de longueur 7 et de la sous-chaîne de caractères commençant au rang 3 et de longueur 12 (elle aussi impossible).

L'exécution de ce programme fournit l'affichage suivant :

```
0
4
0
0
1
Le langage C et le langage C++
Extraction impossible !
langage
Extraction impossible !
```

11

Manipulation de séquences

11.1 Objectif

L'objectif de ce chapitre est l'étude de la manipulation en C de **séquences**. Par séquence, on entend une suite ordonnée d'éléments. Par exemple : la séquence des noms des jours de la semaine du lundi au dimanche, la séquence des températures moyennes journalières du 1er au 31 janvier 2001 à Marseille, la séquence des étudiants de la licence d'informatique classée par ordre alphabétique de nom.

Cette étude se déroulera selon la démarche suivante :

1. Etude abstraite d'une séquence, c.-à-d. de ses propriétés et de ses opérateurs indépendamment d'une représentation en termes d'un langage de programmation spécifique
2. Choix d'une représentation d'une séquence en C,
3. Programmation en C d'un jeu d'opérateurs de base.

Cette démarche peut être comparée à celle du traitement des nombres sur un ordinateur. On dispose des propriétés mathématiques des nombres et de leurs opérateurs et il s'agit de choisir une représentation de ces nombres et une implantation de ces opérateurs qui soit adaptée à un ordinateur spécifique.

L'étude abstraite (étape 1) traitera des séquences indépendamment du type de leurs éléments (nombres, chaînes de caractères, valeurs composées, etc.). Par contre, pour des raisons de simplicité, nous nous limiterons (étapes 2 et 3) à la représentation et à la programmation en C de séquences de nombres entiers.

11.2 Etude abstraite des séquences

11.2.1 Qu'est-ce qu'une séquence ?

Une séquence est une suite ordonnée d'éléments de même type.

Nous appellerons **séquence vide**, une séquence de zéro éléments et **longueur d'une séquence** son nombre d'éléments.

On notera :

[]	une séquence vide,
[e]	une séquence de 1 élément e,
[e_0, …, e_n]	une séquence de n + 1 éléments e_0, …, e_n.

Par exemple :

[*lundi*, *mardi*, *mercredi*, *jeudi*, *vendredi*, *samedi*, *dimanche*]

[]

11.2.2 Manipulation d'une séquence

Une séquence peut être créée, par exemple comme une séquence vide, puis évoluer par insertion, remplacement ou suppression d'éléments.

Par exemple :

[]	création d'une séquence vide
[6, 4]	insertion de l'élément 6 au rang 0, puis de l'élément 4 au rang 1
[6, 8, 13, 4]	insertion de l'élément 8 au rang 1, puis de l'élément 13 au rang 2
[6, 8, 13, 1]	remplacement de l'élément de rang 3 par l'élément 1
[6, 8, 13]	suppression de l'élément de rang 3

Dans une séquence, on peut extraire des éléments dont le rang est donné. Par exemple, l'élément de rang 2 dans la séquence [6, 8, 13, 14] est 13.

Une séquence peut être manipulée par éléments ou par sous-séquences. Voyons la différence sur l'opération d'insertion. Une insertion par éléments peut être réalisée à l'aide de l'opération :

insérer(*s*, *i*, *e*)

qui insère dans la séquence *s*, à partir du rang *i*, l'élément *e*. Une insertion par sous-séquences peut-être réalisée à l'aide de l'opération :

insérer(*s*, *i*, *s'*)

qui insère dans la séquence *s*, à partir de du rang *i*, la sous-séquence *s'*.

Considérons par exemple la transformation *T* suivante :

T : [*printemps*, *hiver*] → [*printemps*, *été*, *automne*, *hiver*]

Dans une manipulation par éléments, la transformation *T* pourra être réalisée par l'enchaînement des deux opérations suivantes :

insérer([*printemps*, *hiver*], 1, *été*)
insérer([*printemps*, *hiver*], 2, *automne*)

et dans une manipulation par sous-séquences, elle pourra être réalisée par l'unique opération suivante :

insérer([*printemps*, *hiver*], 1, [*été*, *automne*]).

Dans la suite de ce chapitre nous manipulerons les séquences par éléments à l'aide du jeu d'opérations de base suivant :

vider(*s*)	met à zéro la longueur de la séquence *s*
longueur(*s*)	retourne la longueur de la séquence *s*
élément(*s*, *i*)	retourne l'élément de rang *i* de la séquence *s* contrainte : $0 \leq i < longueur(s)$
insérer(*s*, *i*, *e*)	insère dans la séquence *s*, au rang *i*, l'élément *e* contrainte : $0 \leq i \leq longueur(s)$
remplacer(*s*, *i*, *e*)	remplace dans la séquence *s*, l'élément de rang *i*, par l'élément *e* contrainte : $0 \leq i < longueur(s)$
supprimer(*s*, *i*)	supprime dans la séquence *s*, l'élément de rang *i* contrainte : $0 \leq i < longueur(s)$

11.3 Représentation d'une séquence d'entiers en C

Le principal problème posé par la représentation des séquences est la variabilité de leurs longueurs. L'allocation dynamique de mémoire permet de résoudre ce problème mais elle ne sera pas étudiée dans ce cours. La seule structure dont nous disposions est donc celle de tableau.

Une séquence d'éléments de type *T* peut être représentée en C par un tableau d'éléments de type *T* et de dimension égale ou supérieure à la longueur de cette séquence. Nous appellerons ce tableau : le **support de la séquence**. Il faut de plus indiquer la longueur de la séquence ou marquer sa fin par un élément, un élément de type *T* qui ne soit pas un élément de la séquence. Nous retiendrons la première solution et représenterons une séquence d'entiers (*T* = `int`) par la structure à trois champs suivante :

```
struct sequence
  {
  int support[LONMAX], lon;
  }
```

où le tableau `support` est le support de la séquence, la constante `LONMAX` est la taille de ce support et donc la longueur maximum d'une séquence et l'entier `lon` est la longueur de la séquence.

La constante `LONMAX` sera définie par une macro-instruction placée avant la déclaration de la structure `sequence`.

```
#define LONMAX v
```

Une macro-instruction est traitée par le pré-processeur. Dans notre cas, le pré-processeur remplacera chaque instance de `LONMAX` par la valeur *v* indiquée dans le texte du programme, avant de compiler ce programme.

11.4 Programmation du jeu d'opérations de base

Chaque opération de base est réalisée par une fonction de même nom. La séquence sur laquelle porte l'opération est transmise par adresse c.-à-d. comme un pointeur vers la structure qui représente cette séquence.

Afin de signaler au programme appelant qu'une opération n'a pu être réalisée, on supposera l'existence d'une variable globale `erreur` de type `int` qui sera mise à 0, si l'opération a pu

être réalisée et à *i* (*i* > 0) si l'opération n'a pu être réalisée pour une cause identifiée par le numéro *i*.

11.4.1 Vidage d'une séquence

Pour vider une séquence il faut mettre à zéro sa longueur. La définition de la fonction `vider` est donc la suivante :

```
/*
 * Vidage d'une séquence s
 */
void vider(struct sequence *s)
   {
   s->lon = 0;
   return;
   }
```

11.4.2 Longueur d'une séquence

La définition de la fonction `longueur` est la suivante :

```
/*
 * Longueur d'une séquence s
 */
int longueur(struct sequence *s)
   {
   return s->lon;
   }
```

11.4.3 Extraction d'un élément

Pour extraire l'élément de rang *i* d'une séquence *s*, il faut :

1. vérifier que *i* est bien un rang d'un élément de la séquence, c.-à-d. que $0 \leq i \leq$ *longueur*(*s*) (`erreur` = 1 sinon) ;
2. retourner la valeur du i^e élément du support de *s*.

La définition de la fonction `element` est donc la suivante :

```
/*
 * Extraction dans une séquence s de l'élément de rang i
 */
int element(struct sequence *s, int i)
  {
  /*
   * On teste la validité de i
   */
  if ((i < 0) || (i >= s->lon))
     {
     erreur = 1;
     return 0;
     }
  /*
   * On retourne l'élément de rang i
   */
  erreur = 0;
  return s->support[i];
  }
```

11.4.4 Insertion d'un élément

Pour insérer un élément *x* au rang *i* d'une séquence *s*, il faut :

1. vérifier que la longueur de *s* est inférieure à la taille de son support (`erreur` = 2 sinon) et que *i* est bien un rang d'un élément de la séquence, c.-à-d. que $0 \leq i \leq longueur(s)$ (`erreur` = 1 sinon) ;
2. déplacer d'une case vers la droite tous les éléments de rang supérieur à *i* en commençant par le dernier élément de *s* afin de ne pas écraser à chaque fois l'élément suivant ;
3. enregistrer *x* dans la i^e case du support de s ;
4. incrémenter de 1 la longueur de *s*.

La définition de la fonction `insertion` est la suivante :

```
/*
 * Insertion d'un élément x au rang i d'une séquence s
 */
void inserer(struct sequence *s, int i, int x)
  {
  int j;
  /*
   * On teste si la séquence a sa longueur maximum.
   */
  if (s->lon == LONMAX)
    {
    erreur = 2;
    return;
    }
  /*
   * On teste la validité de i.
   */
  if (i < 0 || i > s->lon)
    {
    erreur = 1;
    return;
    }
  /*
   * Si l'insertion ne se fait pas en fin de séquence
   * on décale les éléments de i à ls d'un rang vers la droite
   * en partant de la fin de la séquence.
   */
  if (i <= s->lon)
    for (j = s->lon - 1; j >= i; j--)
      s->support[j + 1] = s->support[j];
  /*
   * On insère l'élément x au rang i.
   */
  s->support[i] = x;
  /*
   * On incrémente la longueur.
   */
  s->lon = s->lon + 1;
  erreur = 0;
  return;
  }
```

11.4.5 Remplacement d'un élément

Pour remplacer dans une séquence *s*, l'élément de rang *i*, par un élément de rang *x*, il faut :

1. vérifier que *i* est bien un rang d'un élément de la séquence, c.-à-d. que $0 \leq i \leq$ *longueur*(*s*) (`erreur` = 1 sinon) ;
2. enregistrer *x* dans la i^e case du support de *s*.

La définition de la fonction `remplacer` est donc la suivante :

```
/*
 * Remplacement dans une séquence s, de l'élément de rang i, par x
 */
int remplacer(struct sequence *s, int i, int x)
  {
  /*
   * On teste la validité de i
   */
  if ((i < 0) || (i >= s->lon))
     {
     erreur = 1;
     return;
     }
  /*
   * On remplace l'élément de rang i par x.
   */
  s->support[i] = x;
  erreur = 0;
  return;
  }
```

11.4.6 Suppression d'un élément

Pour supprimer l'élément de rang *i* dans une séquence *s*, il faut :

1. vérifier que *i* est bien un rang d'un élément de la séquence, c.-à-d. que $0 \leq i \leq$ *longueur*(*s*) (`erreur` = 1 sinon) ;
2. déplacer d'une case vers la gauche tous les éléments de rang supérieur à *i* en commençant par le dernier élément de *s* afin de ne pas écraser à chaque fois l'élément suivant.
3. décrémenter de 1 la longueur de *s*.

La définition de la fonction `supprimer` est donc la suivante :

```
/*
 * Suppression dans une séquence s, de l'élément de rang i
 */
void supprimer(struct sequence *s, int i)
  {
  int j;
  /*
   * On teste la validité de i
   */
  if ((i < 0) || (i >= s->lon))
    {
    erreur = 1;
    return;
    }
  /*
   * Si la suppression n'est pas celle du dernier élément de s on
   * décale les éléments de rang > i d'un rang vers la gauche.
   */
  if (i < s->lon)
    for (j = i; j <= s->lon; j = j + 1)
      s->support[j] = s->support[j + 1];
  /*
   * On décrémente de 1 la longueur de s.
   */
  s->lon = s->lon - 1;
  erreur = 0;
  return;
  }
```

11.4.7 Exemple

Considérons le programme C suivant :

```
#include <stdio.h>
#include <stdlib.h>

#define LONMAX 10

struct sequence
  {
  int support[LONMAX], lon;
  };
int erreur;
void vider(struct *sequence s)
  …
int longueur(struct *sequence s)
  …
int element(struct *sequence s, int i)
  …
void inserer(struct *sequence s, int i, int x)
  …
void remplacer(struct *sequence s, int i, int x)
  …
void supprimer(struct *sequence s, int i)
  …
void afficher(struct *sequence *s)
  {
  int lon, i;
  lon = longueur(s);
  printf("[");
  if (lon > 0)
     {
     printf("%d", element(s, 0));
     for (i = 1; i < lon; i++)
       printf(", %d", element(s, i));
     }
  printf("]\n");
  }
main()
  {
  struct sequence *ma_sequence;

  vider(ma_sequence);
  afficher(ma_sequence);
  inserer(ma_sequence, 0, 5);
  afficher(ma_sequence);
  inserer(ma_sequence, 1, 13);
  afficher(ma_sequence);
  inserer(ma_sequence, 1, 7);
  afficher(ma_sequence);
  remplacer(ma_sequence, 1, 11);
  afficher(ma_sequence);
  supprimer(ma_sequence, 2);
  afficher(ma_sequence);
  }
```

La fonction `afficher` a pour effet de bord d'afficher la séquence à laquelle elle s'applique sous la forme :

[e_1, …, e_n]

La seule petite difficulté dans la définition de cette fonction est de faire en sorte que tout élément soit suivi d'un point-virgule sauf s'il est le dernier élément de la séquence.

La fonction `main` crée une séquence vide, y insère l'élément 5 (au rang 0), puis à sa droite l'élément 13 (au rang 1), puis entre eux l'élément 7 (au rang 1) ; elle remplace ensuite l'élément 7 par l'élément 11 (au rang 1) ; enfin elle supprime l'élément 13 (au rang 2). Elle affiche de plus la séquence obtenue après chaque opération. Supposant que les arguments sont valides, l'indicateur d'erreur n'est pas testé.

On notera que dans le corps de ces deux fonctions une séquence est uniquement manipulée au travers des opérateurs de base sans avoir à connaître sa représentation. C'est le principe de la programmation par types abstraits.

L'exécution de ce programme fournit l'affichage suivant :

```
[]
[5]
[5, 13]
[5, 7, 13]
[5, 11, 13]
[5, 11]
```

12

Récursivité

12.1 Notion de récursivité

Une fonction récursive est une fonction qui s'appelle elle-même. Considérons, par exemple, la fonction factorielle (!). Elle est définie par :

$0 ! = 1$
$n ! = 1 \times 2 \times \ldots \times (n - 1) \times n$ si $n > 0$

Si l'on remarque que le produit $1 \times 2 \times \ldots \times (n - 1)$ est égal à $(n - 1) !$, on peut définir la factorielle de la façon suivante :

$n ! = 1$, si $n = 0$
$n ! = (n - 1) ! \times n$, si $n > 0$

Une telle définition est dite récursive car la fonction factorielle intervient dans sa propre définition.

Remarquons que le calcul de la factorielle selon cette définition se termine car n décroit à chaque application de la factorielle et que pour $n = 0$, on connaît le résultat. On a par exemple :

$3 ! = (2 !) \times 3 = ((1 !) \times 2) \times 3 = (((0 !) \times 1) \times 2) \times 3 = ((1 \times 1) \times 2) \times 3 = 6$

Une définition récursive est formée d'une **relation de récurrence** et d'un **cas de base**. Par exemple, dans la définition récursive de la factorielle :

- la relation de récurrence est « $n ! = (n - 1) ! \times n$, si $n > 0$ » ;
- le cas de base est « $0 ! = 1$ ».

Comme tous les langages de programmation modernes, le langage C permet de définir des fonctions récursives. Voici la définition de la factorielle en C :

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

La valeur retournée par la fonction `fac` a été déclarée de type `unsigned long` car elle peut être très grande.

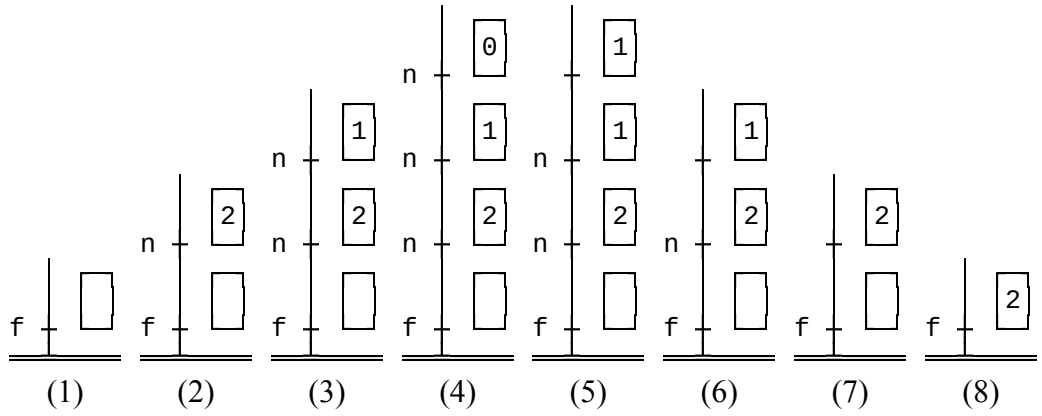
12.2 Evaluation d'une fonction récursive

L'appel d'une fonction récursive déclenche une succession d'appels de cette fonction jusqu'à atteindre le cas de base à partir duquel il est possible de terminer l'évaluation de ces appels dans l'ordre inverse de leur déclenchement. Par exemple, l'appel `fac(2)` déclenche l'appel `fac(1)` qui déclenche l'appel `fac(0)` qui peut être évalué puisque le cas de base est atteint. Il est alors possible de terminer l'évaluation de `fac(1)` puis celle de `fac(2)`. C'est la structure en pile de l'environnement qui permet de conserver la mémoire des appels successifs de la fonction.

Considérons, par exemple, le programme suivant qui affecte à la variable `f` la valeur de la factorielle de 2.

```
unsigned long fac(int n)
  {
  if (n == 0)
    return 1;
  else
    return fac(n - 1) * n;
  }
main()
  {
  unsigned long f;
  f = fac(2)
  }
```

L'évolution de l'environnement au cours de l'exécution de ce programme est le suivant :



1. La fonction `main` a été appelée ; la variable `f` est empilée au sommet de l'environnement.
2. La fonction `fac` a été appelée avec l'argument effectif `2` ; la variable `n` est empilée au sommet de l'environnement avec la valeur `2`.
3. La fonction `fac` a été appelée avec l'argument effectif `1` ; la variable `n` est empilée au sommet de l'environnement avec la valeur `1`.
4. La fonction `fac` a été appelée avec l'argument effectif `0` ; la variable `n` est empilée au sommet de l'environnement avec la valeur `0`.
5. L'exécution du corps de la fonction `fac` retourne la valeur `1` qui est empilée au sommet de l'environnement puis dépilée après avoir été substituée à l'expression `fac(0)`.
6. L'exécution du corps de la fonction `fac` retourne la valeur `1` qui est empilée au sommet de l'environnement puis dépilée après avoir été substituée à l'expression `fac(1)`.

7. L'exécution du corps de la fonction `fac` retourne la valeur `2` qui est empilée au sommet de l'environnement puis dépilée après avoir été substituée à l'expression `fac(2)`.
8. La valeur `2` est affectée à `f`.

12.3 Récursivité et déclarativité

La fonction `fac` aurait pu être définie itérativement de la façon suivante :

```
unsigned long fac(int n)
   {
   unsigned long f;
   int i;
   f = 1;
   for (i = 2; i <= n; i = i + 1)
      f = f * i;
   return f;
   }
```

Dans cette définition on a indiqué comment calculer la factorielle, alors que dans la définition récursive on avait simplement donné la formule permettant de la calculer. Dans le premier cas on parle de **programmation impérative** et dans le second de **programmation déclarative**.

En idéalisant quelque peu, on peut dire que la programmation déclarative consiste à donner la formule vérifiée par le résultat et à laisser l'ordinateur choisir la façon de le calculer alors que la programmation impérative consiste à donner les instructions permettant de calculer ce résultat. La programmation déclarative produit en général des programmes plus lisibles que la programmation impérative.

12.4 Définition d'une fonction récursive

Nous proposons la méthodologie suivante pour définir une fonction récursive en C :

1. Etablir la relation de récurrence.
2. Etablir le cas de base.
3. Vérifier que l'application de la relation de récurrence convergera vers le cas de base.
4. Ecrire la définition de la fonction en C en utilisant une instruction conditionnelle de la forme suivante :

```
if cas de base
   le traiter
else
   appliquer la relation de récurrence
```

Considérons, par exemple, la fonction *somme* qui appliquée à deux entiers i et j ($i \leq j$) calcule la somme des entiers compris entre i et j.

1. **Relation de récurrence**. La somme des entiers compris entre i et j ($i < j$) est égale à i plus la somme des entiers compris entre $i + 1$ et j :

   $somme(i, j) = i + somme(i + 1, j)$ si $i < j$
2. **Cas de base**. La somme des entiers compris entre i et i est égale à i :

$somme(i, i) = i$

3. **Convergence**. Le calcul se termine puisqu'à chaque appel de la fonction *somme* on ajoute 1 à i qui deviendra égal à j au $(j - i + 1)^e$ appel, ce qui arrêtera la récursion puisque le cas de base est atteint, pourvu, évidemment, que la condition $i \leq j$ ait été respectée.

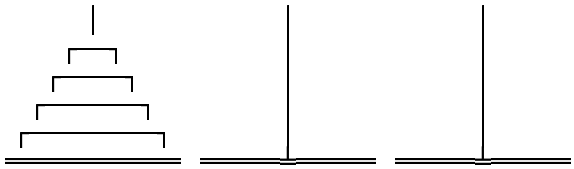
4. **Programme C**.

```
int somme(int i, int j)
  {
  if (i == j)
    return i;
  else
    return i + somme(i + 1, j);
  }
```

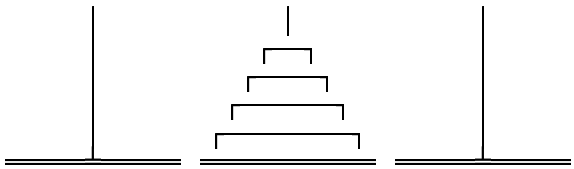
(On suppose que les arguments vérifient la contrainte `i <= j`.)

12.5 Les tours d'Hanoï

Pour terminer, voici l'un des exemples les plus classiques d'application de la récursivité : le jeu des tours d'Hanoï. On dispose de 3 tiges verticales *A*, *B* et *C*, sur lesquelles peuvent être enfilés des disques percés en leur milieu. Une pile de disques est enfilée sur la tige *A* par tailles décroissantes et aucun disque n'est enfilé sur les tiges *B* et *C*. Le jeu consiste à déplacer la pile de disques de la tige *A* à la tige *B* par une suite de mouvements dont chacun consiste à déplacer un disque du sommet d'une pile au sommet d'une autre sans jamais empiler un disque sur un disque de plus petite taille.



(avant)



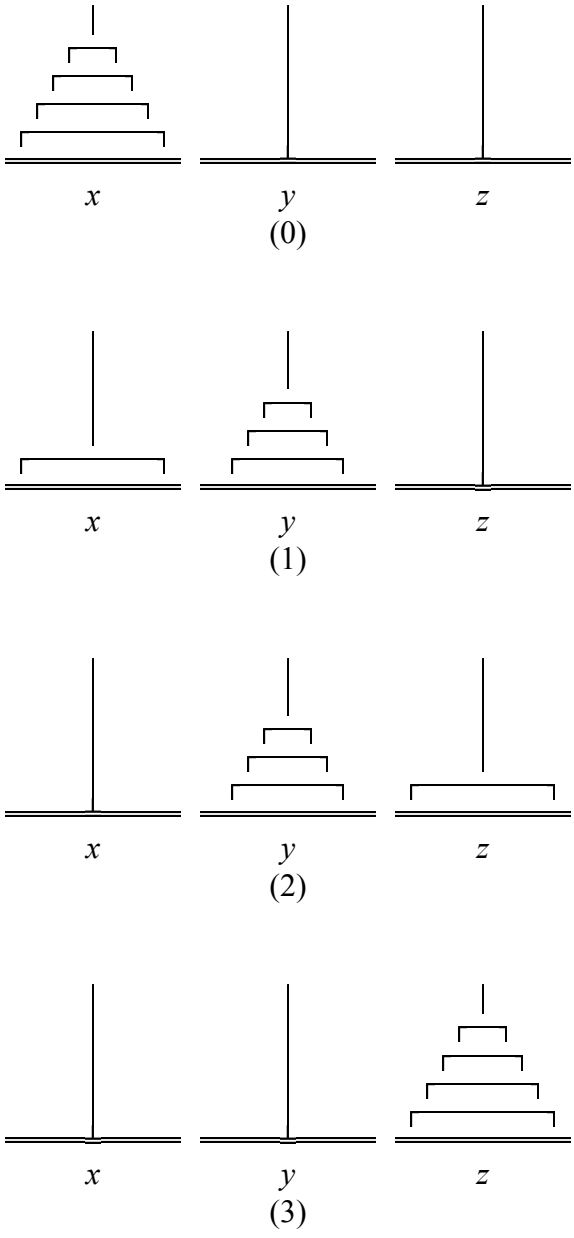
(après)

La solution n'est à priori pas évidente pas mais elle est étonnamment simple si on l'exprime de façon récursive :

1. Pour déplacer une pile de n disques d'une tige x à une tige z par l'intermédiaire d'une tige y, il suffit de déplacer la pile des $n - 1$ disques supérieurs de la tige x à la tige y en utilisant la tige z comme intermédiaire, puis de déplacer le disque supérieur de la tige x au sommet de la tige z et enfin de déplacer les $n - 1$ disques supérieurs de la tige y à la tige z en utilisant la tige x comme intermédiaire (**règle de récurrence**).

2. Pour déplacer une pile vide aucun mouvement n'est nécessaire (**cas de base**).

Cette stratégie est illustrée par la figure suivante :



Ecrivons maintenant le programme C permettant de jouer. Il se réduit à la définition et à l'appel d'une fonction unique :

hanoi(*x*, *y*, *z*, *n*)

qui déplace une pile de *n* disques de la tige *x* vers la tige *z* en utilisant la tige intermédiaire *y*. Les arguments *x*, *y* et *z* sont des noms de tige ('A', 'B' ou 'C') et *n* est un entier positif ou nul. Le déplacement d'un disque sera commandé par l'écriture à chaque mouvement du message :

« Déplace un disque de la tige *p* à la tige *q* »

Le jeu démarrera par l'appel :

hanoi(‘A’, ‘B’, ‘C’, *n*)

et l’appel :

hanoi(*x*, *y*, *z*, 0)

ne déclenchera aucun mouvement.

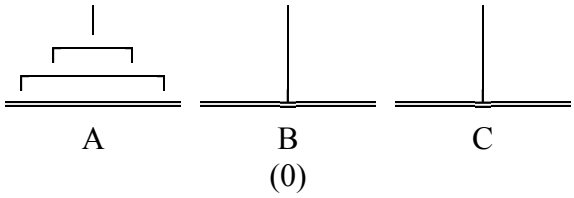
Le programme C est donc le suivant :

```
#include <stdio.h>

void hanoi(char x, char y, char z, int n)
   {
   if (n == 0)
      return;
   else
      {
      hanoi(x, z, y, n – 1);
      printf("Deplace un disque de la tige %c a la tige %c\n", x, z);
      hanoi(y, x, z, n – 1);
      }
   }

main()
   {
   /*
    * On joue avec deux disques
    */
   hanoi('A', 'B', 'C', 2);
   }
```

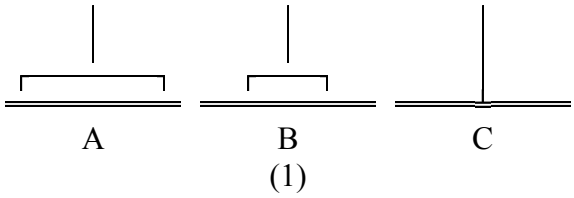
dont l’exécution donnera :

```
Deplace un disque de la tige A a la tige B
Deplace un disque de la tige A a la tige C
Deplace un disque de la tige B a la tige C
```

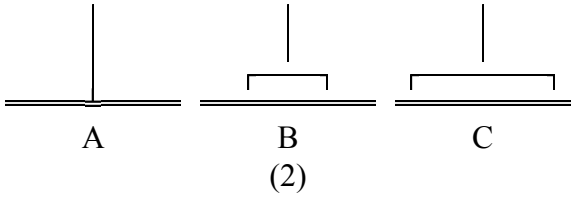
c’est à dire :



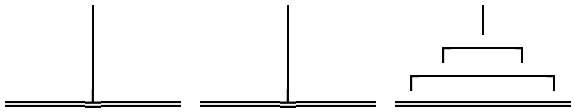
(0)



(1)



(2)



(3)