Programmation structurée en C
DESS IMTS

Ph. PREUX[1]
Université du Littoral Côte d'Opale
Calais

27 septembre 2000

[1] philippe.preux@univ-lille3.fr

Résumé

Ce polycopié a pour objectif d'être un support du cours d'initiation à la programmation structurée en C.

La maîtrise de la programmation est un apprentissage de longue haleine qui nécessite beaucoup de rigueur et beaucoup de temps passé devant un papier avec un crayon à la main et ensuite, encore plus de temps passé devant l'ordinateur.

La maîtrise des techniques de programmation des ordinateurs est un outil indispensable aujourd'hui pour tout étudiant en sciences. Qu'il soit mathématicien, physicien, chimiste ou biologiste, l'étudiant en science doit impérativement savoir programmer.

Notation utilisée

Dans ce document, nous utilisons des caractères en fonte `courrier` pour indiquer ce qui est frappé (les programmes en général) ou ce qui apparaît à l'écran de l'ordinateur.

Chaque chapitre se termine par une section « Ce qu'il faut retenir » qui reprend les points clés discutés dans le chapitre. Ces points doivent être **parfaitement** maîtrisés pour aborder la suite.

Chapitre 1

Introduction

Étant donné un problème à résoudre, son analyse consiste à le décomposer en sous-problèmes plus simples jusqu'à atteindre des problèmes élémentaires que l'ordinateur est capable de résoudre (voir figure 1.1).

Le langage de programmation permet de transcrire le résultat de l'analyse dans un formalisme compréhensible par l'ordinateur.

Il existe différentes classes de langages de programmation. Citons :

- langages procéduraux : Ada, Fortran, Pascal, Cobol, C, ...
- langages fonctionnels : Scheme, Lisp, ...
- langages objets : Java, C++, ...

À chaque classe de langages est associée une méthodologie de conception particulière.

Dans les langages procéduraux, on distingue toujours deux éléments fondamentaux :

- les données qui correspondent en quelques sortes aux variables en mathématiques. Il est important de mentionner immédiatement que les données ne se restreignent pas à des variables numériques ;
- les traitements qui s'appliquent à des données et correspondent en quelques sortes aux fonctions mathématiques. À nouveau, il faut noter que les traitements ne sont pas forcément de nature numérique, bien au contraire.

Il est fondamental qu'un programme soit auto-documenté : un programme dont on ne comprend pas le fonctionnement en le lisant est inutile. En effet, un programme a une vie : il est écrit par une personne, ou une équipe de personnes plus généralement ; ensuite, il peut être modifié par d'autres personnes, parfois sans aucun lien avec celles qui l'ont écrit. Si cette équipe ne comprend pas ce qui a été écrit, le programme devient inutile. Il faut savoir que la conception et la réalisation d'un logiciel sont des opérations très coûteuses.

Aussi, il est indispensable de prendre de bonnes habitudes de rédaction de programmes dès le début. Il faudra donc toujours présenter et documenter correctement les programmes que nous

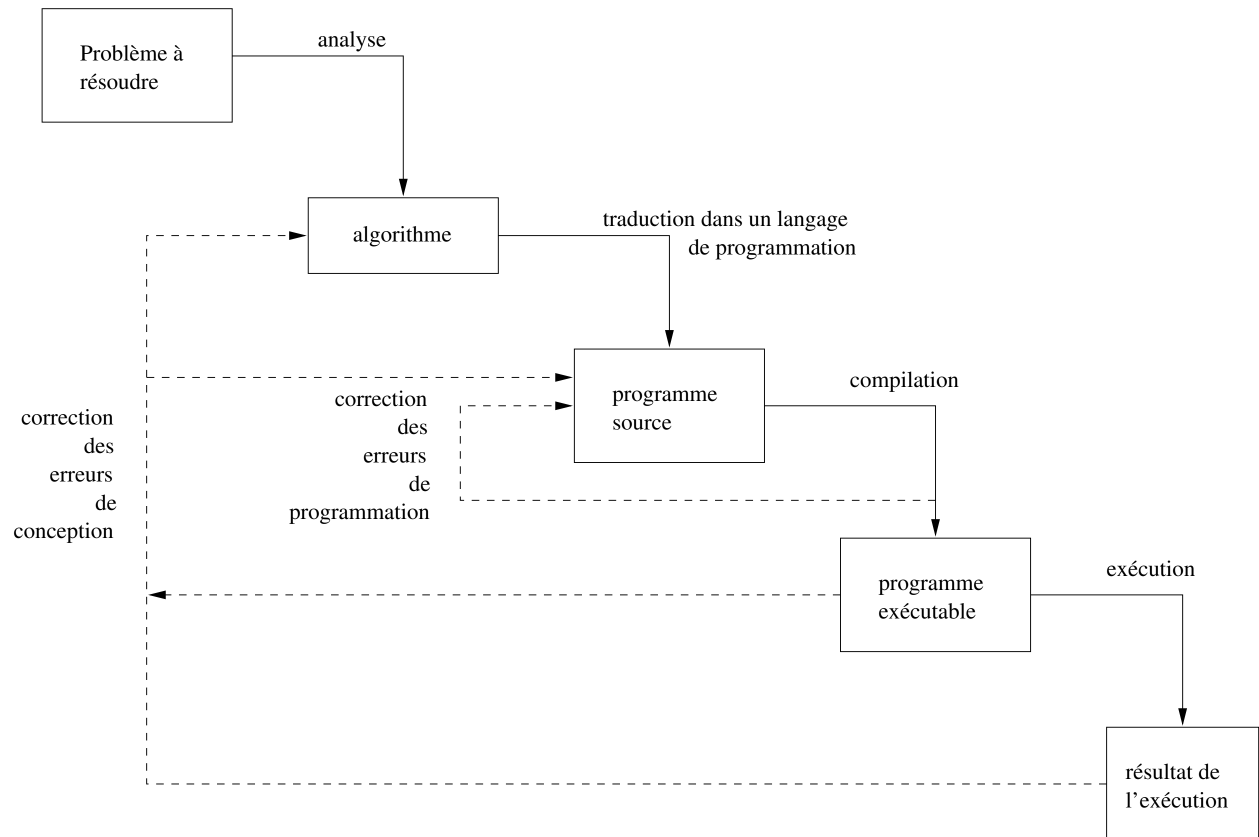


FIG. 1.1 – Résolution informatique d'un problème

écrions.

On utilise les arbres programmatiques pour exprimer les algorithmes. Les arbres programmatiques sont une représentation graphique indépendante du langage de programmation utilisé. Ils permettent de se concentrer sur l'algorithme et laisser de côté les problèmes de syntaxe particuliers à chaque langage.

On utilise le langage de programmation C dans le cadre du cours, des TD et des TP.

Chapitre 2

Les données, les types, les entiers

Parmi les données, on distingue immédiatement :
- les constantes qui possèdent une valeur qui ne change jamais ; par exemple, π ;
- les variables dont la valeur peut être modifiée.

Une donnée porte :
- un nom, ou identificateur, constitué de lettres, de chiffres et du caractère `_` en respectant la règle qu'un identificateur débute forcément par une lettre ou `_` (mais c'est à éviter). C fait la distinction entre majuscule et minuscule. Par exemple, `abc`, `x3j26` et `a_d` sont trois identificateurs valides en C ; `abc`, `Abc`, `aBC` sont trois identificateurs différents ;
- un type qui spécifie l'ensemble des valeurs que peut prendre la donnée et les opérations qui lui sont applicables.

Intuitivement, une donnée est une boîte dans laquelle on peut mettre une valeur.

Dans un programme, toute donnée doit être déclarée avant d'être utilisée ; son nom et son type ne peuvent pas être modifiés par la suite dans le programme.

Quelques types souvent rencontrés sont : entier, réel, caractère, chaîne de caractères.

2.1 Le type `int`

Le type `int` représente les valeurs entières, positives et négatives.

2.1.1 Introduction

La déclaration et l'utilisation la plus simple d'une variable de type `int` s'exprime de la manière suivante en C :

```
int i;

I = 10;
```

La première ligne `int i;` déclare une variable dont le nom est `i` et dont le type est `int`. Le `;` à la fin de la ligne indique la fin de la déclaration. Déclarée de type `int`, la variable `i` pourra contenir une valeur entière, positive ou négative.

La ligne suivante `i = 10;` affecte la valeur `10` à la variable `i`. Le `;` indique la fin de l'instruction.

Remarque d'ordre générale : toute déclaration et toute instruction se termine par un `;`.

Si l'on omet de déclarer une variable, le compilateur indique une erreur dans le programme. De même, si on essaie d'affecter une valeur qui ne correspond pas au type de la variable, une erreur est indiquée.

On peut également affecter le résultat d'un calcul :

```
int i;

i = 10 + 3 * 2;
```

Dans ce cas, `i` reçoit la valeur `16`. On peut également faire des calculs sur des variables :

```
int i, j;

i = 5;
j = 3;
i = i * j;
j = i * (j + 5) / 2;
```

Dans cet exemple, on note que l'on peut déclarer plusieurs variables de même type en une seule fois (ligne 1) ; après avoir affecté `5` à la variable `i` et `3` à la variable `j` (lignes 2 et 3), l'instruction de la ligne 4 affecte le résultat du produit de la valeur de `i` par la valeur de `j` ; la valeur de la variable `i` est donc modifiée de `5` à `15`. L'instruction de la ligne 5 modifie la valeur de `j` qui devient `15 * (3 + 5) / 2` soit `60`.

2.1.2 Opérations sur les valeurs de type `int`

Les opérateurs qui peuvent être utilisés sur des valeurs entières sont :

- `+` pour l'addition ;
- `-` pour la soustraction ;
- `*` pour le produit ;
- `/` pour la division entière : par exemple, `5 / 2` donne `2` ;
- `%` pour le reste de la division entière : par exemple, `5 % 2` donne `1`, le reste de la division entière de `5` par `2` ;

Lorsque plusieurs opérateurs sont utilisés dans une expression (par exemple, `2+3*12-4`), l'ordre dans lequel les opérateurs sont évalués doit être fixé $((2+3) \times (12-4)$, ou $((2+3) * 12) - 4$, ou $2+3*(12-4)$, ...). Pour cela, une *priorité* est associée à chaque opérateur. On a les règles suivantes :

- `*`, `/` et `%`, puis
- `+` et `-`.

et les expressions sont évaluées de la gauche vers la droite ; ainsi, `a / b + c` a pour valeur $\frac{a}{b} + c$, alors que `a / (b + c)` vaut $\frac{a}{b+c}$. L'exemple pus haut (`2+3*12-4` a donc pour valeur $2+(3\times12)-4 = 34$).

2.2 Initialisation de variables lors de leur déclaration

On peut initialiser la valeur d'une variable dès sa déclaration, plutôt que d'avoir une instruction qui le fait explicitement. Pour cela, on utilise la notation suivante :

```
int i = 10;
```

qui déclare la variable `i` de type `int` et l'initialise avec la valeur `10`.

2.3 Les constantes

Une constante est une donnée dont la valeur ne peut pas être modifiée. On la déclare de la manière suivante :

```
const int cst = -39;
```

qui déclare une constante dénommée `cst` de type `int` et dont la valeur est `-39`. Par la suite, il est interdit d'essayer de modifier la valeur de `cst`. Ainsi, l'instruction `cst = 28;` provoquera une erreur de compilation.

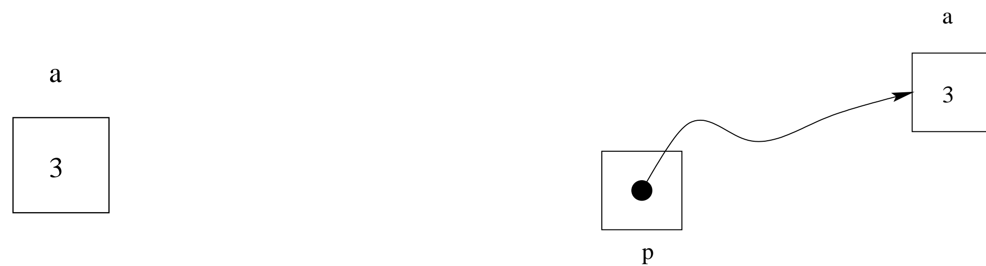
2.4 Les pointeurs

La notion de « pointeur » est une notion fondamentale en C. Aussi, nous la présentons immédiatement.

Une variable peut être vue comme une boîte portant un nom et contenant une valeur d'un certain type (voir la figure 2.1(a)).

Un pointeur est simplement une variable qui contient une « flèche » vers une autre variable ; plutôt que flèche, on parle de référence ou d'adresse. Le type du pointeur indique le type de la variable vers lequel il pointe (voir la figure 2.1(b)).

En C,



(a) Une variable est une boîte qui contient une valeur d'un certain type.

(b) Un pointeur est une flèche vers une boîte qui contient une valeur d'un certain type.

FIG. 2.1 – Variable et pointeur

```
int a;
```

déclare une variable dont le nom est `a` et dont le type est `int`, c'est-à-dire entier. Une fois déclarée, une valeur peut être affectée à une variable :

```
a = 3;
```

Pour déclarer un pointeur sur une variable de type entier, on écrit :

```
int *p;
```

On peut ensuite lui donner une valeur ; par exemple, pour le faire pointer sur le contenu de la variable `a`, on écrira :

```
p = & a;
```

`&` est un opérateur qui s'applique à une variable et fournit une référence sur cette variable. Dès que l'instruction précédente a été exécutée, la variable `a` et le pointeur `p` sont liés. On peut accéder à la valeur référencée par un pointeur par l'opérateur `*` ; si l'on écrit `*p`, c'est exactement comme si l'on écrit `a` ; la valeur de ces deux objets est `3`.

On peut également modifier la valeur de la variable `a` via le pointeur `p`. Ainsi, si l'on écrit :

```
*p = 7;
```

la valeur de la variable `a` est désormais `7`.

Supposons qu'une variable `b` ait été déclarée de type `int`, on peut ensuite écrire :

```
b = 8;
p = & b;
```

*p vaudra alors 8 puisque p référence désormais la variable b et non plus la variable a.

2.5 Exercice

```
int a, b, c;
int *p, *q;

a = 10; b = 20; c = 30;
p = &b; q = p;
/* valeurs de a, b, c, *p, *q ? */
*p = *p + 3;
/* valeurs de a, b, c, *p, *q ? */
*q = 27;
/* valeurs de a, b, c, *p, *q ? */
p = &c; *p = 18; *q = 23;
/* valeurs de a, b, c, *p, *q ? */
```

2.6 Ce qu'il faut retenir du chapitre

- la notion de variable
- la notion de constante
- la notion de type
- les caractéristiques du type int
- la notion de pointeur

Chapitre 3

Les arbres programmatiques ; les tests et les boucles

Les traitements concernent la réalisation des opérations ; les traitements agissent sur des données : les traitements prennent des données en entrée et fournissent des résultats.

On peut voir l'analogue avec une machine outil qui prend de la matière première en entrée et fournit des pièces usinées en sortie, ou le pétrin du boulanger qui reçoit en entrée farine, levure, sel et eau, et produit en sortie de la pâte à pains.

Dans le chapitre précédent, nous avons rencontré les traitements les plus simples :
- les opérations sur les nombres entiers ;
- l'affectation d'une valeur à une variable.

Nous décrivons maintenant les instructions avec lesquels TOUS[1] les programmes sont finalement réalisés. Nous les décrivons sous la forme d'arbres programmatiques. Outre les opérations de base et l'affectation vues au chapitre précédent, ces instructions se résument en trois structures :
- la séquence ;
- le test ;
- la boucle.

Ensuite, nous indiquerons la traduction des arbres programmatiques en C.

[1]véritablement TOUS : les programmes que nous écrirons en TP, les tableurs et autres traitements de textes vendus dans le commerce, les jeux vidéos, les navigateurs Internet, les compilateurs, ...

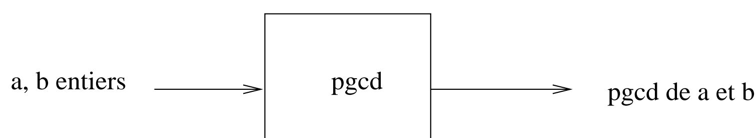


FIG. 3.1 – Un traitement consomme des données et fournit un résultat : ici, deux entiers (les données) sont transformées en un entier, résultat du traitement « pgcd ».

3.1 Les arbres programmatiques ; la séquence d'instructions

Pour exprimer le résultat de l'analyse d'un problème, nous utiliserons la notion d'arbre programmatique. Un arbre programmatique permet de se focaliser sur l'algorithme en laissant de côté les particularités de tel ou tel langage. Dans la suite du cours, nous aborderons en premier lieu les notions de traitement à l'aide d'arbres programmatiques ; de même, en TD et en TP, nous nous attacherons à obtenir l'arbre programmatique du problème posé. Ensuite seulement, nous le traduirons en C.

Nous allons immédiatement donner un exemple : donner l'arbre programmatique de l'action suivante :

1. saisie au clavier de la valeur de deux entiers ;
2. calcul de la moyenne de ces deux entiers ;
3. affichage de cette moyenne.

L'arbre est indiqué à la figure 3.2. Généralement, en informatique, un arbre se lit de haut en bas : sa racine est en haut, d'où partent des branches jusqu'à atteindre ses feuilles en bas. C'est simplement une question d'habitude.

On note les éléments suivants :

- un rectangle indiquant le nom de l'action : MOYENNE ;
- en dessous, un symbole $\boxed{\rightarrow}$ qui signifie « séquence d'instructions ». Les instructions en question sont situées en dessous de ce symbole. Les rectangles rattachés sous ce symbole sont exécutées dans l'ordre, de gauche à droite ;
- en dessous, à gauche, un rectangle indiquant l'action réalisée ici : « saisie de deux nombres au clavier » et en dessous, un rectangle indiquant les actions à réaliser : « saisir A et B » ;
- une fois cette action réalisée (les variables A et B contiennent donc les valeurs dont il faut calculer la moyenne), l'exécution se poursuit, en séquence, avec l'action qui se trouve à sa droite, soit « calcul de la moyenne ». À nouveau, l'action est commentée dans un rectangle rattaché à l'action effective de calcul de la moyenne et de son affectation à la variable MOY. Le symbole $\leftarrow$ est celui de l'affectation d'une valeur à une variable. Il signifie ici : affecter à la variable MOY la valeur de $\frac{A+B}{2}$;
- une fois cette action réalisée (les variables A et B contiennent donc toujours les valeurs dont il faut calculer la moyenne et MOY contient cette moyenne), l'exécution se poursuit, en séquence, avec l'action qui se trouve à droite, soit « affichage du résultat ». Encore une fois, l'action est commentée puis elle est décrite dans le rectangle en dessous ;
- puisqu'il n'y a plus d'action à la droite de celle-ci, l'exécution de l'action MOYENNE est terminée.

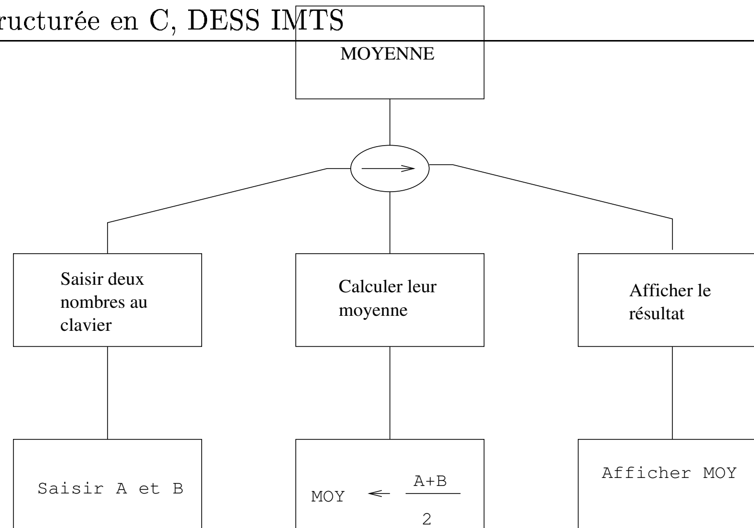


FIG. 3.2 – Arbre programmatique de l'action calcul de moyenne.

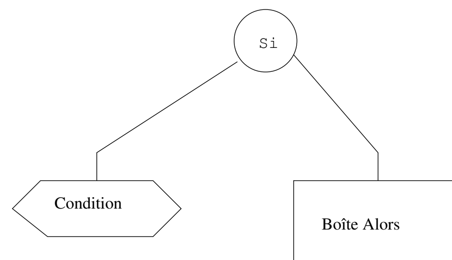


FIG. 3.3 – Arbre programmatique d'un test Si/Alors

3.2 Les tests

Les instructions de test permettent l'exécution d'instructions sous condition. Ces tests peuvent prendre deux formes, les tests Si/Alors et les tests Si/Alors/Sinon.

3.2.1 Test Si/Alors

Sous forme d'arbre programmatique, le test Si/Alors est représenté à la figure 3.3. On trouve une condition est une boîte d'instruction que nous nommerons « boîte-alors ».

Le principe d'exécution est le suivant :

1. la `condition` est évaluée ; sa valeur est soit VRAI soit FAUX ;
2. si elle est vraie, les instructions de la boîte-alors sont exécutées ;
3. si elle est fausse, aucune instruction n'est exécutée.

La figure 3.4 propose un exemple : la valeur d'une variable est testée et un message est affiché si le nombre est nul.

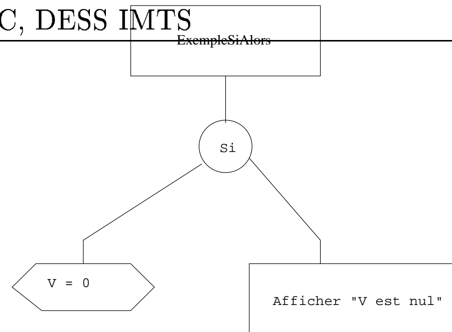


FIG. 3.4 – Exemple d'un test Si/Alors : la condition teste si la valeur de la variable est nulle. Dans ce cas, la boîte-alors est exécutée et affiche un message.

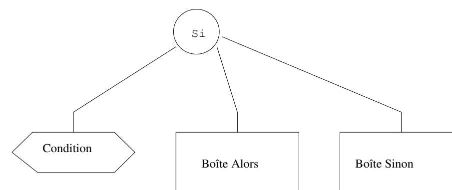


FIG. 3.5 – Arbre programmatique d'un test Si/Alors/Sinon

3.2.2 Test Si/Alors/Sinon

Sous forme d'arbre programmatique, le test Si/Alors/Sinon est représenté à la figure 3.5. On retrouve comme précédemment une condition, mais nous avons cette fois-ci deux boîtes d'instructions, la boîte-alors située à droite de la condition, et la « boîte-sinon » située à sa droite.

Le principe d'exécution est le suivant :

1. la `condition` est évaluée ; sa valeur est soit VRAI soit FAUX ;
2. si elle est vraie, les instructions de la boîte-alors sont exécutées ;
3. si elle est fausse, les instructions de la boîte-sinon sont exécutées.

Dans un test Si/Alors/Sinon, soit la boîte-alors soit la boîte-sinon est exécutée ; à l'exécution du test, l'une des deux est forcément exécutée mais jamais les deux.

La figure 3.6 propose un exemple : la valeur d'une variable est testée et un message est affiché en fonction de la parité du nombre.

3.3 Les boucles

Les boucles permettent de réaliser plusieurs fois de suite la même séquence d'instructions. Il en existe plusieurs formes et nous traiterons des boucles Tant-que et des boucles Pour.

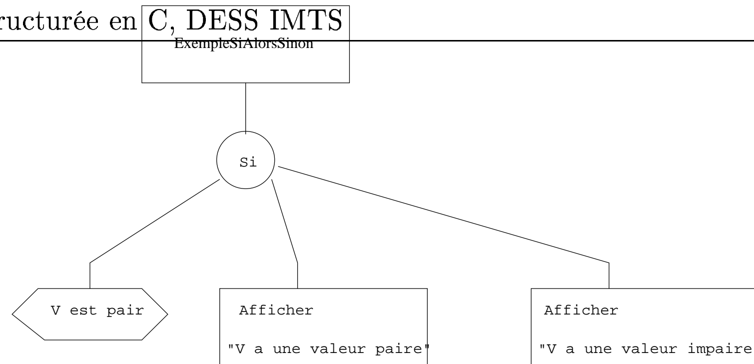


FIG. 3.6 – Exemple d'un test Si/Alors/Sinon : la condition teste si la valeur de la variable est paire. Selon le cas, la boîte-alors ou la boîte-sinon est exécutée et affiche un message.

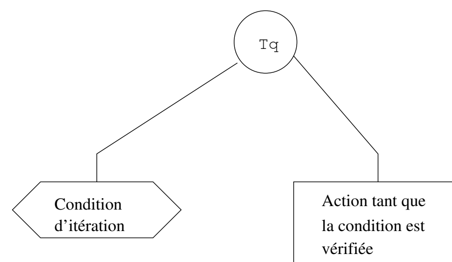


FIG. 3.7 – Arbre programmatique d'une boucle Tant-que

3.3.1 La boucle Tant-que

Dans une boucle Tant-que, la séquence d'instructions est répétée tant qu'une certaine condition est vérifiée. L'arbre programmatique correspondant est indiqué à la figure 3.7.

Le principe d'exécution d'une boucle Tant-que est le suivant :

1. la condition est calculée. Comme dans un test, sa valeur est VRAI ou FAUX ;
2. si la condition est vérifiée, l'action indiquée à sa droite est exécutée. À l'issue de l'exécution de cette action, on revient à l'étape 1 ;
3. si la condition n'est pas vérifiée, l'exécution du Tant-que est terminée.

On note que si la condition n'est pas vérifiée lors de sa première évaluation, l'action indiquée n'est jamais exécutée.

Règle à respecter : il faut impérativement s'assurer que la condition d'itération devient fausse à un moment pour en sortir.

Considérons un exemple : écrire une action qui calcule la somme des 10 premiers entiers naturels (les entiers de 1 à 10). L'arbre programmatique en est indiqué à la figure 3.8. Pour cela, on considére une variable I qui égrène les valeurs de 1 à 10 et, à chaque itération, on ajoutera sa valeur à une

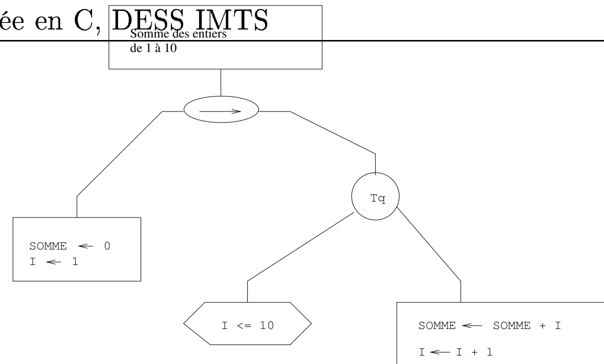


FIG. 3.8 – Un arbre programmatique pour une action consistant à faire la somme des entiers de 1 à 10 et utilisant une boucle Tant-que.

autre variable, SOMME initialisée au préalable à 0. Par ailleurs, à chaque itération, la variable I est incrémentée. Lorsque I atteint la valeur 11, on a terminé, ce qui explique la condition d'itération du Tant-que.

On peut composer des actions plus complexes en utilisant plusieurs traitements dans une même action. Ainsi, on peut définir une action qui, pour les entiers de 5 à 1, affiche s'il est pair ou impair (voir figure 3.9).

3.3.2 La boucle Pour

Dans une boucle Pour, on connaît le nombre d'itérations à réaliser. L'arbre programmatique correspondant est indiqué à la figure 3.10.

Le principe d'exécution d'une boucle Pour est le suivant :

1. une variable (appelée « indice de boucle ») est utilisée ; elle va successivement prendre les valeurs indiquées dans l'intervalle. Elle est tout d'abord initialisée avec la borne inférieure de l'intervalle ;
2. l'action associée est exécutée ; la valeur de l'indice peut être utilisée ;
3. l'indice de boucle est incrémenté ;
4. si l'indice de boucle est toujours inférieur ou égal à la borne supérieure de l'intervalle, on retourne à l'étape 2 ; sinon, l'exécution de la boucle est terminée.

Considérons à nouveau l'exemple traité avec la boucle Tant-que : écrire une action qui calcule la somme des 10 premiers entiers naturels (les entiers de 1 à 10). On utilise cette fois-ci une boucle Pour. L'arbre programmatique est indiqué à la figure 3.11. Cette fois-ci, la variable I qui égrène les valeurs de 1 à 10 est l'indice de boucle. L'intervalle dans lequel sa valeur varie est spécifié. Son incrémentation est cette fois automatique. L'action action consiste donc simplement à ajouter le contenu courant de l'indice de boucle à la variable SOMME.

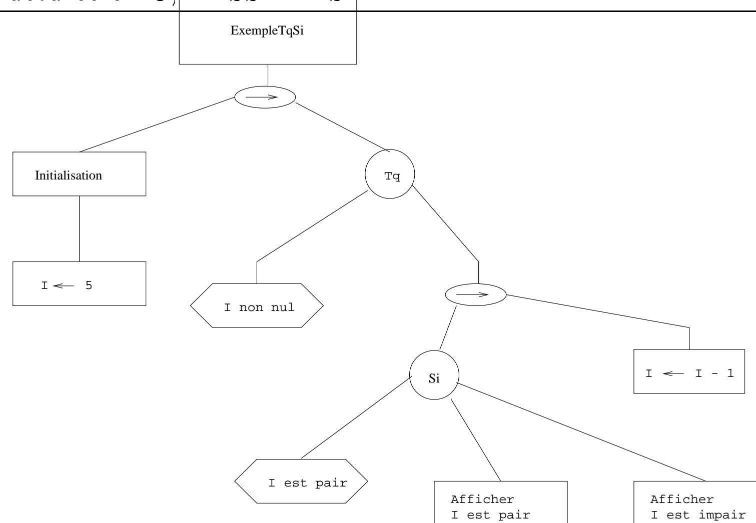


FIG. 3.9 – Un arbre programmatique pour une action consistant, pour les entiers de 5 à 1, à afficher s'il est pair ou impair. On voit que l'on peut composer des actions complexes en combinant les traitements de base : séquence, test, boucle.

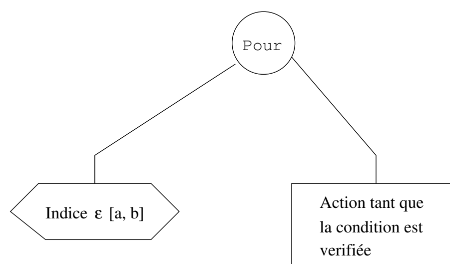


FIG. 3.10 – Arbre programmatique d'une boucle Pour

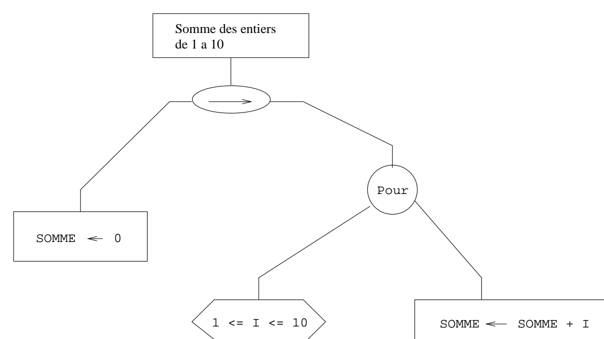


FIG. 3.11 – Un arbre programmatique pour une action consistant à faire la somme des entiers de 1 à 10 et utilisant une boucle Pour.

La traduction de ce problème à l'aide d'une boucle Pour est beaucoup plus naturelle qu'avec une boucle Tant-que ; en effet, on connaît à l'avance le nombre d'itérations à réaliser. Aussi, la bonne solution à ce problème consiste à utiliser une boucle Pour et non une boucle Tant-que.

3.4 Exercices

Donner un arbre programmatique pour les problèmes suivants :

1. échanger la valeur de deux variables : avant l'action, la variable A contient une valeur x, la variable B contient une valeur y ; à la suite de l'action, A contient la valeur y et B contient la valeur x ;
2. avant l'action, la variable A contient une valeur x, la variable B contient une valeur y ; à la suite de l'action, A contient la valeur la plus petite parmi x et y et B contient la valeur la plus grande parmi x et y ;
3. avant l'action, trois variables A, B et C qui sont les coefficients d'une équation du second degré (A x^2+ B x+ C = 0). Afficher les solutions de l'équation en considérant que certaines variables peuvent avoir une valeur nulle ;
4. avant l'action, trois variables A, B et C qui contiennent respectivement les valeurs x, y et z. À l'issue de l'action, la valeur de A est inférieure ou égale à celle de B, elle-même inférieure ou égale à celle de C ;
5. avant l'action, on a une variable N de type entier. À l'issue de l'action, X contient sa racine carrée par excès, c'est-à-dire le plus petit entier tel que $X^2 \geq N$;
6. avant l'action, deux entiers A et B. L'action affiche la valeur de A élevée à la puissance de la valeur de B ;
7. avant l'action, la variable A contient une année (un entier). L'action détermine si c'est une année bissextile et affiche un message correspondant ;
8. avant l'action, deux horaires, l'un de départ, l'autre d'arrivée sous la forme heure, minute et secondes (HD, MD, SD, HA, MA, SA) dans la même journée ; à l'issue de l'action, la durée du trajet est affichée ;
9. avant l'action, la variable N contient un entier. L'action calcule et affiche sa factorielle (attention aux cas particuliers) ;
10. avant l'action, la variable N contient un entier. L'action saisit N nombres entiers au clavier et affiche leur somme ;
11. saisir une suite d'entiers au clavier terminée par la valeur -1 et afficher leur somme ;
12. avant l'action, on a une valeur N entière ; à l'issue de l'action est affichée la valeur de :

$$\sum_{i=1}^{i=N} \frac{(-1)^i}{i}$$

13. multiplication égyptienne : c'est une manière de calculer un produit en ne faisant que des additions, des multiplications par 2 et des divisions entières par 2. Considérons un exemple : quel est le produit de 36 par 43 ? On effectue la division entière 43 par 2 (on aurait pu prendre 36 à la place bien entendu) et on recommence sur le résultat jusqu'à atteindre 1 ; en vis-à-vis de chaque résultat, on écrit le produit de 36 par 2, puis encore par 2, ... On obtient :

$$
\begin{array}{ccccccccccc}
43 & \xrightarrow{/2} & 21 & \xrightarrow{/2} & 10 & \xrightarrow{/2} & 5 & \xrightarrow{/2} & 2 & \xrightarrow{/2} & 1 \\
36 & \xrightarrow{\times 2} & 72 & \xrightarrow{\times 2} & 144 & \xrightarrow{\times 2} & 288 & \xrightarrow{\times 2} & 576 & \xrightarrow{\times 2} & 1152
\end{array}
$$

On fait ensuite la somme des multiples de 36 apparaissant dans la deuxième ligne qui correspondent aux valeurs impaires de la première ligne, soit 36, 72, 288 et 1152 ; c'est le résultat du produit de 36 par 43.

Donner l'arbre programmatique de la multiplication égyptienne en considérant que les valeurs à multiplier sont dans deux variables A et B avant l'action, et que le produit est affiché à l'écran à l'issue de l'action.

Chapitre 4

Instructions C élémentaires et structures de contrôle

Dans ce chapitre, nous passons rapidement en revue les instructions de base et les structures de contrôle de C.

4.1 Identificateurs

En C, les identificateurs de variable, type, fonction, ... sont écrits avec des lettres, des chiffres et des soulignés. Un identificateur commence forcément par une lettre ou un souligné (les soulignés doivent être évités). C fait la distinction entre lettres majuscules et lettres minuscules.

4.2 Instructions élémentaires du C

4.2.1 Quelques opérateurs particuliers en C

+= et autres opérateurs du même genre

Le langage C est compact : un certain nombre d'opérations couramment rencontrées dans les programmes ont ainsi une écriture abrégée qu'il faut connaître. Ainsi, il est courant d'avoir une instruction du genre :

```
a = a + 5;
```

Dans ce cas, on peut écrire :

```
a += 5;
```

Ce raccourci peut être utilisé avec les autres opérateurs tels `-`, `*`, `/` et `%`.

Les opérateurs ++ et -

On rencontre également fréquemment l'instruction :

ou

`j = j - 1;`

Dans ces cas, comme on vient de le voir, on peut écrire :

`i += 1;`

et

`j -= 1;`

On peut écrire cela de manière encore plus compacte :

`i ++;`

et

`j --;`

Il nous faut ici préciser une subtilité du langage C. En C, toute expression possède une valeur ; ainsi, comme on s'y attend, une variable possède une valeur, l'addition de deux variables possède une valeur ; mais aussi, une affectation possède une valeur : l'expression `a = 3 + 8;` vaut 11. Donc, `i ++` et `j --` possèdent une valeur ; si `i` et `j` vallent initialement `10` toutes les deux, `i++` vaut `10` et `j--` vaut `10`. On peut également écrire `++i` et `-- j` ; `++i` vaut `11` et `--j` vaut `9`.

Opérateur ? :

L'opérateur `?:` est dénommé « opérateur conditionnel ». Sa forme générale est la suivante :

`condition ? valeur_1 : valeur_2`

La valeur de cette expression est déterminée de la manière suivante : si la `condition` est vérifiée, l'expression prend la `valeur_1` ; si la `condition` n'est pas vérifiée, l'expression prend la `valeur_2`.

Nous montrons son utilisation sur un exemple. Soit le traitement suivant :

```
if (a == 10)
  b = 9;
else
  b = 1;
```

qui affecte 9 à la variable `b` si `a` vaut 10, affecte 1 à `b` sinon. On peut l'écrire de manière beaucoup plus compacte sous la forme :

`b = a == 10 ? 9 : 1`

Les valeurs binaires

C a été conçu pour pouvoir manipuler facilement des données exprimées en binaires. On peut écrire les nombres en base 10, mais aussi dans les bases 8 et 16. Pour indiquer qu'un entier est écrit en base 8, on le fait précéder d'un `0` ; pour indiquer qu'un entier est écrit en base 16, on le fait

précéder de 0x. Par exemple, 015 est une valeur en base 8 qui équivaut à 13 en base 10 ; 0x25 est une valeur en base 16 qui équivaut à 37 en base 10. Les nombres en base 8 sont écrits avec les chiffres compris entre `0` et `7` ; les nombres en base 16 sont écrits avec les chiffres de `0` à `9` et les lettres `a` à `f` pour les chiffres compris entre 10 et 15.

Des opérateurs binaires existent en C. Considérons que `a` et `b` sont deux variables entières, `a` vaut `0x36` (soit 54 en base 10 et `00110110` en base 2), `b` vaut `0x4e` (soit 78 en base 10 et `01001110` en base 2), on a :

- `a & b` indique le ET logique, soit `00000110` en base 2, c'est-à-dire `6` en base 10 ;
- `a | b` indique le OU logique, soit `01111110` en base 2, c'est-à-dire `126` en base 10 ;
- `a ^ b` indique le OU-exclusif, soit `01111000` en base 2, c'est-à-dire `120` en base 10 ;
- `~a` indique la négation de `a`, soit `11001001` en base 2, c'est-à-dire `201` en base 10 ;
- `a << 2` indique le décalage vers la gauche de `2` bits de la valeur de `a`, soit `11011000` en base 2, soit `216` en base 10 ;
- `a >> 3` indique le décalage vers la droite de `3` bits de la valeur de `a`, soit `00000110` en base 2, soit `6` en base 10 si l'on considère la valeur initiale de `a` (`54`).

4.2.2 Le type `char`

Une variable de type `char` contient un caractère. Une constante de type `char` est indiquée entre apostrophes, telle `'e'`.

Une valeur entière est associée à chaque caractère. On peut comparer des caractères entre-eux en fonction de cet ordre.

Pour passer d'un caractère à son suivant, on lui ajoute simplement 1, par exemple `'e' + 1` ; pour passer à son prédécesseur, on lui retire 1 : `'e' - 1`. Pour obtenir son code, on peut l'affecter à une variable entière ; la conversion est réalisée automatiquement :

```
{
  int a;

  a = 'e';
}
```

La conversion inverse est tout aussi simple : on peut écrire :

```
{
  char c;

  c = 48;
}
```

Sachant que 48 est le code du 'A', l'instruction précédente est équivalente à écrire c = 'A', mais elle est moins claire. Aussi, on évitera d'écrire ce genre d'instructions.

Le retour-chariot s'écrit `'\n'`, la tabulation `'\t'`.

Fonctions pré-définies sur les caractères

Un certain nombre de fonctions sont pré-définies sur les caractères. Pour les utiliser, il faut écrire `#include <ctype.h>` en début de programme. On peut ensuite utiliser :

- `isalnum (c)` indique si `c` est une lettre ou un chiffre ;
- `isalpha (c)` indique si `c` est une lettre ;
- `isdigit (c)` indique si `c` est un chiffre ;
- `islower (c)` indique si `c` est une lettre minuscule ;
- `ispunct (c)` indique si `c` est un caractère qui n'est ni une lettre, ni un chiffre, ni un caractère blanc ;
- `isspace (c)` indique si `c` est un caractère blanc (espace, tabulation, retour-chariot, ...) ;
- `isupper (c)` indique si `c` est une lettre majuscule ;
- `toupper (c)` qui, si `c` contient une lettre minuscule, la transforme en son équivalent en majuscule, sinon ne modifie pas la valeur de `c` ;
- `tolower (c)` qui, si `c` contient une lettre majuscule, la transforme en son équivalent en minuscule, sinon ne modifie pas la valeur de `c`.

4.2.3 Déclarations de variables et types

En C, un bloc d'instructions débute toujours par une `{` et se termine toujours par `}`. Après chaque début de bloc `{`, on peut déclarer des types et des variables qui ne sont visibles que dans ce bloc.

On peut également déclarer des variables en dehors des fonctions. Ce sont alors des variables globales au fichier dans lequel elles sont déclarées. Elles sont visibles après leur déclaration.

4.2.4 Déclarations de constantes

On peut déclarer une constante comme suit :

```
const int constante_entiere = 56;
const float constante_reelle = 78.45;
const char caractere_constant = 'e';
```

4.2.5 Les entrées-sorties en C

Les entrées-sorties peuvent être réalisées de différentes manières en C. Nous en décrivons une, la plus simple et la plus générale.

En début de programme C, il faut écrire :

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

Affichage

Pour afficher un message à l'écran, on écrit :

```
printf ("Ceci est un message\n");
```

Les deux caractères `\n` à la fin du message indiquent qu'il faut ensuite passer à la ligne.

Pour afficher le contenu d'une variable `v` de type `int`, on écrit :

```
printf ("%d", v);
```

La notation `%d` indique qu'il faut afficher le contenu d'une variable de type entier ; le nom de cette variable est indiquée ensuite.

En fonction du type de la variable à afficher, on utilise différents caractères après `%` :

- `%c` pour une variable de type `char` ;
- `%f` pour une variable de type `float` ou `double` ;
- `%s` pour une variable de type chaîne de caractères.

On peut afficher le contenu de plusieurs variables et les mélanger aisément dans un texte :

```
int a = 10;
float x = 6.3;
char c = 'm';

printf ("Le contenu de a est %d,\ncelui de x est %f\net celui de c est %c", a, x, c);
```

qui affichera :

```
Le contenu de a est 10,
celui de x est 6.3
et celui de c est m
```

On note que l'on peut passer à la ligne dans le message à l'aide de `\n`.

Il faut prendre garde à l'ordre dans lequel sont mis les `%` qui doit correspondre à l'ordre des variables, ainsi qu'à leur nombre : en effet, aucun contrôle n'est effectué quant à la cohérence entre les paramètres de `printf`.

Saisie au clavier

Pour saisir la valeur d'une variable au clavier, le principe est le même que pour l'affichage ; la fonction à utiliser est `scanf`.

Ainsi, si l'on a trois variables :

```
int a;
float x;
char c;
```

on pourra saisir leur contenu au clavier en écrivant :

```
scanf ("%d%f%c", &a, &x, &c);
```

Le type des variables est indiqué comme dans un `printf` ; les variables apparaissent ensuite avec l'opérateur `&` parce qu'il faut fournir une référence sur la variable et non la variable elle-même. Ce `&` est très important et si on l'oublie, le compilateur ne le signale pas. (Voir également page 34 pour des compléments.)

Comme pour `printf`, il faut être vigilant quant aux paramètres de `scanf` sur lesquels le compilateur n'effectue aucun contrôle de validité.

4.3 Un exemple de programme C

Nous donnons ci-dessous le programme correspond à l'arbre programmatique de la figure 4.1.

```
/*
 * Programme moyenne.c
 *
 * 19 decembre 1999
 *
 * Ph. Preux, ULCO
 *
 * Ce programme saisit deux nombres, calcule leur moyenne
 * et l'affiche a l'ecran.
 */
#include <stdio.h>

main (void)
{
  int i, j, moy;
```

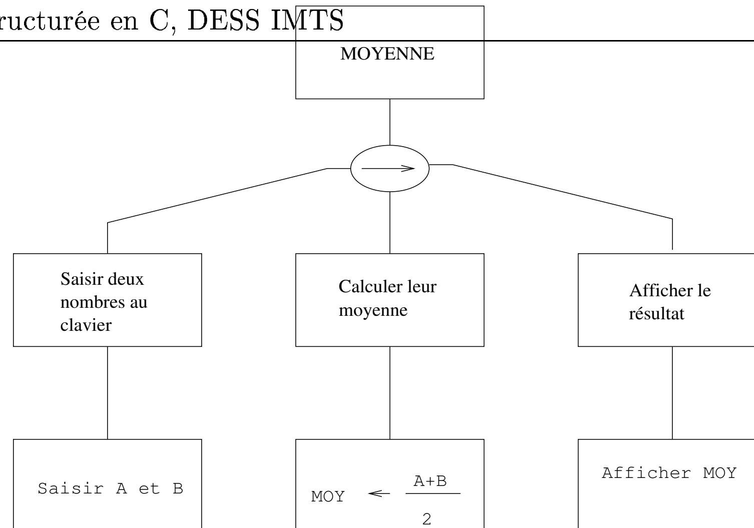


FIG. 4.1 – Arbre programmatique de l'action consistant à saisir deux nombres au clavier, calculer leur moyenne et l'afficher à l'écran.

```
  scanf ("%d%d", &i, &j);
  moy = (i + j) / 2;
  printf ("La moyenne de %d et %d est %d\n", i, j, moy);
}
```

Dans ce programme, on trouve les éléments suivants :

- En C, les commentaires sont situés entre `/*` et `*/` : tout ce qui est compris entre ces deux marques est un commentaire ; celui-ci peut être rédigé sur plusieurs lignes.
- la ligne `#include <stdio.h>` nécessaire car le programme effectue des entrées-sorties (voir le paragraphe 4.2.5) ;
- la ligne `main (void)` qui indique le début du programme principal : en C, le programme principal est toujours une fonction qui s'appelle `main`. Comme pour toute fonction (voir la page 37), les paramètres de cette fonction sont ensuite indiqués entre `()`. Lorsqu'une fonction n'a pas de paramètre, on indique `void` ;
- on trouve ensuite les instructions de la fonction `main` entre `{}` :
  - la déclaration de trois variables locales `i`, `j` et `moy` de type `int` ;
  - les instructions du programme.

4.4 Déclaration de types

On peut déclarer de nouveaux types en C de la manière suivante :

```
typedef int Longueur;
```

Cette instruction déclare le type `Longueur` comme équivalent au type `int`. On peut ensuite déclarer des variables de type `Longueur` :

```
Longueur l;
```

qui déclare une variable `l` de type `Longueur`.

On peut également déclarer un type pointeur sur un entier comme suit :

```
typedef int *Ptr_int;
```

4.4.1 Type énuméré

On peut déclarer un type par énumération des valeurs que peut prendre une variable de ce type :

```
typedef enum { Bleu, Vert, Rouge, Jaune, Blanc, Noir} Couleur;
```

4.4.2 Enregistrement

On peut décrire et déclarer des enregistrements qui portent le nom de « structures » en C :

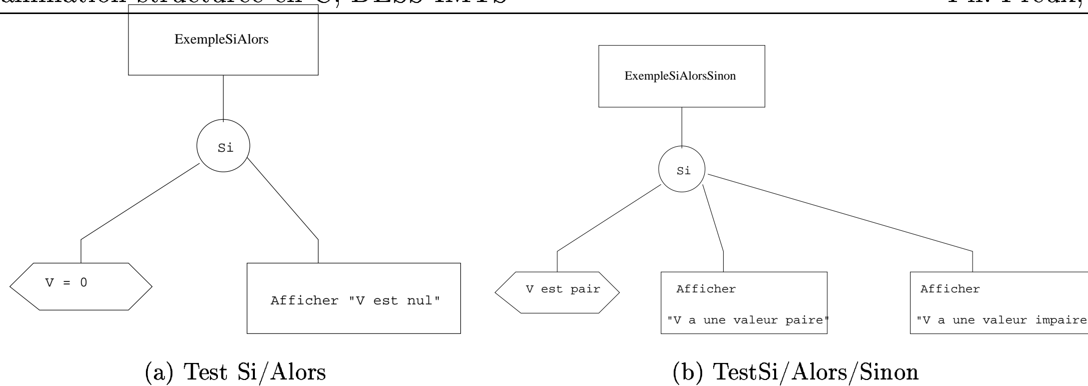
```
typedef struct {
  int partie_reelle, partie_imaginaire;
} Complexe;
```

Cette instruction déclare un type structure dénommé `Complexe` et composé de deux champs, `partie_reelle` et `partie_imaginaire` de type `int` tous les deux.

On accède aux champs d'une variable de type enregistrement à l'aide de la notation pointée :

```
Complexe c;

c. partie_reelle = 8;
c. partie_imaginaire = -3;
```

initialise la variable `c` comme étant le nombre complexe $8 - 3i$.



(a) Test Si/Alors (b) TestSi/Alors/Sinon

FIG. 4.2 – Les tests.

4.5 Structures de contrôle

4.5.1 Les tests

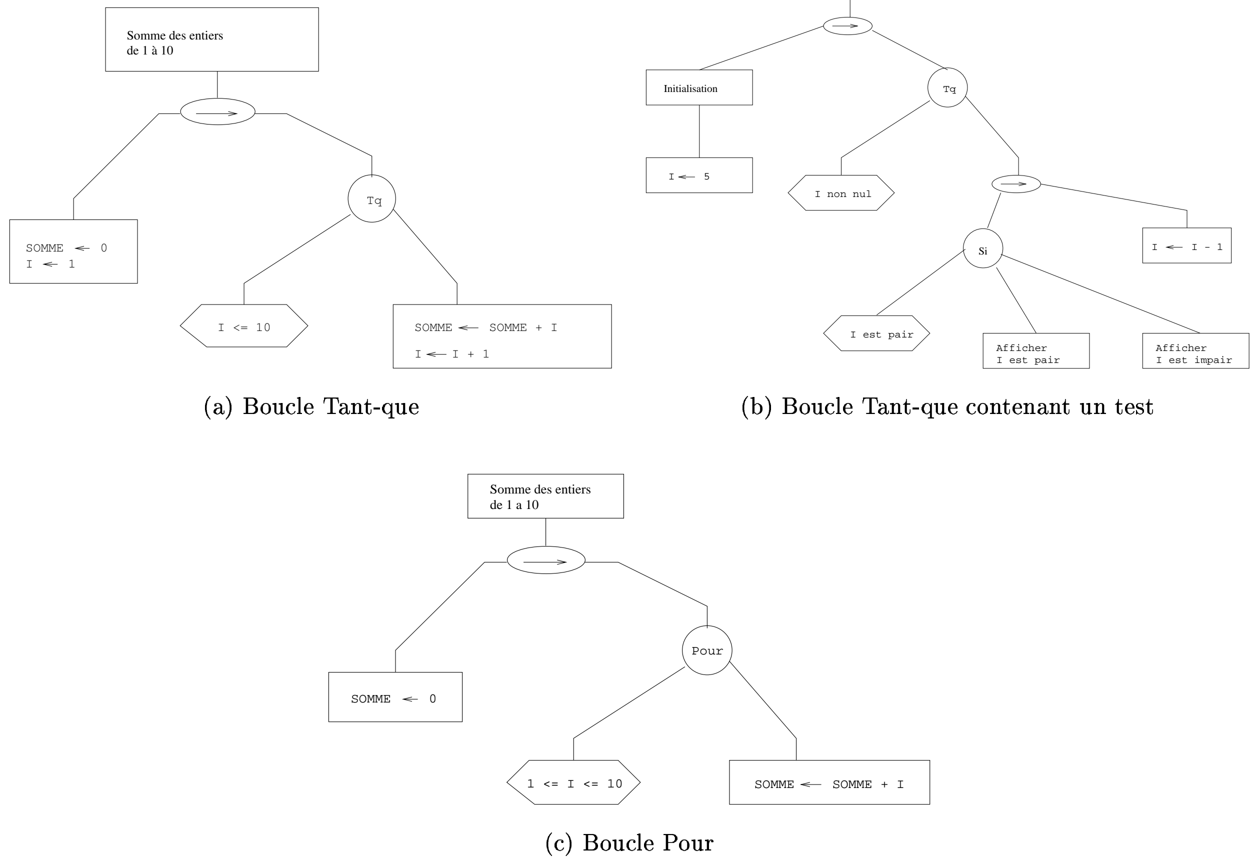
L'arbre programmatique de la figure 4.2(a) se traduit de la manière suivante en C :

```
{
  int v;

  if (v == 0)
    printf ("v est nul\n");
}
```

Nous remarquons que le mot-clé `then` n'existe pas en C. Par ailleurs, les conditions sont toujours entre `()`. Nous insistons sur la distinction entre `=` et `==` : `=` dénote l'affectation alors que `==` dénote le test d'égalité. Il faut être très prudent car en C, on peut également écrire `if (v = 0)` qui est syntaxiquement correct, donc ne déclenchera aucun message d'erreur de la part du compilateur.

L'arbre programmatique de la figure 4.2(b) se traduit de la manière suivante en C :

```
{
  int v;

  if (v % 2 == 0)
    printf ("v a une valeur paire\n");
  else
```



(a) Boucle Tant-que (b) Boucle Tant-que contenant un test

(c) Boucle Pour

FIG. 4.3 – Les boucles.

```
    printf ("v a une valeur impaire\n");
}
```

`v % 2` dénote le reste de la division eucliedienne de

4.5.2 Les boucles

Boucle tant-que

L'arbre programmatique de la figure 4.3(a) se traduit de la manière suivante en C :

```
{
  int somme = 0;
  int i = 1;
```

```
  while (i <= 10) {
    somme += i;
    i ++;
  }
}
```

L'arbre programmatique de la figure 4.3(b) se traduit de la manière suivante en C :

```
{
  int i;

  i = 5;
  while (i != 5) {
    if (i % 2 == 0)
      printf ("%d est pair\n");
    else
      printf ("%d est impair\n");
    i --;
  }
}
```

Boucle pour

L'arbre programmatique de la figure 4.3(c) se traduit de la manière suivante en C :

```
{
  int i;
  int somme = 0;

  for (i = 1; i <= 10; i++)
    somme += i;
}
```

L'instruction `for (i = 1; i <= 10; i++)` indique les choses suivantes :

- entre `(` et le premier `;`, on indique un traitement qui doit être exécuté avant la première itération : c'est donc l'initialisation de la boucle, en l'occurence, l'initialisation de la valeur de l'indice de boucle ;
- entre les deux `;`, on indique la condition d'itération de la boucle ;

entre le deuxième ; et), on indique une instruction qui sera exécutée à l'issue de chaque itération ; c'est à cet endroit qu'il faut modifier la valeur de l'indice de boucle.

4.5.3 Conditions composées

On peut composer plusieurs conditions pour réaliser des conditions composées. Les opérateurs sont les suivants :

- `c1 && c2` pour la conjonction de deux conditions `c1` et `c2` ;
- `c1 || c2` pour la disjonction de deux conditions `c1` et `c2` ;
- `! c` pour la négation de la condition `c`.

4.6 Les nombres réels

Il existe deux types de nombres réels en C, les `float` et les `double`. Les `float` sont moins précis que les `double`. Il ne faut pas les confondre dans les appels de fonction.

4.6.1 Entrées-sorties de nombres réels

Pour afficher la valeur d'un nombre réel, on a vu que l'on utilise `printf ("%f", x);` (voir page 27).

Pour saisir au clavier la valeur d'un nombre réel, il faut distinguer les deux types :

- pour un `float`, on écrit : `scanf ("%f", &variable_de_type_float);`
- pour un `double`, on écrit : `scanf ("%lf", &variable_de_type_double);`

La différence est subtile (le `l` entre `%` et `f`), mais elle fait tout : sans ce `l`, cela ne fonctionne pas, et l'erreur est très difficile à détecter dans un programme (aucun message d'erreur, ni à la compilation, ni à l'exécution).

4.6.2 Fonctions mathématiques

Pour l'utilisation des fonctions logarithmiques ou trigonométriques, on doit mettre la ligne suivante en début de programme :

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

Ensuite, on peut utiliser les fonctions suivantes :

- `sqrt (x)` qui fournit $\sqrt{x}$;
- `sin (x)`, `cos (x)`, `tan (x)` qui fournissent respectivement le sinus, le cosinus et la tangente de l'angle `x` exprimé en radians ;
- `asin (x)`, `acos (x)`, `atan (x)` qui fournissent respectivement l'arcsinus, l'arccosinus, l'arc-tangente de `x` (l'angle résultat est exprimé en radians) ;

– log (x), log10 (x) qui fournissent le logarithme naturel et le logarithme en base 10 de x ;
– exp (x) qui fournit e^x.

avec un paramètre de type double.

On peut également utiliser la constante M_PI dont la valeur est une approximation de π.

4.6.3 Compatibilité et conversion entre entiers et réels

On peut combiner des entiers avec des flottants dans une expression en C. Par exemple, on peut écrire :

```
int a;
int b = 3;
double x = 1.675;

a = 0.89 + b * x;
```

La valeur de a est ensuite 5, c'est-à-dire la partie entière de 5.915, résultat de l'expression $0.89 + 3 \times 1.675$.

D'une manière générale, nous avons la règle de conversion suivante : quand on affecte une valeur réelle à une variable de type int, la valeur affectée est la partie entière de la valeur réelle ; ce n'est jamais une valeur arrondie.

4.6.4 Conversion

D'une manière générale, une conversion[1] peut s'effectuer de la manière suivante :

```
(type) expression
```

qui convertit l'expression dans le type spécifié. Par exemple, on peut écrire :

```
char c;
int i;
float f;

c = (char) 67;
i = (int) 'e';
f = (float) i;
i = (int) 6.97;
```

[1]on parle également de « trans-typage » ou *cast*

La première instruction affecte le caractère `C` à la variable `c` ; la deuxième ligne affecte la valeur `101` à la variable `i` ; la troisième ligne affecte la valeur `101.0` à la variable `f` ; la dernière ligne affecte la valeur `6` à la variable `i`.

4.7 Exercices

1. écrire un programme qui saisit un entier positif, calcul de la factorielle et l'affiche ;
2. écrire un programme qui saisit les valeurs des coefficients d'un polynome du 2^{e}degré, calcule puis affiche les solutions de l'équation (en considérant que certaines variables peuvent avoir une valeur nulle) ;
3. écrire un programme qui saisit un entier, calcule sa racine carrée par excès et l'affiche ;
4. écrire un programme qui saisit une suite d'entiers au clavier terminée par la valeur -1 et affiche leur somme ;
5. multiplication égyptienne : c'est une manière de calculer un produit en ne faisant que des additions, des multiplications par 2 et des divisions entières par 2. Considérons un exemple : quel est le produit de 36 par 43 ? On effectue la division entière 43 par 2 (on aurait pu prendre 36 à la place bien entendu) et on recommence sur le résultat jusqu'à atteindre 1 ; en vis-à-vis de chaque résultat, on écrit le produit de 36 par 2, puis encore par 2, ... On obtient :

   $$43 \xrightarrow{/2} 21 \xrightarrow{/2} 10 \xrightarrow{/2} 5 \xrightarrow{/2} 2 \xrightarrow{/2} 1$$
   $$36 \xrightarrow{\times 2} 72 \xrightarrow{\times 2} 144 \xrightarrow{\times 2} 288 \xrightarrow{\times 2} 576 \xrightarrow{\times 2} 1152$$

   On fait ensuite la somme des multiples de 36 apparaissant dans la deuxième ligne qui correspondent aux valeurs impaires de la première ligne, soit 36, 72, 288 et 1152 ; c'est le résultat du produit de 36 par 43.

   Écrire un programme qui saisit deux nombres, applique la multiplication égyptienne sur ces deux nombres et affichele résultat.

Chapitre 5

Les fonctions et le passage de paramètres

5.1 Introduction

Prenons l'exemple de la fonction qui calcule la factorielle d'un nombre entier. Celle-ci s'écrit de la manière suivante en C :

```
int factorielle (int n)
{
  int i;
  int facto = 1;

  for (i = 2; i <= n; i ++)
    facto *= i;

  return facto;
}
```

On trouve :

- la ligne `int factorielle (int n)` qui indique (dans l'ordre) :
  - que la fonction renvoie une valeur de type `int` ; ce type est appelé le « type de la fonction » ;
  - que la fonction se nomme `factorielle` ;
  - qu'elle prend un paramètre de type `int`, de nom `n` ;
- entre `{ }`, le contenu de la fonction :
  - la déclaration de deux variables locales, `i` et `facto` ;
  - une boucle `for` qui effectue le calcul de la factorielle ;
  - une instruction `return` qui renvoie la valeur de la fonction. Cette instruction arrête brutalement l'exécution de la fonction en cours d'exécution.

On peut ensuite écrire un programme qui utilise la fonction `factorielle` comme suit :

```
#include <stdio.h>

/* inserer la fonction factorielle precedente */

main (void)
{
  int n, f;

  scanf ("%d", &n);
  f = factorielle (n);
  printf ("La factorielle de %d est %d\n", n, f);
}
```

Notons qu'en C :

- on ne peut pas déclarer une fonction à l'intérieur d'une autre fonction ;
- une fonction qui ne renvoie pas de valeur (qui se comporte donc comme une procédure) est déclarée de type `void` ;
- si on ne spécifie pas son type, une fonction est de type `int` par défaut.

5.2 Passage de paramètres

En C, le passage d'une valeur par paramètre repose sur une règle très simple : la valeur d'un paramètre peut être modifiée dans la fonction, mais cette valeur ne change pas du point de vue de la fonction qui l'appelle. On peut dire également qu'en C, lors d'un appel de fonction, la valeur des paramètres effectifs est recopiée et c'est sur cette copie que la fonction appelée travaille. Ainsi, si l'on écrit :

```
void f (int a, int b)
{
  a += b;
  b = 78;
  printf ("f: a = %d, b = %d\n", a, b);
}

main (void)
{
```

```
  int x, y;

  x = 10;
  y = 20;
  f (x , y);
  printf ("main: x = %d, y = %d\n", a, b);
}
```

on obtiendra l'affichage suivant :

```
f: a = 30, b = 78
main: x = 10, y = 20
```

La première ligne correspond à l'affichage provoqué par le `printf` de la fonction `f`, la seconde à l'affichage provoqué dans la fonction `main`. On constate que l'on a changé la valeur des paramètres `a` et `b` dans la fonction `f` et que cela n'a pas modifié la valeur de `x` et `y` dans la fonction `main`.

5.2.1 Modification d'un paramètre

Cette règle de passage des paramètres en C étant absolue, un moyen doit être trouvé pour pouvoir modifier la valeur d'un paramètre dans une fonction de telle manière que cette modification soit prise en compte dans la fonction appelante, au retour de l'appel de fonction. Pour cela, au lieu de passer la valeur du paramètre, on passe un pointeur sur cette valeur. Ce principe est illustré dans le programme suivant, qui ressemble beaucoup au précédent, mais en est fondamentalement différent :

```
void f (int *a, int *b)
{
  *a += *b;
  *b = 78;
  printf ("f: a = %d, b = %d\n", *a, *b);
}

main (void)
{
  int x, y;

  x = 10;
  y = 20;
```

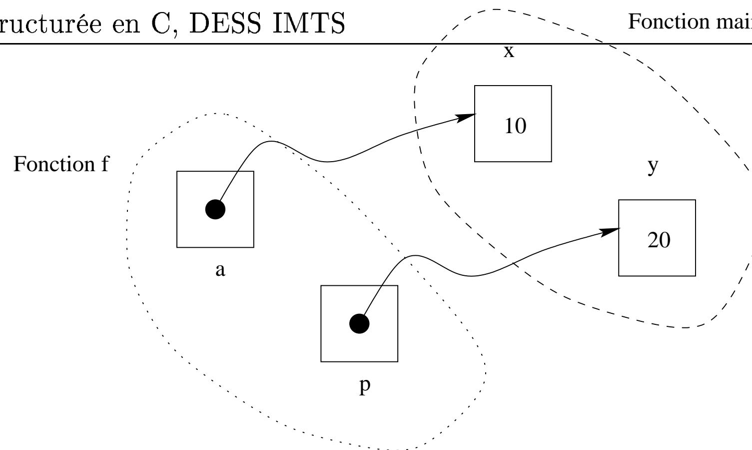


FIG. 5.1 – Illustration graphique du passage de paramètre par pointeur (voir le texte pour plus d'explications).

```
  f (&x , &y);
  printf ("main: x = %d, y = %d\n", a, b);
}
```

L'affichage est cette fois-ci le suivant :

```
f: a = 30, b = 78
main: x = 30, y = 78
```

Les deux affichages sont identiques : la fonction `f` a modifié la valeur des deux paramètres.

Le fonctionnement est le suivant (voir également la figure 5.1) :

- dans la fonction `main`, on appelle la fonction en passant des références sur les variables `x` et `y` au lieu de leur valeur, ceci grâce à l'opérateur `&` (voir le chapitre 1) ;
- dans la fonction `f`, les paramètres sont déclarés non pas de type `int`, mais de type pointeur sur un `int` (en utilisant une `*`) ; aussi, dans `f`, les variables `a` et `b` sont des références sur les valeurs des variables `x` et `y` ; les valeurs sont ensuite manipulées *via* ces références, donc en mettant une `*` devant le nom des paramètres ;
- au retour dans la fonction `main`, les valeurs de `x` ey `y` ont été modifiées *via* les références passées en paramètre à la fonction `f`.

5.3 Exercices

1. écrire une fonction qui prend deux paramètres de valeur `int` et échange leurs contenus ;
2. transformer les programmes écrits au chapitre 2 en fonctions.

Chapitre 6

Les tableaux

Si un tableau, comme dans les autres langages, est un ensemble de valeurs du même type, chacune étant repérée par son indice, il faut leur réserver un traitement particulier en C du fait qu'un tableau ressemble beaucoup à un pointeur.

Nous traiterons également des chaînes de caractères qui sont des tableaux de caractères.

Enfin, les tableaux multi-dimensionnels sont en C des tableaux de tableaux.

6.1 Les tableaux

6.1.1 Introduction

En C, un tableau est constitué d'un ensemble d'éléments d'un type donné ; ils sont toujours indicés à partir de 0 ; s'il contient `N` éléments, ceux-ci sont numérotés de 0 à `N-1`. Aucun contrôle n'est réalisé quant à la valeur des indices qui sont utilisés : l'utilisation d'un indice supérieur ou égal à `N` ne provoque aucun message clair ; éventuellement, le programme se plante, mystérieusement, de temps en temps.

Un tableau est déclaré comme suit :

```
int tableau_de_dix_entiers [10];
```

On peut ensuite accéder aux éléments du tableau individuellement par la notation `tableau_de_dix_entiers [2]`.

On peut initialiser la valeur de tous les éléments d'un tableau lors de sa déclaration comme suit :

```
int tableau [5] = { 23, 87, -54, 8, 1 };
```

Cette ligne déclare un tableau dénommé `tableau` de 5 entiers ; ceux-ci sont initialisés dans l'ordre à partir de l'élément d'indice 0 avec les 5 valeurs qui suivent entre `{}`. Il faut impérativement qu'il y ait autant de valeurs indiquées entre `{}` que d'éléments déclarés pour le tableau (ici 5).

On peut également omettre la taille dans la déclaration ; dans ce cas, le nombre de valeurs spécifiées pour l'initialisation indique la taille du tableau :

```
int tableau [] = { 23, -54, 8 };
```

`tableau` est de taille 3.

Il est très courant, et c'est une bonne habitude à prendre, de spécifier la taille d'un tableau non pas *via* une constante numérique, mais *via* une macro constante qui se définit comme suit :

```
#define TAILLE 10
```

Attention, le caractère `#` doit impérativement se trouver en début de ligne (1^{re}colonne).

Passage d'un tableau en paramètre

Pour déclarer un paramètre formel comme un tableau, on écrit :

```
f (int tab [])
```

ce qui signifie que le paramètre `tab` est un tableau d'entiers. Son nombre d'éléments n'est pas indiqué. Il est de bon ton de l'indiquer dans un autre paramètre :

```
f (int tab [], int longueurDeTab)
```

Comparaison et affectation de deux tableaux

La comparaison de deux tableaux ou l'affectation d'un tableau à un autre doivent être effectuées élément par élément dans une boucle. Par exemple, pour affecter un tableau à un autre, on écrira :

```
void affecteTableaux (int t1 [], int t2 [], int longueurDeTab)
{
  int i;

  for (i = 0; i < longueurDeTab; i ++)
    t1 [i] = t2 [i];
}
```

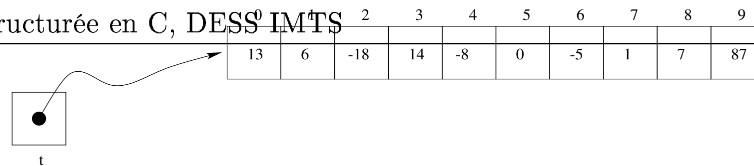


FIG. 6.1 – Lien entre tableau et pointeur (voir le texte pour les explications).

Remarquons que les éléments de `t1` sont modifiés au retour de la fonction, ce qui semble contredire ce qui a été dit à la section 5.2 du chapitre 3. Ce point très important sera discuté à la section 6.1.3.

Pour tester l'égalité de deux tableaux, on écrira une fonction du genre :

```
int egaliteTableaux (int t1 [], int t2 [], int longueurDeTab)
{
  int i;

  for (i = 0; i < longueurDeTab; i ++)
    if (t1 [i] != t2 [i])
      return 0;
  return 1;
}
```

en rappelant que 0 signifie « faux », 1 (ou toute autre valeur non nulle) signifie « vrai ».

6.1.2 Tableaux et pointeurs

En C, lorsque l'on déclare un tableau, on déclare implicitement un pointeur. Considérons la déclaration suivante :

```
int t [10];
```

`t` est un pointeur sur un `int` et il référence l'élément d'indice 0 du tableau (voir la figure 6.1). Dès lors, `t [0]` et `*t` référencent la même valeur (dans la figure 6.1, tous deux vallent 13).

L'addition est définie sur les pointeurs. Ainsi, `t+1` référence l'élément `t [1]`, `t+2` référence l'élément `t [2]`, et ainsi de suite jusque `t+9` qui référence l'élément `t [9]`.

Pour afficher le contenu du tableau `t`, on peut écrire de manière tout à fait classique la boucle suivante :

```
  int i;
  int t [10];

  for (i = 0; i <= 9; i ++)
    printf ("%d ", t [i]);
}
```

En C, on peut également écrire :

```
{
  int i;
  int t [10];

  for (i = 0; i <= 9; i ++) {
    printf ("%d ", *t);
    t ++;
  }
}
```

mais on aura alors perdu la valeur de `t`. Il faudrait plutôt utiliser un pointeur pour parcourir le tableau `t` :

```
{
  int i;
  int t [10];
  int *ptr;

  ptr = t;
  for (i = 0; i <= 9; i ++) {
    printf ("%d ", *ptr);
    ptr ++;
  }
}
```

que l'on peut encore écrire de manière plus compacte :

```
    int i;
    int t [10];
    int *ptr;

    ptr = t;
    for (i = 0; i <= 9; i ++, ptr ++)
      printf ("%d ", *ptr);
  }
```

6.1.3 Passage de tableau en paramètre

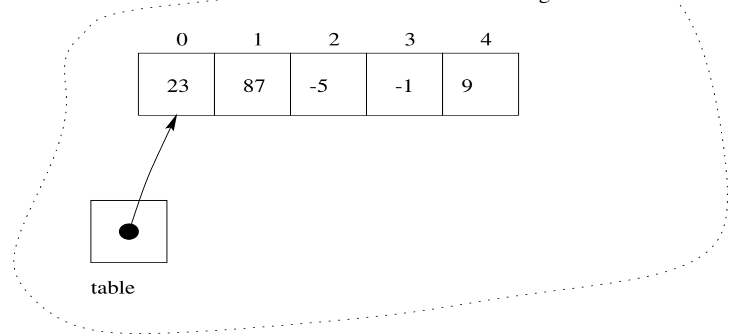
La conséquence immédiate de ce qui vient d'être dit sur les tableaux et les pointeurs est que lors du passage d'un tableau en paramètre d'une fonction, une référence sur le premier élément du tableau est passé, et non le tableau complet. Disposant d'une référence, la fonction peut librement modifier les éléments du tableau de manière visible pour la fonction l'ayant appelée (voir la figure 6.2).

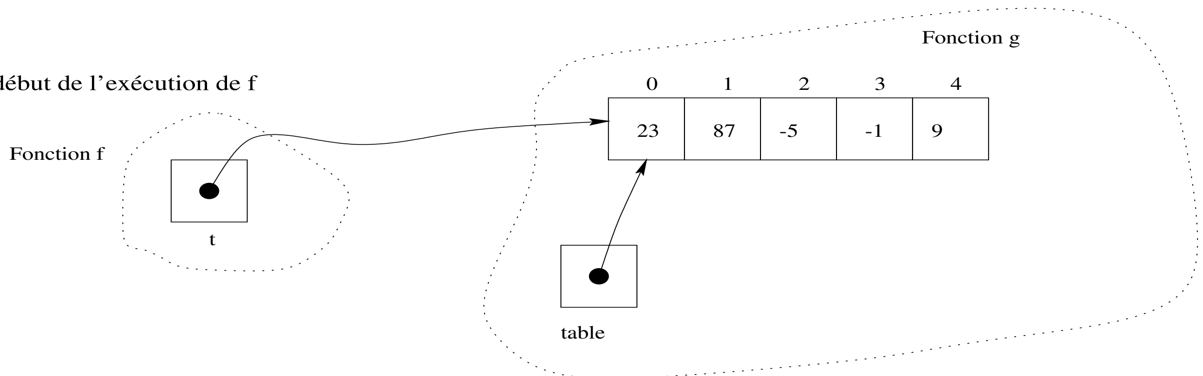
```
void f (int t [], int lg)
{
  int i;

  for (i = 0; i < lg; i ++)
    t [i] *= 2;
}

void g (void)
{
  int table [5] = { 23, 87, -5, -1, 9 };
  int i;

  f (table, 5);
  for (i = 0; i < lg; i ++)
    printf ("%d\n", t [i]);
}
```

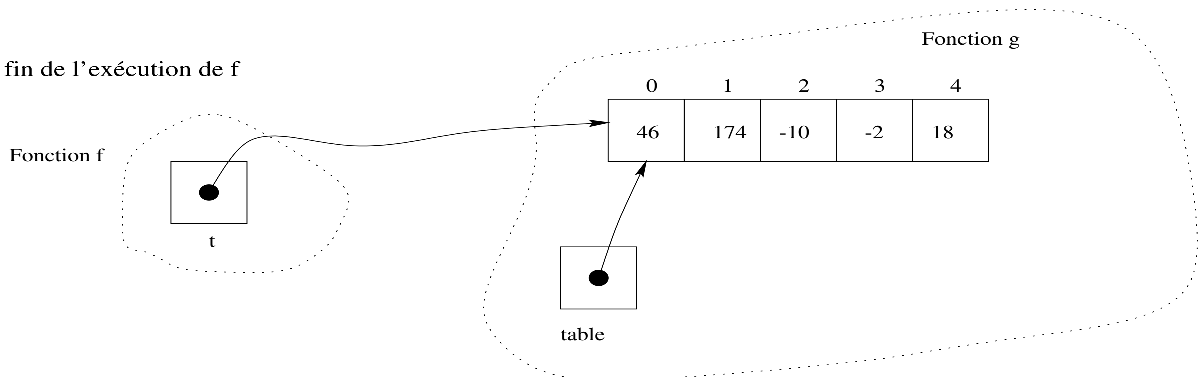
6.1.4 Les chaînes de caractères

Une chaîne de caractères est un tableau d'un certain nombre de caractères :

Avant l'appel de la fonction f







Au retour dans la fonction g

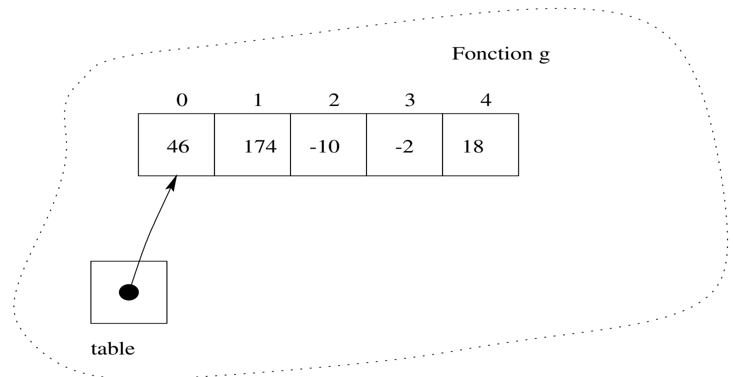


FIG. 6.2 – Passage d'un tableau en paramètre (voir le texte pour les explications).

```
char chaine_de_caracteres [10];
```

Une chaîne de caractères constante est délimitée par des guillemets `"`, telle `"une chaine de caracteres constante"`.

Par convention, un caractère de code nul (`'\000'` ou `(char) 0`) termine une chaîne de caractères constante. Aussi, la chaîne `"bonjour"` contient 8 caractères et non pas 7. Le fait de placer un caractère nul en fin de chaîne de caractères est une convention généralement utilisée en C ; ainsi, nul n'est besoin d'utiliser une variable auxiliaire spécifiant sa longueur ; la chaîne s'arrête lorsque le caractère nul est rencontré.

Un certain nombre de fonctions peuvent être utilisées sur des chaînes de caractères. Celles-ci doivent impérativement se terminer par un caractère nul. Pour cela, il faut mettre la ligne :

```
#include <string.h>
```

en début de fichier. Ces focntions sont :

- `int strlen (const char *s)` renvoie le nombre de caractère composant la chaîne de caractères `s`, sans compter le caractère nul. Par exemple, `strlen ("bonjour")` renvoie 7 ;
- `char *strdup (const char *s)` renvoie une copie de la chaîne de caractères `s` ;
- `int strcmp (const char *s1, const char *s2)` compare les deux chaînes de caractères `s1` et `s2` :
  - si `s1` est inférieure à `s2`, la valeur -1 est renvoyée ;
  - si `s1` est supérieure à `s2`, la valeur 1 est renvoyée ;
  - si `s1` est égale à `s2`, la valeur 0 est renvoyée.

  Par exemple, `strcmp ("bonjour", "hello")` renvoie -1 ;
- `int strncmp (const char *s1, const char *s2, int n)` compare les `n` premiers caractères des deux chaînes de caractères `s1` et `s2` :
  - si la chaîne composée des `n` premiers caractères de `s1` est inférieure à la chaîne composée des `n` premiers caractères de `s2`, la valeur -1 est renvoyée ;
  - si la chaîne composée des `n` premiers caractères de `s1` est supérieure à la chaîne composée des `n` premiers caractères de `s2`, la valeur 1 est renvoyée ;
  - si la chaîne composée des `n` premiers caractères de `s1` est égale à la chaîne composée des `n` premiers caractères de `s2`, la valeur 0 est renvoyée.

  Par exemple, `strcmp ("abc", "abcd", 4)` renvoie -1, `strcmp ("abc", "abcd", 3)` renvoie 0, `strcmp ("dbc", "abcd", 3)` renvoie 1 ;
- `char *strcpy (char *s1, const char *s2)` copie dans `s1` la chaîne `s2` (y compris le caractère nul) et renvoie un pointeur sur cette copie ;
- `char *strncpy (char *s1, const char *s2, int n)` copie dans `s1` les `n` premiers caractères de la chaîne `s2` (y compris le caractère nul) et renvoie un pointeur sur cette copie ;

- char *strcat (char *s1, const char *s2) ajoute les caractères de la chaîne s2 à ceux de s1 à partir et en écrasant le caractère nul de s1 ; strcat renvoie un pointeur sur cette nouvelle chaîne de caractères. Par exemple, strcat ("au", " revoir") renvoie la chaîne "au revoir" ;
- char *strncat (char *s1, const char *s2, int n) ajoute les n premiers caractères de la chaîne s2 à ceux de s1 à partir et en écrasant le caractère nul de s1 ; strncat renvoie un pointeur sur cette nouvelle chaîne de caractères. Par exemple, strncat ("au", " revoir, a bientot", 6) renvoie la chaîne "au revoir"

Nous mentionnons deux autres fonctions très utiles pour effectuer des conversions :

- long atol (const char *ch) qui convertit une chaîne de caractères en un nombre entier. Pour l'utiliser, il faut mettre la ligne #include <stdlib.h> en début de programme. Par exemple, atol ("325") renvoie l'entier 325 ;
- double atof (const char *ch) qui convertit une chaîne de caractères en un nombre flottant(double). Pour l'utiliser, il faut mettre la ligne #include <stdlib.h> en début de programme. Par exemple, atof ("3.25") renvoie le double 3.25.

Dans ces deux fonctions, il faut que la chaîne de caractères à convertir se termine par un caractère nul.

Enfin, la fonction sprintf est très pratique pour construire des chaînes de caractères assez complexes. Elle s'utilise exactement comme la fonction printf mis à part que le résultat, au lieu d'être affiché à l'écran, est stocké dans une variable passée en paramètre de sprintf :

```
int d = 7;
float x = 8.6;
char t [100];

sprintf (t, "La valeur de d est %d, celle de x est %f", d, x);
```

Le tableau t contient ensuite la chaîne de caractères "La valeur de d est 7, celle de x est 8.6".

La chaîne de caractères résultat est terminée par un caractère nul.

6.1.5 Les tableaux multi-dimensionnels

Généralités

On peut déclarer des tableaux de tableau, ce qui n'est pas tout à fait identique à une matrice. Ainsi, on peut déclarer :

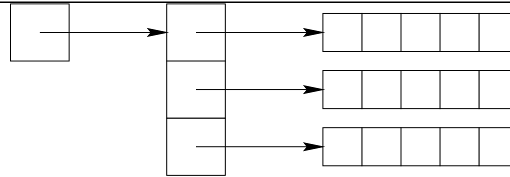


FIG. 6.3 – Représentation graphique d'un tableau à deux dimensions.

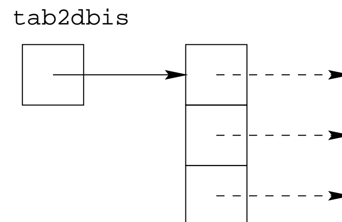


FIG. 6.4 – Représentation graphique d'un tableau à deux dimensions.

```
int tab2d [3][5];
```

qui est un tableau de 3 tableaux de 5 éléments entiers. On peut ensuite accéder à ses éléments par une notation du genre `tab2d [1][3] = 19;`.

`tab2d` est un pointeur sur un tableau d'entiers, ou encore, un pointeur de pointeur d'entiers. Les éléments `tab2d [0]`, `tab2d [1]` et `tab2d [2]` sont des tableaux de 5 entiers, donc des pointeurs sur entiers.

Lors de la déclaration, on peut omettre de spécifier le nombre d'éléments dans la dernière dimension. Ainsi :

```
int tab2dbis [3][];
```

déclare un tableau de 3 tableaux d'entiers. Ces 3 tableaux n'existent cependant pas (on verra comment les créer au chapitre suivant).

On peut également déclarer un simple pointeur de tableaux sans spécifier aucune dimension :

```
int *tab2dter [];
```

Ici, `tab2d` est seulement une variable apte à pointer sur un tableau de tableaux d'entiers.

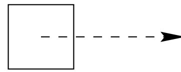


FIG. 6.5 – Représentation graphique d'un tableau à deux dimensions.

6.1.6 Les paramètres de la fonction `main`

La fonction `main` qui est la première fonction exécutée lors du lancement d'un programme écrit en C peut recevoir deux paramètres qui sont constitués des paramètres spécifiés sur la ligne de commande qui lance le programme. Ces paramètres se déclarent :

```
int main (int argc, char *argv[])
```

`argv` est un tableau de tableaux de caractères, soit un tableau de chaînes de caractères. Celui-ci contient `argc` chaînes. L'élément `argv [0]` contient toujours le nom du programme lui-même.

Ainsi, si on lance un programme dénommé `prgC` avec les paramètres : `prgC abc defgh i 67`, `argc` aura la valeur 4 et :

- `argv [0]` contient la chaîne de caractères `"prgC"` ;
- `argv [1]` contient la chaîne de caractères `"abc"` ;
- `argv [2]` contient la chaîne de caractères `"defgh"` ;
- `argv [3]` contient la chaîne de caractères `"i"` ;
- `argv [4]` contient la chaîne de caractères `"67"`.

6.2 Exercices

1. écrire une fonction qui prend 3 tableaux en paramètres `t`, `t1` et `t2`, ainsi que leur taille (les 3 tableaux sont de la même taille). La fonction initialise `t1` et `t2` avec, respectivement, les éléments pairs et impairs de `t`. Les éléments non utilisées dans `t1` et `t2` sont mises à -1 ;
2. écrire une fonction qui recherche le début de la plus longue monotonie dans un tableau d'entiers et renvoie l'indice de son 1^{er}élément ;
3. écrire une focntion qui calcule un carré magique d'ordre impair ; cet ordre est passé en paramètre à la fonction. Concevoir l'algorithme qui produit les carrés magiques d'ordre 3, 5 et 7 suivant :

4	9	2
3	5	7
8	1	6

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

22	47	16	41	10	35	4
5	23	48	17	42	11	29
30	6	24	49	18	36	12
13	31	7	25	43	19	37
38	14	32	1	26	44	20
21	39	8	33	2	27	45
46	15	40	9	34	3	28

4. écrire la fonction qui calcule le triangle de Pacal jusqu'à une certaine ligne passée en paramètre en utilisant un tableau mono-dimensionnel dans lequel on ne stocke qu'une seule ligne, la ligne en cours de calcul ;
5. écrire une fonction qui renvoie le miroir d'une chaîne de caractères. Par exemple, si on lui fournit la chaîne `abcdef`, la fonction renvoie `fedcba` ;
6. écrire une fonction qui prend en entrée une chaîne de caractères et détermine si c'est un palindrome, c'est-à-dire une chaîne qui peut se lire à l'identique dans les deux sens et renvoie un entier (booléen) ;
7. écrire une fonction qui prend deux chaînes de caractères en paramètre et renvoie le nombre de caractères communs aux deux chaînes ;
8. en utilisant un tableau de `BOOLEAN`, ećrire l'arbre programmatique de la procédure qui réalise le crible d'Ératosthène ;
9. on se donne un type `typedef enum {bleu, rouge, vert} Tricolore` et un tableau de `N` élément de ce type `typedef Tricolore Tabtricolore [N]`.

   Ce tableau est initialement rempli de manière aléatoire. Écrire une fonction tri le tableau de telle manière que tous les éléments de valeur `bleu` se trouvent aux indices 0 à `m`, les éléments de valeur `rouge` se trouvent aux indices `m+1` à `N-1`.

Chapitre 7

Les structures dynamiques

On aborde maintenant des notions du langage C liées aux structures dites « dynamiques », c'est-à-dire des structures qui sont créées ou dimensionnées lors de l'exécution du programme et non pas à la déclaration.

7.1 La fonction malloc et la fonction free

On peut créer des nouveaux objets, de n'importe quel type, avec la fonction malloc. Pour utiliser cette fonction, il faut mettre la ligne #include <stdlib.h> en début de programme. Ensuite, on l'utilise comme suit ;

```
int *p;

p = (int *) malloc (sizeof (int));
```

À l'issue de cette instruction, le pointeur p pointe vers un objet (il n'a pas de nom, il est anonyme) de type int. On peut ensuite accéder à cet objet *via* le pointeur : *p = 7;.

On dit que malloc « alloue » un objet, c'est-à-dire qu'il le crée.

Lorsque l'objet n'est plus nécessaire, on peut le libérer en utilisant la fonction free : free (p);.

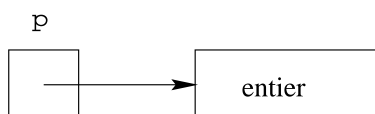


FIG. 7.1 – Un pointeur sur un int.

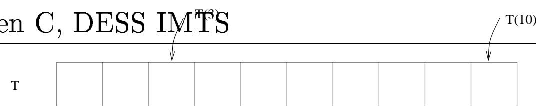


FIG. 7.2 – Un pointeur vers un tableau d'entiers.

7.2 Les tableaux dynamiques

Grâce à la fonction `malloc`, on peut dimensionner un tableau à volonté.

Un problème souvent rencontrer est de devoir utiliser un tableau dont on ne connaît le nombre d'éléments qu'à l'exécution du programme. Pour cela, on peut écrire :

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

main (void)
{
  int n;
  int *tableau;

  printf ("Entrer la taille du tableau :\n");
  scanf ("%d", &n);

  tableau = (int *) malloc (sizeof (int) * n);
}
```

L'appel de la fonction `malloc` est cette fois-ci légérement différent au cas précédent : pour indiquer que l'on veut créer un objet contenant `n` valeur de type `int`, le paramètre de `malloc` est `sizeof (int) * n`.

Ensuite, on peut utiliser `tableau` comme un tableau tout à fait normal, en écrivant par exemple `tableau [3] = 7;`.

Quand le tableau est créé ainsi au cours de l'exécution du programme, on dit qu'il est alloué « dynamiquement ».

Si pour une raison ou une autre la fonction `malloc` ne parvient pas à modifier la taille de l'objet, un pointeur nul est renvoyé. Ce pointeur est noté `NULL`. Il ne référence aucun objet.

7.2.1 La fonction `realloc`

Lorsqu'on a alloué un tableau dynamiquement et que celui-ci se révèle trop petit, on peut l'agrandir (ou le rapetisser) avec la fonction `realloc`. Elle s'utilise comme suit :

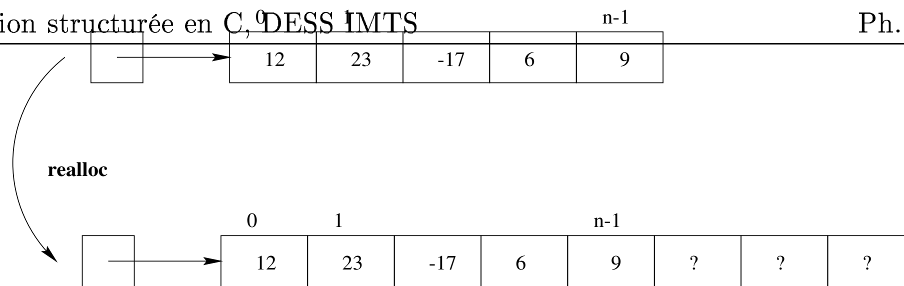


FIG. 7.3 – Effet d'un `realloc` sur un tableau d'entiers.

```
tableau = (int *) realloc (tableau, sizeof (int) * (2*n));
```

À l'issue de cette instruction, `tableau` pointe sur un tableau de `2*n` éléments de type `int`. Il faut noter que les `n` premiers éléments de `tableau` sont identiques avant et après l'appel à `realloc`.

Si pour une raison ou une autre la fonction `realloc` ne parvient pas à modifier la taille de l'objet, un pointeur nul est renvoyé.

7.3 Listes chaînées

Une structure de données très utilisée est la « liste chaînée ». Nous montrons comment une liste est créée sur un exemple.

Considérons une liste d'entiers. Il faut tout d'abord définir une structure qui représentera un chaînon, ou maillon, de cette liste :

```
typedef struct sle {
  int        valeur;
  struct sle *suiv;
} MaillonListeEntiers, *PListeEntiers;
```

Cela définit deux types ; le premier, `MaillonListeEntiers`, est un enregistrement composé de deux champs, l'un contient une valeur entière, l'autre référence un enregistrement de même type ; le second, `PListeEntiers`, est un pointeur sur un enregistrement de ce type.

L'objectif est de constituer une liste comme celle représentée à la figure 7.4.

Pour cela, il faut un objet qui référence le début (ou « tête ») de la liste. Déclarons-le comme suit : `PListeEntiers tete = NULL;`.

Il est important de l'initialiser à la valeur `NULL` pour indiquer qu'il ne référence pour l'instant aucun objet.

Pour créer le premier maillon de la liste, on pourra alors écrire :

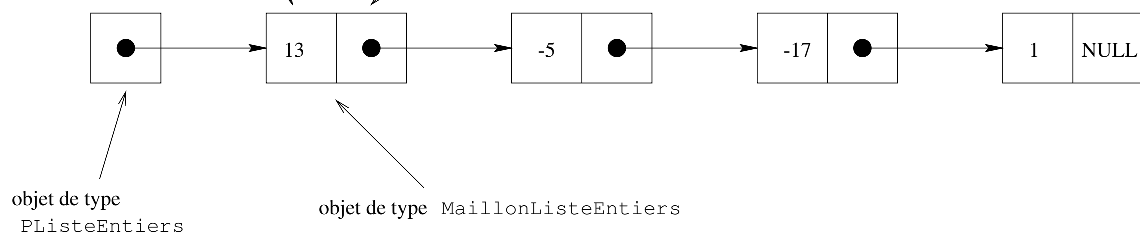


FIG. 7.4 – Représentation graphique d'une liste chaînée.

```
PListeEntiers p = (PListeEntiers) malloc (sizeof (MaillonListeEntiers));

p-> valeur = 1;
p-> suiv = tete;

tete = p;
```

La première instruction alloue un maillon. Les deux instructions suivantes affectent une valeur à chacun des champs. On note l'utilisation de `->` pour indiquer que l'on veut accéder au champ `valeur` *via* le pointeur `p`.

La deuxième instruction attache ce maillon à la liste déjà existante (pour l'instant elle est vide, mais si elle ne l'étai pas, on écrirait exactement la même instruction) en affectant la valeur du champ `suiv` avec la valeur de la variable `tete`.

Désormais, la nouvelle tête de liste est pointée par `p` ; il reste donc à mettre à jour la variable `tete` en lui affectant la valeur de `p`.

On peut ensuite ajouter les maillons suivantes :

```
/* ce qui precede */

/* on ajoute la valeur -17 */
p = (PListeEntiers) malloc (sizeof (MaillonListeEntiers));
p-> valeur = -17;
p-> suiv = tete;
tete = p;

/* on ajoute la valeur -5 */
p = (PListeEntiers) malloc (sizeof (MaillonListeEntiers));
```

```
p-> valeur = -5;
p-> suiv = tete;
tete = p;

/* on ajoute la valeur 13 */
p = (PListeEntiers) malloc (sizeof (MaillonListeEntiers));
p-> valeur = 13;
p-> suiv = tete;
tete = p;
```

À l'issue de cette séquence d'instructions, on a obtenu la liste de la figure 7.4.

On peut alors parcourir cette liste et afficher les valeurs qui s'y trouvent à l'aide de la boucle suivante :

```
for (p = tete ; p ; p = p-> suiv)
  printf ("%d\n", p-> valeur);
```

7.4 Exerccices

7.4.1 Liste chaînée

1. écrire une fonction qui recherche une valeur donnée dans une liste de type `PListeEntiers` et renvoie l avaleur 0 si elle est absente, une valeur non nulle si elle est présente ;
2. écrire la séquence d'instructions qui ajoute une valeur dans une `PListeEntiers` en faisant en sorte que celle-ci soit triée par ordre croissant. Ainsi, au lieu d'avoir les valeurs dans l'ordre précédent, elles seraient dans l'ordre suivant : -17, -5, 1, 13 ;
3. transformer cette séquence d'instructions en une fonction ;
4. écrire une fonction qui détruit une liste chaînée en appelant la fonction `free` sur chacun de ses maillons ;
5. écrire une fonction qui retire une valeur donnée d'une `PListeEntiers` ;

7.4.2 Liste bilatère

1. donner la déclaration en C d'un maillon de liste bilatère comme représentée à la figure 7.5 ;
2. donner la fonction qui ajoute une nouvelle valeur à une liste bilatère ;
3. donner une fonction qui indique si une valeur particulière est présente dans une liste bilatère ;
4. donner une fonction qui retire une valeur particulière d'une liste bilatère ;

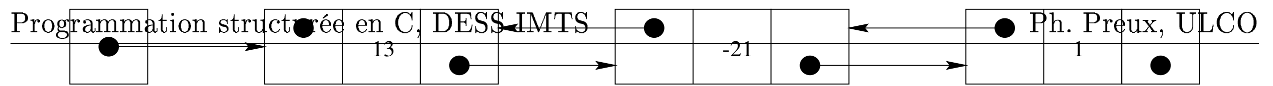


FIG. 7.5 – Représentation graphique d'une liste bilatère.

5. on veut maintenant que les valeurs soient triées en ordre croissant lors de leur insertion dans la liste ; ré-écrire toutes les fonctions précédentes dans ce cas ;

Chapitre 8

Les fichiers

D'une manière générale, un fichier C est un fichier de caractères. Les fonctions permettant d'utiliser les fichiers sont :

- créer un fichier ;
- ouvrir un fichier ;
- lire un caractère, ou une séquence de caractères, depuis un fichier ;
- écrire un caractère, ou une séquence de caractères, dans un fichier ;
- fermer un fichier.

Il existe deux types d'objets permettant la manipulation de fichiers : soit un fichier est référencé par un objet de type `FILE *`, soit il est référencé par un entier. La première solution est plus sophistiquée que la seconde. Dans un programme, il ne faut pas mélanger les deux.

Pour manipuler des fichiers, il faut mettre la ligne `#include <stdio.h>` en début de programme.

Nous ne décrivons que la première manière d'accéder aux fichiers, *via* les objets de type `FILE`.

8.1 Le type `FILE` et sa manipulation

8.1.1 Ouverture d'un fichier

Dans ce cas, on utilise un objet dénommé « descripteur de fichier » de type `FILE *` pour manipuler un fichier ; à chaque fichier est associé un descripteur de fichier. Sa valeur est obtenue à l'ouverture du fichier. Un fichier peut-être ouvert en lecture, en écritutre, ou en lecture-écriture. Supposons que nous ayons déclaré une variable `FILE *fd`, on ouvre un fichier comme suit :

- `fd = fopen ("nom-du-fichier", "r")` ouvre le fichier portant le nom `nom-du-fichier` en lecture. Le pointeur de flux est placé en début de fichier ;
- `fd = fopen ("nom-du-fichier", "r+")` ouvre le fichier portant le nom `nom-du-fichier` en lecture et écriture. Le pointeur de flux est placé en début de fichier ;
- `fd = fopen ("nom-du-fichier", "w")` ouvre le fichier portant le nom `nom-du-fichier` en écriture. Si le fichier n'existait pas, il est créé. Le fichier est vidé s'il contenait quelque chose.

Le pointeur de flux est placé en début de fichier ;

- `fd = fopen ("nom-du-fichier", "w+")` ouvre le fichier portant le nom `nom-du-fichier` en lecture et écriture. Si le fichier n'existait pas, il est créé. Le fichier est vidé s'il contenait quelque chose. Le pointeur de flux est placé en début de fichier ;
- `fd = fopen ("nom-du-fichier", "a")` ouvre le fichier portant le nom `nom-du-fichier` en écriture. Si le fichier n'existait pas, il est créé. Contrairement au cas précédent, le fichier n'est pas vidé. Le pointeur de flux est placé en début de fichier ;
- `fd = fopen ("nom-du-fichier", "a+")` ouvre le fichier portant le nom `nom-du-fichier` en lecture et écriture. Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé en début de fichier.

Si l'ouverture n'est pas possible, la fonction `fopen` renvoie le pointeur `NULL`. Dans ce cas, le descripteur de fichier n'est pas initialisé et ne peut pas être utilisé.

8.1.2 Accès en lecture

Pour lire des données dans un fichier ainsi ouvert, on utilise la fonction `fscanf` qui fonctionne exactement comme la fonction `scanf`, si ce n'est qu'elle prend un descripteur de fichier comme premier argument. On peut donc lire aisément la valeur d'un nombre dans un fichier, une chaîne de caractères, ... :

```
int i;
float x;
char s [100];

fscanf (fd, "%d%s%f", &i, &x, &x);
```

Lorsqu'on lit une chaîne de caractères, celle-ci s'arrête au premier caractère blanc.

8.1.3 Accès en écriture

Pour écrire des données dans un fichier ainsi ouvert, on utilise la fonction `fprintf` qui fonctionne exactement comme la fonction `printf`, si ce n'est qu'elle prend un descripteur de fichier comme premier argument. On peut donc écrire aisément la valeur d'un nombre dans un fichier, une chaîne de caractères, ... :

```
int i;
float x;
char s [100];

fprintf (fd, "%d%s%f", &i, &x, &x);
```

8.1.4 Fermeture d'un fichier

Quand on a terminé d'utiliser un fichier, on doit le fermer par l'appel de la fonction `fclose (fd)`.

8.1.5 Fichiers standards d'entrées-sorties

Sous Unix, trois fichiers sont particuliers :

- le fichier standard d'entrée qui est le fichier accédé par la fonction `scanf` ; celui-ci peut être accédé *via* son descripteur de fichier contenu dans la variable `stdin`. A priori, `stdin` est associé au clavier ;
- le fichier standard de sortie qui est le fichier accédé par la fonction `printf` ; celui-ci peut être accédé *via* son descripteur de fichier contenu dans la variable `stdout`. A priori, `stdout` est associé à l'écran ;
- le fichier standard d'erreur qui ne peut être accédé que *via* son descripteur de fichier contenu dans la variable `stderr`. A priori, `stdout` est associé à l'écran et ne doit être utilisé que pour afficher des messages d'erreur.

8.2 Exercices

1. écrire un programme qui copie un fichier dans un autre ;
2. écrire un programme qui compte le nombre de lignes, de mots et de caractères d'un fichier et affiche ces trois nombres ;
3. écrire un programme qui recherche une chaîne de caractères dans un fichier et affiche le nombre de lignes qui la contient.

Chapitre 9

Compléments

Ce chapitre est un fourre-tout contenant des informations utiles une fois que le reste du cours a été digéré.

9.1 Structures de contrôle

9.1.1 La boucle répéter

La boucle répéter a la forme suivante :

```
int compteur = 0;
do {
  compteur ++;
} while (compteur < 10);
```

Cette boucle va effectuer 10 itérations au cours desquelles la variable `compteur` va être incrémentée. D'une manière générale, le corps de la boucle est répété tant que la condition d'itération est vérifiée ; dès que celle-ci est fausse, l'exécution du programme se poursuit avec l'instruction suivant la boucle.

9.1.2 L'aiguillage

L'aiguillage est une structure de contrôle classique qui permet d'éviter d'imbriquer de nombreux tests. Elle a la forme suivante :

```
int valeur;

scanf ("%d", &valeur);
```

```
switch (valeur) {
  case 0:
    printf ("La valeur est nulle\n");
    break;
  case 1, 2, 3, 4, 5:
    printf ("La valeur est comprise entre 1 et 5\n");
    break;
  default:
    printf ("La valeur correspond a un cas qui n'est pas pris en compte\n");
    break;
}
```

On liste donc les cas qui nous intéressent dans des clauses `case` après quoi on indique le traitement à réaliser dans ce cas-là. L'instruction finale `break` est extrêmement importante. Elle déclenche la poursuite de l'exécution du programme à la suite de l'aiguillage. Si elle est omise (cela ne provoque pas une erreur de compilateur), plusieurs cas peuvent être exécutés, ce qui peut poser des problèmes.

9.1.3 Complément sur les boucles `for`

L'instruction `break`

Dans une boucle `for`, l'instruction `break` déclenche une sortie immédiate de la boucle. Par exemple, on peut écrire :

```
/* char c; int i; char *tableau; int lgTableau; */
for (i = 0; i < lgTableau; i ++) {
  if (tableau [i] == c)
    break;
  /* suite du corps de la boucle */
}
```

Cette boucle aura le comportement suivant : si le caractère `c` n'est pas trouvé dans le `tableau`, la boucle sera itérée jusqu'au dernier caractère du `tableau` ; au contraire, dès que le caractère `c` est rencontré, la boucle est immédiatement arrêtée et l'exécution du programme se poursuit avec l'instruction qui suit la boucle `for`.

L'instruction continue

Dans une boucle `for`, l'instruction `continue` déclenche un arrêt immédiat de l'itération courante et passage à l'itération suivante. Par exemple, on peut écrire :

```
/* char c; int i; char *tableau; int lgTableau; */
for (i = 0; i < lgTableau; i ++) {
  if (tableau [i] == c)
    continue;
  /* suite du corps de la boucle */
}
```

Cette fois-ci, lorsque le caractère `c` est trouvé dans le tableau, il y a déclenchement immédiat de l'itération suivante.

9.2 Visibilité extérieure des objets définis dans un fichier source C

Lorsque l'on fait de la compilation séparée, c'est-à-dire que le source d'un programme est composé de plusieurs fichiers, il est indispensable de pouvoir indiquer si une fonction ou une variable définie dans un fichier est visible et accessible depuis un autre fichier source. Pour cela, on utilise les mots-clés `static` et `extern` comme suit :

- si l'on veut que l'objet ne soit pas accessible depuis un autre fichier source, dans sa déclaration, on fait précéder son type par le mot-clé `static` ;
- si l'on veut que l'objet puisse être accessible depuis un autre fichier source, dans sa déclaration, on fait précéder son type par le mot-clé `extern`.

Par exemple,

```
int variable_visible_a_l_exterieur;
static int variable_locale_au_fichier_courant;

int fonction_visible (...) { ... }
static void fonction_invisible_de_l_exterieur (...) { ... }
```

Par défaut, les variables définies en dehors des fonctions et les fonctions elles-mêmes sont visibles à l'extérieur (`extern` par défaut).

9.3 Les variables rémanentes

On a parfois besoin d'une variable locale à une fonction qui garde sa valeur d'un appel sur l'autre de la fonction. Ainsi, supposons que nous voulions avoir une variable qui garde en mémoire le nombre d'appel à une fonction donnée. Si l'on écrit :

```
void f (void)
{
  int nombreDAppel;
}
```

la variable `nombreDAppel` est différente à chaque appel de `f`. Par contre, on peut écrire :

```
void f (void)
{
  static int nombreDAppel = 0;

  nombreDAppel ++;
}
```

qui va donner le résultat voulu. Le mot-clé `static`, utilisé ici à l'intérieur d'une fonction, indique que la variable `nombreDAppel` est rémanente, c'est-à-dire qu'elle conserve sa valeur d'un appel sur l'autre de la fonction. Il faut alors nécessairement l'initiliser dans sa déclaration. Ensuite, l'instruction d'incrémentation présente dans la fonction comptera les appels à la fonction. On peut ainsi écrire le programme complet :

```
#include <stdio.h>

void f (void)
{
  static int nombreDAppel = 0;

  nombreDAppel ++;
  printf ("nombreDAppel = %d\n", nombreDAppel);
}

main (void)
{
```

```
  int i;
  for (i = 0; i < 10; i ++)
    f ();
}
```

9.4 Les directives du pré-processeur ; la compilation conditionnelle

On peut vouloir que certaines parties d'un programme soient compilées seulement dans certaines conditions déterminées lors de la compilation et non pas à l'exécution. Par exemple, on peut vouloir compiler certaines instructions dans une version de mise au point d'un programme et que ces instructions ne soient plus exécutées dans la version définitive du programme, sans nécessairement vouloir les retirer du source, en prévision d'évolutions futures du programme. Pour cela, la solution est la compilation conditionnelle. Nous montrons cela sur un exemple :

```
#define DEBUG_MODE 1

void f (void)
{
#if DEBUG_MODE
  /* instructions a compiler lors de la mise au point */
  printf ("Programme en cours de mise au point\n");
  ....
#else
  /* instructions a ne compiler que si l'on n'est pas en cours de mise au point */
  ...
#endif
  /* instructions a toujours compiler */
}
```

Ainsi, en changeant simplement la valeur de `DEBUG_MODE` de 1 à 0, certaines instructions sont compilées ou pas.

À la place de `#if` on peut utiliser `#ifdef nom` qui est vrai si `nom` correspond à une constante définie par une instruction `#define`. On peut également utiliser `#ifndef nom` qui, au contraire, est vrai si `nom` n'a pas été défini.

`#if` peut être suivi d'une condition dont la valeur est calculée lors de la compilation. Par exemple :

```
#define LG 10
```

```
#if LG<45
  /* instructions si LG < 45 */
#else
  /* instruction si LG >= 45 */
#endif
```

Dans toutes ces constructions, la partie sinon est facultative.

Index

π, 35
*, 11
&, 11
*=, 23
++, 24
+=, 23
-, 24
-=, 23
/=, 23
==, 23, 31
=, 23
? :, 24
FILE, 61
M_PI, 35
NULL, 56
#define, 44
#else, 69
#ifdef, 69
#ifndef, 69
#if, 69
%=, 23
%, 23
&&, 34
acos, 34
argc, 52
argv, 52
asin, 34
atan, 34
atof, 50
atol, 50
break, 66
case, 65
char, 25
const, 26
continue, 66
cos, 34
default, 65
double, 34
do, 65
enum, 30
exp, 34
extern, 67
fclose, 63
float, 34
fopen, 61
for, 33
fprintf, 62
free, 55
fscanf, 62
int, 7, 10
- *, 9
- +, 8
- -, 8
- /, 9
- %, 9
- déclaration, 7
- opérateur, 8

log10, 34
log, 34
main, 29, 52
malloc, 55
printf, 27

`realloc`, 56
`scanf`, 28
`sin`, 34
`sprintf`, 50
`sqrt`, 34
`static`, 67, 68
`stderr`, 63
`stdin`, 63
`stdout`, 63
`strcat`, 50
`strcmp`, 49
`strcpy`, 49
`strdup`, 49
`string.h`, 49
`strlen`, 49
`strncat`, 50
`strncmp`, 49
`strncpy`, 49
`struct`, 30
`switch`, 65
`tan`, 34
`typedef`, 29
`while`, 32

adresse, 10
affectation, 8
  tableau, 44
aiguillage, 65
allocation, 55
  tableau, 56
analyse, 3
arbre programmatique, 14
arbre programmatique, 5
  boucle
    Pour, 18
    Tant-Que, 17
  séquence, 14
  test
    Si/Alors, 15
    Si/Alors/Sinon, 16

boucle
  indice, 33
  Pour, 18
  pour, 33
  répéter, 65
  Tant-Que, 17
  tant-que, 32

caractère
  nul, 47
chaîne de caractères
  tableau, 47
commentaire, 29
comparaison
  tableau, 44
compilation
  conditionnelle, 68
constante, 7, 44
  déclaration, 9, 26
conversion, 35
  réel en entier, 35

déclaration
  constante, 26
  tableau, 43
  type, 26, 29
  variable, 26
descripteur de fichier, 61
donnée, 3
données, 7

enregistrement, 30

fichier
  écriture, 62
  descripteur, 61
  fermeture, 63

lecture, 62
ouverture, 61
fonction, 37
main, 29
passage de paramètre, 38

identificateur, 7
indice de boucle, 18
indice de boucle, 33
initialisation, 9

libération, 55
liste chaînée, 57

opérateur
*=, 23
++, 24
+=, 23
-, 24
-=, 23
->, 58
., 30
/=, 23
==, 23
=, 23
? :, 24
%=, 23
%, 23
&&, 34
conditionnel, 24

paramètre
d'une fonction, 38
de main, 52
tableau, 44
passage de paramètre, 38
pointeur, 10
nul, 56
tableau, 45
priorité, 9

référence, 10

séquence, 14
structure, 30

tableau, 43
affectation, 44
chaîne de caractères, 47
comparaison, 44
déclaration, 43
de tableau, 50
dynamique, 56
paramètre, 44
pointeur, 45
test
Si/Alors, 15
Si/Alors/Sinon, 16
traitement, 3, 13
type
BOOLEAN, 7
int, 7
d'une fonction, 37
déclaration, 26

variable, 7, 10
déclaration, 26
rémanente, 67

Table des matières

1 Introduction **3**

2 Les données, les types, les entiers **7**

2.1 Le type `int` 7

2.1.1 Introduction 7

2.1.2 Opérations sur les valeurs de type `int` 8

2.2 Initialisation de variables lors de leur déclaration 9

2.3 Les constantes 9

2.4 Les pointeurs 9

2.5 Exercice 11

2.6 Ce qu'il faut retenir du chapitre 11

3 Les arbres programmatiques ; les tests et les boucles **13**

3.1 Les arbres programmatiques ; la séquence d'instructions 14

3.2 Les tests 15

3.2.1 Test Si/Alors 15

3.2.2 Test Si/Alors/Sinon 16

3.3 Les boucles 16

3.3.1 La boucle Tant-que 17

3.3.2 La boucle Pour 18

3.4 Exercices 20

4 Instructions C élémentaires et structures de contrôle **23**

4.1 Identificateurs 23

4.2 Instructions élémentaires du C 23

4.2.1 Quelques opérateurs particuliers en C 23

4.2.2 Le type `char` 25

4.2.3 Déclarations de variables et types 26

4.2.4 Déclarations de constantes 26

4.2.5 Les entrées-sorties en C . . . 27
4.3 Un exemple de programme C . . . 28
4.4 Déclaration de types . . . 29
4.4.1 Type énuméré . . . 30
4.4.2 Enregistrement . . . 30
4.5 Structures de contrôle . . . 31
4.5.1 Les tests . . . 31
4.5.2 Les boucles . . . 32
4.5.3 Conditions composées . . . 34
4.6 Les nombres réels . . . 34
4.6.1 Entrées-sorties de nombres réels . . . 34
4.6.2 Fonctions mathématiques . . . 34
4.6.3 Compatibilité et conversion entre entiers et réels . . . 35
4.6.4 Conversion . . . 35
4.7 Exercices . . . 36

5 Les fonctions et le passage de paramètres 37
5.1 Introduction . . . 37
5.2 Passage de paramètres . . . 38
5.2.1 Modification d'un paramètre . . . 39
5.3 Exercices . . . 40

6 Les tableaux 41
6.1 Les tableaux . . . 41
6.1.1 Introduction . . . 41
6.1.2 Tableaux et pointeurs . . . 43
6.1.3 Passage de tableau en paramètre . . . 45
6.1.4 Les chaînes de caractères . . . 45
6.1.5 Les tableaux multi-dimensionnels . . . 48
6.1.6 Les paramètres de la fonction `main` . . . 50
6.2 Exercices . . . 50

7 Les structures dynamiques 53
7.1 La fonction `malloc` et la fonction `free` . . . 53
7.2 Les tableaux dynamiques . . . 54
7.2.1 La fonction `realloc` . . . 54
7.3 Listes chaînées . . . 55
7.4 Exerccices . . . 57
7.4.1 Liste chaînée . . . 57

7.4.2 Liste bilatère 57

8 **Les fichiers** **59**

8.1 Le type `FILE` et sa manipulation 59

8.1.1 Ouverture d'un fichier 59

8.1.2 Accès en lecture 60

8.1.3 Accès en écriture 60

8.1.4 Fermeture d'un fichier 61

8.1.5 Fichiers standards d'entrées-sorties 61

8.2 Exercices 61

9 **Compléments** **63**

9.1 Structures de contrôle 63

9.1.1 La boucle répéter 63

9.1.2 L'aiguillage 63

9.1.3 Complément sur les boucles `for` 64

9.2 Visibilité extérieure des objets définis dans un fichier source C 65

9.3 Les variables rémanentes 66

9.4 Les directives du pré-processeur ; la compilation conditionnelle 67