C,
un premier
langage de
programmation

Jacques Le Maitre

UNIVERSITÉ du SUD
Toulon-Var

Table des matières

Avant-propos

Ce livre est le fruit de plusieurs années d'enseignement de la programmation en 1ère année de la licence d'informatique et de la licence de mathématiques à l'Université du Sud Toulon-Var. Il est, comme son titre l'indique, destiné à des débutants en programmation.

Une question s'était posée lors de la mise en place de cet enseignement : quel langage choisir ? Il y avait principalement trois candidats : Caml, Pascal ou C. Caml a pour lui son interactivité, la rigueur de son typage et la facilité avec laquelle des données structurellement complexes peuvent être construites et manipulées. Mais l'expérience montre que les débutants sont souvent rebutés par le côté abstrait de la programmation fonctionnelle. Pascal a pour lui sa syntaxe claire et rigoureuse, qui en fait un très bon langage sur le plan pédagogique, mais il est très peu utilisé dans les applications professionnelles contrairement au langage C, dont il est proche, et que nous avons finalement retenu, pour les raisons suivantes :

- c'est un des langages de programmation les plus utilisés au monde et ce pour des applications très diverses ;
- c'est un langage simple, ayant un nombre réduit de constructions, ce qui le rend relativement facile à apprendre et a permis le développement de compilateurs performants ;
- c'est un langage qui a servi de base à d'autres langages tels que C++ ou Java, eux-mêmes très utilisés ;
- c'est un langage qui s'intègre parfaitement à l'environnement UNIX, puisqu'il a été initialement conçu comme langage de programmation de ce système.

Plusieurs critiques, justifiées, sont émises à l'encontre du langage C. Il ne permet pas la manipulation globale des tableaux ou des chaînes de caractères. Il demande au programmeur une certaine agilité à manipuler les adresses des données : les fameux pointeurs, ce qui n'est pas toujours évident pour les débutants. L'utilisation de certains opérateurs peut conduire à des programmes peu lisibles. Cependant ces défauts peuvent être palliés par une écriture rigoureuse des programmes et par l'utilisation d'options de compilation fournissant un maximum d'avertissements sur les incorrections rencontrées dans le texte d'un programme.

Le C qui est décrit dans ce livre est conforme à la norme ANSI dite C89.

Au-delà de l'apprentissage de la programmation en C, c'est l'apprentissage de la programmation impérative et de la programmation tout court que vise ce livre. Le lecteur ayant bien assimilé les connaissances qui y sont délivrées devrait être bien armé pour apprendre d'autres langages.

Ce livre contient un très grand nombre d'exemples de programme et d'exercices. Il est fortement conseillé de le lire à côté de son ordinateur pour expérimenter ces exemples et faire ces exercices : c'est en programmant que l'on devient programmeur !

Pour une toute première initiation à la programmation en C, on pourra se restreindre au parcours suivant du contenu de ce livre : anatomie, compilation et exécution d'un programme C simple (chapitre 1, paragraphe 1.3 et 1.5) ; données, expressions et instructions (chapitres 2, 3 et 4) ; fonctions (chapitre 5, paragraphes 5.1 et 5.3) ; saisie et affichage de données (chapitre

6, paragraphe 6.1) ; tableaux monodimensionnels et structures (chapitre 7, paragraphes 7.1, 7.2 et 7.5).

Il faut aussi dire ce que n'est pas ce livre. Ce n'est pas un manuel de référence de C. Certains aspects de ce langage n'y sont pas traités : opérateurs de manipulation de bits, lectures et écriture de fichiers binaires, traitement des exceptions (beaucoup plus difficile en C que dans des langages plus évolués tels que Caml ou Java), etc. Ce n'est pas non plus un livre d'algorithmique. Bien qu'il soit courant, en première année de licence, de mêler dans un même enseignement l'apprentissage de la programmation et celui de l'algorithmique, mon avis est que ces deux enseignements peuvent être dissociés. La maîtrise d'un langage de programmation est indispensable pour poursuivre des études en informatique. Consacrer un cours exclusif à l'apprentissage de la programmation n'est donc pas superflu.

Pour terminer, je voudrais citer trois livres sur lesquels je me suis fortement appuyé. Tout d'abord deux livres sur C :

- *The C Programming Language*, 2nd edition, de Brian W. Kernighan et Dennis M. Ritchie (Prentice Hall), traduit en français : *Le langage C, Norme ANSI* (Dunod).
- *C : Langage, bibliothèque, applications* (InterEditions) d'Henri Garetta.

Le premier, incontournable, est celui des inventeurs de C. Le second, est issu d'une longue pratique de l'enseignement du C. Ses explications sont claires et précises et ses conseils toujours très judicieux. Ces deux livres, au dire même de leurs auteurs, s'adressent à des lecteurs ayant déjà une première expérience de la programmation. Une fois ses premiers pas en C accomplis, le programmeur pourra s'y plonger avec profit. Le troisième livre n'est pas un livre sur C, mais sur Caml, langage évoqué ci-dessus :

- *Le langage Caml* (Dunod) de Pierre Weiss et Xavier Leroy.

Je le cite ici, car c'est un excellent livre d'apprentissage de la programmation qui s'appuie sur des exemples particulièrement bien choisis et de difficulté croissante. Sa lecture ne peut que donner envie d'apprendre la programmation et au-delà : l'informatique.

Jacques Le Maitre
Professeur à l'Université du Sud Toulon-Var

1
Introduction

Dans ce chapitre introductif, nous brosserons tout d'abord (paragraphe 1.1) un panorama des langages de programmation disponibles actuellement. Nous donnerons une classification de ces langages ainsi qu'une chronologie des plus marquants d'entre eux, en y incluant le langage C.

Un programme est destiné à être exécuté par un ordinateur. Nous expliquerons (paragraphe 1.2) comment un programme écrit dans un langage de programmation est traduit, on dit « compilé », en langage machine c.-à-d. en une suite d'instructions exécutables par cet ordinateur. Nous montrerons que cette compilation s'appuie sur la définition de ce langage : son vocabulaire, sa grammaire et sa sémantique et sur l'architecture de cet ordinateur.

Avant d'en étudier séparément chaque constituant, il est nécessaire d'avoir une vision d'ensemble d'un programme C. Nous présenterons donc (paragraphe 1.3) un exemple de programme C, très simple, et le disséquerons.

Un programme est un objet qui évolue dans le temps. Entre sa conception et son abandon. il est, en général, exécuté et modifié de nombreuses fois. Nous dirons quelques mots de ce cycle de vie (paragraphe 1.4).

Nous terminerons ce chapitre en expliquant (paragraphe 1.5) la marche à suivre pour saisir, compiler et exécuter un programme C simple ce qui permettra au lecteur de faire les exercices proposés dans les premiers chapitres en attendant d'apprendre à organiser, à assembler et à exécuter un programme C dans toute sa généralité.

1.1 Langages de programmation : un panorama

Le rôle d'un langage de programmation est de décrire des tâches à soumettre à un ordinateur. Il existe à l'heure actuelle plusieurs centaines de langages de programmation. Ils se distinguent par leur niveau, par leur degré de généralité ou de spécialisation et par leur paradigme de programmation.

Un langage de programmation est dit de bas niveau s'il est proche du langage machine de l'ordinateur sur lequel il est exécuté. C'est le cas par exemple, des langages d'assemblage. Inversement un langage de programmation est dit de haut niveau s'il n'est pas lié à un ordinateur particulier et s'il permet au programmeur de décrire de façon aussi concise que possible ce que doit faire une tâche sans décrire en détail comment elle doit être exécutée. Les langages d'interrogation de bases de données, par exemple, sont des langages de haut niveau : l'utilisateur pose une question et le système de gestion de bases de données établit la stratégie la plus efficace pour y répondre.

Un langage de programmation généraliste est conçu pour programmer tout type d'application. Le langage C, objet de cet ouvrage, est un langage généraliste. Un langage de programmation spécialisé est conçu, au contraire, pour une classe d'application particulière. Par exemple, un langage pour la création de sites web. Les langages spécialisés sont souvent des langages de plus haut niveau que les langages généralistes.

On distingue quatre grands paradigmes de programmation : la programmation impérative, la programmation fonctionnelle, la programmation logique et la programmation orientée objet. Dans sa version la plus simple, un programme impératif se présente comme une suite d'instructions adressées à l'ordinateur et dont chacune a pour effet de changer l'état de la mémoire ou de communiquer avec les unités périphériques de l'ordinateur (clavier, écran ou imprimante, par exemple). La mémoire d'un programme impératif contient un ensemble de variables. Une variable a un nom et une valeur qui est rangée dans une case de la mémoire associée à cette variable. Cette valeur peut changer au cours de l'exécution du programme. Voici un exemple de programme impératif :

```
(1) var x, y, z: entier;
(2) x := 2;
(3) y := 5;
(4) z := x + y;
(5) afficher z.
```

(Notons que le texte de ce programme a été écrit en fonte non proportionnelle. C'est une convention classique que nous adopterons dans ce cours.) La ligne 1 contient la définition de trois variables : `x`, `y` et `z` dont les valeurs sont des nombres entiers. Les lignes 2, 3 et 4 contiennent chacune une instruction d'affectation qui demande l'enregistrement de la valeur de l'expression qui suit le symbole := dans la case mémoire associée à la variable dont le nom précède ce symbole. La ligne 5 contient une instruction qui demande l'affichage à l'écran de la valeur de l'expression qui suit le mot-clé `afficher`. L'exécution d'un programme impératif consiste à exécuter les instructions les unes après les autres, ce qui dans le cas du programme ci-dessous produira l'affichage à l'écran du nombre 7.

Un programme fonctionnel est une suite de définitions de fonctions. Par exemple :

```
(1) pi = 3.14;
(2) carré(x) = x * x;
(3) aire_disque(r) = pi * carré(r);
```

La ligne 1 définit `pi` comme étant la fonction constante 3,14. La ligne 2 définit `carré` comme étant la fonction qui appliquée à une valeur `x` retourne cette valeur multipliée par elle-même. La ligne 3 définit `aire_disque` comme étant la fonction qui appliquée à une valeur `r` retourne la valeur de la multiplication de `pi` par le carré de la valeur de `r`. L'exécution d'un programme fonctionnel consiste à poser une question sous la forme d'une application de fonction. La réponse est la valeur retournée par cette application. Par exemple, pour connaître l'aire d'un disque de rayon 5, l'utilisateur posera la question :

```
aire_disque(5);
```

La réponse sera 78,5 car :

```
aire_disque(5) = pi * carré(5) = 3.14 * 5 * 5 = 3.14 * 25 = 78.5
```

Un programme logique est constitué d'un ensemble de relations décrivant des faits et des règles. Par exemple :

```
(1) père("Alain", "Jean");
(2) mère("Nicole", "Jean");
(3) père("Paul", "Marie");
(4) mère("Claire", "Marie");
(5) père("Jean", "François");
(6) mère("Marie", "François");
```

```
(7) parent(x, y) si père(x, y);
(8) parent(x, y) si mère(x, y);
(9) grand-parent(x, z) si parent(x, y) et parent(y, z);
```

Les lignes 1 à 6 contiennent des relations qui sont des faits : Alain est le père de Jean, Nicole est la mère de Jean, etc. La ligne 7 contient une relation qui est une règle stipulant que *x* est un parent de *y* si *x* est le père de *y*. La ligne 8 contient une relation qui est une règle stipulant que *x* est un parent de *y* si *x* est la mère de *y*. La ligne 9 contient une relation qui est une règle stipulant que *x* est un grand-parent de *z* s'il existe un *y* tel que *x* est le parent de *y* et *y* est le parent de *z*. L'exécution d'un programme logique est lancée en posant une question sous la forme d'une relation à vérifier. Par exemple, pour connaître les grands-parents de François, l'utilisateur posera la question :

```
grand-parent(x, "François");
```

qui signifie : « Quelles sont les valeurs de `x` pour lesquelles la relation `grand-parent(x, "François")` est vraie ? ». La réponse sera :

```
{x = "Alain", x = "Nicole", x = "Paul", x = "Claire"}.
```

Un programme orienté objet est caractérisé par la façon dont sont représentées les données : un ensemble de classes dont les instances sont des objets. Une classe définit les propriétés des objets de cette classe et les opérations que l'on peut leur appliquer. Par exemple, la classe `Personne` possédant les propriétés `nom` et `année_de_naissance` et l'opération `âge` définie par `âge(p) = année courante – année_de_naissance(p)` où `p` est un objet de la classe `Personne`. Un objet est défini par un identificateur unique et invariable et par les valeurs de ses propriétés qui, elles, sont variables. Une classe peut être fille d'une autre classe. En ce cas, les objets de la classe fille « héritent » de toutes les propriétés et de toutes les opérations de la classe mère. Par exemple, si la classe `Etudiant` est une sous-classe de la classe `Personne` alors, un étudiant est aussi une personne qui a un nom et une année de naissance et dont l'âge peut être calculé par l'opération `âge` définie dans la classe `Personne`. Un étudiant pourra avoir par ailleurs des propriétés supplémentaires telle que `numéro_étudiant`. La manipulation des données peut, elle, être réalisée selon le paradigme impératif ou fonctionnel.

Les langages de programmation mêlent en général plusieurs de ces quatre paradigmes. La possibilité de définir et d'appliquer des fonctions, notamment, est présente dans pratiquement tous les langages de programmation.

Les premiers langages de programmation datent du début des années 1950, et comme nous l'avons dit ci-dessus, des centaines ont vu le jour depuis. Nous citerons les plus marquants des langages généralistes. Le premier est Fortran (Formula translator) apparu en 1954 principalement orienté vers les applications numériques. Quelques années plus tard, en 1959, est apparu le langage Cobol (Common business oriented language) orienté vers un autre grand domaine d'application de l'informatique : la gestion. Le premier langage de programmation fonctionnel a été créé en 1960 : c'est Lisp (List processing), un langage de haut niveau, très adapté à l'intelligence artificielle. Le langage Algol (Algorithmic language) dont la première version date de 1960 et la deuxième de 1968 doit, bien que n'ayant pas dépassé le cadre universitaire, son importance à l'introduction de concepts qui ont ensuite été repris dans la plupart des langages de programmation modernes : structures de bloc, programmation structurée, récursivité, etc. L'avènement, au début des années 1970, des micro-ordinateurs a popularisé deux langages à cause de leur simplicité : le langage Basic (Beginner's all-purpose symbolic instruction code) créé en 1960 et le langage Pascal (du nom du mathématicien Blaise Pascal) dérivé d'Algol, créé en 1967, qui a connu un succès immédiat à cause de la clarté de sa syntaxe, de sa portabilité et d'un très bon environnement de programmation. En

```
(1) var longueur, largeur, perimetre;
(2) saisir longueur;
(3) saisir largeur;
(4) perimetre := 2 * (longueur + largeur);
(5) afficher perimetre.
```

Figure 1.1. *Un programme J : le programme Périmètre*

1970 a été créé le premier langage de programmation logique, Prolog, qui a ensuite été étendu à la programmation par contraintes. Comme Lisp, il est principalement utilisé dans le domaine de l'intelligence artificielle.

Le langage C a été créé au début des années 1970 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. Il a de plus servi de base au langage orienté objet C++ et au langage Java très utilisé dans les applications web.

1.2 Compilation d'un programme

Pour bien comprendre un langage de programmation, il est nécessaire d'avoir un minimum de connaissances sur la façon dont un programme est compilé puis exécuté par un ordinateur. Nous allons l'expliquer sur un exemple très simple, consistant à compiler un programme écrit dans un langage de programmation « jouet », inventé pour la circonstance, que nous appellerons le « langage J », et destiné à être exécuté sur un processeur très simplifié que nous appellerons la « pitchoun-machine ».

1.2.1 **Le langage J**

Le langage J permet d'écrire des programmes qui manipulent des nombres entiers relatifs en les additionnant, les soustrayant, les multipliant, les divisant (division entière), les affectant à des variables, les saisissant au clavier et les affichant à l'écran. La figure 1.1 montre un programme J qui calcule, affecte à la variable `perimetre`, puis affiche le périmètre d'un rectangle dont la longueur et la largeur sont saisies au clavier et affectées respectivement aux variables `longueur` et `largeur`.

Comme tout langage, un langage de programmation est défini par son vocabulaire : les mots du langage, sa syntaxe : les règles de grammaire à appliquer pour rédiger le texte d'un programme à partir de ces mots et sa sémantique : les règles qui donnent une signification à ce texte.

Vocabulaire du langage J

Un mot du langage J peut être :

- un mot-clé (`var saisir afficher`) ;
- un nom de variable : une suite de lettres ou de chiffres commençant par une lettre et différente d'un mot-clé ;
- une constante littérale entière : une suite de chiffres représentant un nombre entier dans l'intervalle [0, 32 767] (par exemple : `0` ou `18427`) ;
- le symbole d'affectation (`:=`) ;
- un opérateur arithmétique (`+ - * / %`) ;

programme → *déclaration liste-instructions* '.'
déclaration → 'var' *liste-noms* ';'
liste-noms → *liste-noms* ',' *nom*
liste-noms → *nom*
liste-instructions → *liste-instructions* ';' *instruction*
liste-instructions → *instruction*
instruction → *nom* ':=' *expression*
instruction → 'saisir' *nom*
instruction → 'afficher' *nom*
expression → *expression opérateur expression*
expression → '(' *expression* ')'
expression → -*expression*
expression → *terme*
opérateur → '+'
opérateur → '-'
opérateur → '*'
opérateur → '/'
opérateur → '%'
terme → *nom*
terme → *cte*

Figure 1.2. *Grammaire du langage J*

- un séparateur (, ;) ;
- le point de fin de programme (.).

Syntaxe du langage J

Pour décrire la syntaxe d'un langage de programmation, on utilise une forme particulière de grammaire : les grammaires BNF[1]. Une grammaire BNF est constituée d'un ensemble de règles de la forme : $x \rightarrow y_1\ y_2\ \ldots\ y_n$ où x est un symbole non terminal représentant un constituant générique d'un programme écrit dans ce langage et chaque y_i est soit un symbole non terminal, soit un symbole terminal (placé entre apostrophes ou souligné) désignant respectivement un mot ou une catégorie de mots de ce langage. Une telle règle se lit : « un constituant x est constitué d'un constituant y_1, suivi d'un constituant y_2, …, suivi d'un constituant y_n ». Une règle peut être récursive : le symbole de la partie gauche peut apparaître dans la partie droite. Lorsqu'un constituant peut être construit de plusieurs façons, on énonce autant de règles qu'il y a de façons de le construire. Parmi les symboles non terminaux, il en existe un, appelé symbole de départ qui désigne le constituant le plus général.

La grammaire du langage J est présentée sur la figure 1.2. Son symbole de départ est le symbole *programme*. Cette grammaire spécifie :

- qu'un programme J (symbole non terminal *programme*) est constitué d'une déclaration de variables (symbole non terminal *déclaration*) suivie d'une liste d'instructions (symbole non terminal *liste-instructions*), suivi d'un point (symbole terminal '.'),

[1] « Backus Naur Form », proposée par deux informaticiens qui ont grandement contribué à la formalisation des langages de programmation : John Backus et Peter Naur.

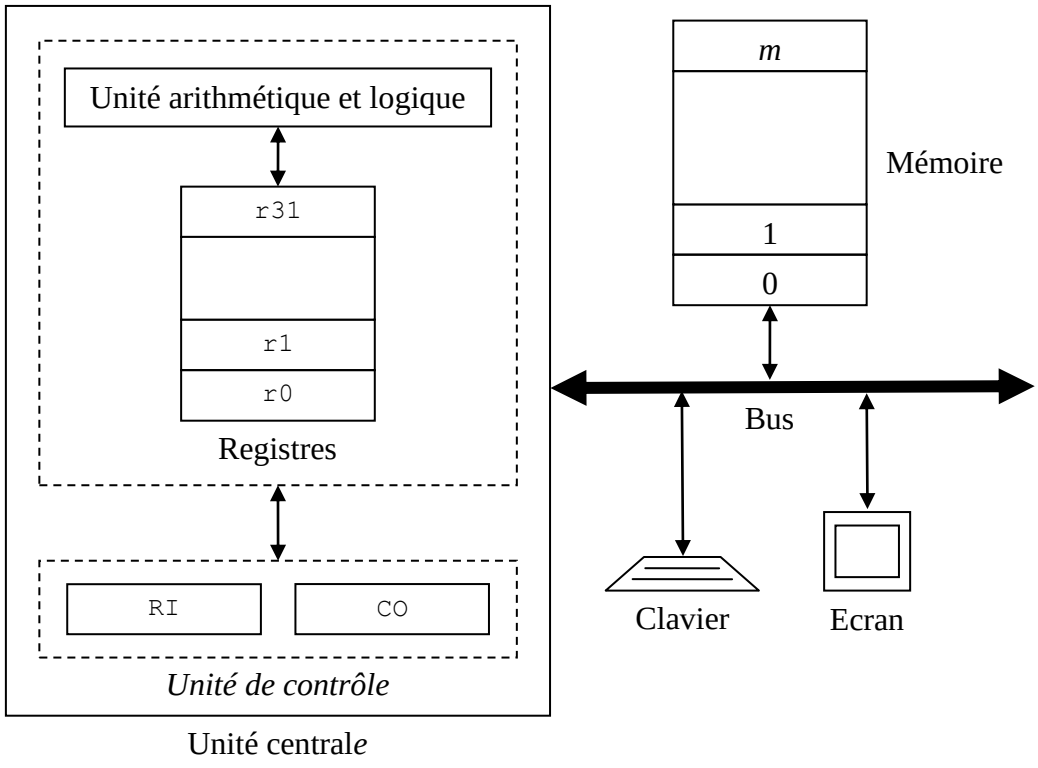


Figure 1.3. *Architecture de la pitchoun-machine*

- qu'une déclaration de variables est constituée du mot-clé « var » (symbole terminal '`var`') suivi d'une liste de noms de variables (symbole non terminal *liste-noms*) suivi d'un point-virgule (symbole terminal '`;`') ;
- qu'une liste de noms de variables est soit une liste de noms de variables suivi d'une virgule (symbole terminal '`,`') suivi d'un nom de variable (symbole terminal ***nom***), soit un nom de variable ;
- qu'une liste d'instructions est soit une liste d'instructions suivie d'un point-virgule suivi d'une instruction (symbole non terminal *instruction*), soit une instruction ;
- qu'une instruction est constituée :
  - soit d'un nom de variable suivi du signe d'affectation (symbole terminal '`:=`'), suivi d'une expression (symbole non terminal *expression*) ;
  - soit du mot-clé « saisir » (symbole terminal '`saisir`') suivi d'un nom de variable ;
  - soit du mot-clé « afficher » (symbole terminal '`afficher`') suivi d'un nom de variable ;
- qu'une expression est constituée :
  - soit d'une constante littérale entière (symbole terminal ***cte***) ;
  - soit d'un nom de variable ;
  - soit d'une expression suivie d'un opérateur (symbole non terminal *opérateur*) suivi d'une expression ;
  - soit d'une parenthèse ouvrante (symbole terminal '`(`'), suivie d'une expression, suivie d'une parenthèse fermante (symbole terminal '`)`') ;
- qu'un opérateur est soit le signe d'addition (symbole terminal '`+`'), soit le signe de soustraction (symbole terminal '`-`'), soit le signe de multiplication (symbole terminal '`*`'),

soit le signe de division (symbole terminal '`/`'), soit le signe de calcul du reste d'une division entière (symbole terminal '`%`').

Il faut de plus spécifier la priorité des opérateurs et leur associativité qui ne le sont pas dans cette grammaire :

- *priorité*(- unaire) > *priorité*(`%`) = *priorité*(`/`) = *priorité*(`*`) > *priorité*(`+`) = *priorité*(- binaire) ;
- les opérateurs de même priorité sont associatifs de gauche à droite.

Selon ces règles l'on aura par exemple :

```
-3 + 3 * 4 / 2 ≡ ((-3) + ((3 * 4) / 2))
```

Sémantique du langage J

La sémantique du langage J est définie par un ensemble de règles qui définissent la valeur d'une expression et les actions réalisées par les instructions.

La valeur d'une expression est le nombre entier obtenu en appliquant les règles suivantes :

- si *c* est une constante littérale entière alors *c* est une expression dont la valeur est le nombre désigné par cette constante ;
- si *v* est un nom de variable alors *v* est une expression dont la valeur est le nombre affecté à cette variable ;
- si *e*, e_1 et e_2 sont des expressions alors :
  - e_1 `+` e_2 est une expression dont la valeur est la somme de la valeur de l'expression e_1 et de la valeur de l'expression e_2 ;
  - e_1 `-` e_2 est une expression dont la valeur est la différence de la valeur de l'expression e_1 et de la valeur de l'expression e_2 ;
  - e_1 `*` e_2 est une expression dont la valeur est le produit de la valeur de l'expression e_1 par la valeur de l'expression e_2 ;
  - e_1 `/` e_2 est une expression dont la valeur est le quotient de la division de la valeur de l'expression e_1 par la valeur de l'expression e_2 ;
  - e_1 `%` e_2 est une expression dont la valeur est le reste de la division de la valeur de l'expression e_1 par la valeur de l'expression e_2 ;
  - `-`*e* est une expression dont la valeur est l'opposée de la valeur de l'expression *e* ;
  - `(`*e*`)` est une expression dont la valeur est égale à celle de l'expression *e*.

Les actions réalisées par les instructions du langage J sont définies par les règles suivantes :

- si *v* un nom de variable et *e* est une expression alors *v* `:=` *e* est une instruction qui affecte la valeur de l'expression *e* à la variable *v* ;
- si *v* est un nom de variable alors `saisir` *v* est une instruction qui demande à l'utilisateur de saisir un nombre au clavier puis affecte ce nombre à la variable *v* ;
- si *v* est un nom de variable alors `afficher` *v* est une instruction qui affiche à l'écran la le nombre affecté à la variable *v*.

1.2.2 **La pitchoun-machine**

L'architecture de la pitchoun-machine est représentée sur la figure 1.3. Elle est composée d'une unité centrale, d'une mémoire centrale, d'une unité d'entrée : le clavier, d'une unité de sortie : l'écran et d'un bus par lequel transitent les données entre les quatre composants précédents. L'unité centrale est composée :

Instruction	Notation	Action
lecture mémoire	`load` *r*, *a*	le contenu de la case mémoire d'adresse *a* est enregistré dans le registre *r*
écriture mémoire	`store` *r*, *a*	le contenu du registre *r* est enregistré dans la case mémoire d'adresse *a*
addition entière	`add` r_1, r_2, r_3	$val(r_2) + val(r_3)$ est enregistré dans le registre r_1
	`addi` r_1, r_2, *c*	$val(r_2) + nb(c)$ est enregistré dans le registre r_1
soustraction entière	`sub` r_1, r_2, r_3	$val(r_2) - val(r_3)$ est enregistré dans le registre r_1
	`subi` r_1, r_2, *c*	$val(r_2) - nb(c)$ est enregistré dans le registre r_1
multiplication entière	`mult` r_1, r_2, r_3	$val(r_2) \times val(r_3)$ est enregistré dans le registre r_1
	`multi` r_1, r_2, *c*	$val(r_2) \times nb(c)$ est enregistré dans le registre r_1
division entière	`div` r_1, r_2, r_3	$val(r_2) \div val(r_3)$ est enregistré dans le registre r_1
	`divi` r_1, r_2, *c*	$val(r_2) \div nb(c)$ est enregistré dans le registre r_1
reste division entière	`mod` r_1, r_2, r_3	$val(r_2)$ *mod* $val(r_3)$ est enregistré dans le registre r_1
	`modi` r_1, r_2, *c*	$val(r_2)$ *mod* $nb(c)$ est enregistré dans le registre r_1
saisie	`read`	appel au système d'exploitation : le nombre saisi au clavier est enregistré dans le registre `r1`
affichage	`write`	appel au système d'exploitation : le nombre contenu dans le registre `r1` est affiché à l'écran
arrêt	`stop`	arrêt du programme

r, r_1, r_2, r_3	nom de registre (`r0`, `r1`, ..., `r`*n*)
c	constante littérale entière
$val(r)$	nombre contenu dans le registre *r*
$nb(c)$	nombre désigné par la constante littérale entière *c*

Figure 1.4. *Le jeu d'instructions de la pitchoun-machine*

– d'une unité arithmétique et logique qui effectue les opérations arithmétiques et logiques de base et qui comporte un jeu de 32 registres (`r0` à `r31`) de 32 bits chacun dans lesquels sont enregistrées les valeurs des opérandes et les résultats des opérations, à l'exception du registre 0 qui contient toujours le nombre 0 ;
– d'une unité de contrôle qui pilote l'exécution du programme et qui comporte 2 registres : le registre `CO` (compteur ordinal) qui contient l'adresse de la prochaine instruction à exécuter et le registre `RI` (registre instruction) qui contient l'instruction en cours d'exécution.

La mémoire centrale est composée d'un ensemble de 65 536 cases de 32 bits numérotées de 0 à 65 535. L'adresse d'un mot est son numéro. L'unité d'adressage est donc le mot et non l'octet, comme c'est en général le cas dans les processeurs réels.

La pitchoun-machine est munie du jeu d'instructions présenté sur la figure 1.4. Ces instructions réalisent : le transfert d'un nombre de la mémoire vers un registre et inversement, les opérations arithmétiques classiques sur les nombres entiers, la saisie et l'affichage d'un nombre entier. Les opérandes de ces instructions peuvent être soit des noms de registres, soit des nombres entiers.

Une instruction est représentée par un mot de 32 bits structuré selon l'un des deux formats suivants montrés sur la figure 1.5 où *op* est le code opérateur, r_1, r_2 et r_3 sont des numéros de

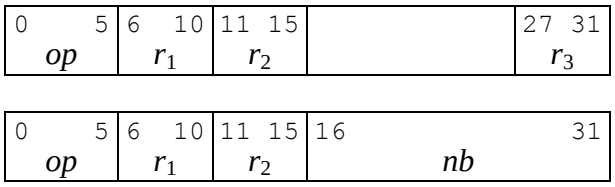


Figure 1.5. *Format des instructions de la pitchoun-machine*

registre et *nb* est un entier codé sur 16 bits qui est soit une adresse, soit le nombre désigné par une constante littérale.

Ce jeu d'instructions, bien que suffisant pour exécuter des programmes écrits dans notre langage jouet J, est bien entendu complété dans un processeur réel par des opérateurs de comparaison, des opérateurs logiques et des instructions de branchement indispensables à l'exécution de véritables programmes.

1.2.3 **Compilation d'un programme J**

La compilation d'un programme est un processus complexe qui se déroule en deux phases : une phase d'analyse à l'issue de laquelle est produite une représentation abstraite du programme et une phase de synthèse dans laquelle cette représentation abstraite est traduite en une suite d'instructions machine : le code exécutable du programme. Nous en donnerons un aperçu en compilant « à la main » le programme *Périmètre* :

– La déclaration de la ligne 1 déclare trois variables : `longueur`, `largeur` et `perimetre`. Le compilateur doit donc allouer une case mémoire à chacune de ces trois variables. Chacune de ces cases est repérée par son adresse. Nous supposerons que le compilateur attribue la case d'adresse 0 à la variable `longueur`, la case d'adresse 1 à la variable `largeur` et la case d'adresse 2 à la variable `perimètre`. On remarquera qu'une déclaration ne génère pas d'instructions dans le code machine d'un programme.

– L'instruction de la ligne 2 demande la lecture du nombre saisi au clavier et son affectation à la variable longueur. Le compilateur traduira cette instruction en langage machine par :

```
read            (le nombre saisi au clavier est enregistré dans le registre r1)
store r1, 0     (le nombre contenu dans le registre r1 est enregistré en mémoire à
                l'adresse 0, celle de la variable longueur)
```

De la même façon, le compilateur traduira l'instruction de la ligne 3 par :

```
read            (le nombre saisi au clavier est enregistré dans le registre r1)
store r1, 1     (le nombre contenu dans le registre r1 est enregistré  mémoire à
                l'adresse 1, celle de la variable largeur).
```

– L'instruction de la ligne 4 affecte la valeur de l'expression `2 * (longueur + largeur)` à la variable `perimetre`. Sa traduction en langage machine est un peu plus difficile. En effet, la pitchoun-machine ne peut réaliser qu'une opération arithmétique à la fois. Or dans cette expression, il y en a deux à réaliser : une addition, celle des valeurs des variables `longueur` et `largeur` et une multiplication : la multiplication par 2 du résultat de l'addition précédente. En supposant que le compilateur décide de charger la valeur de la variable `longueur` dans le registre 1 et celle de la variable `largeur` dans le registre 2, d'enregistrer le résultat de l'addition dans le registre 1 et d'enregistrer le résultat de la multiplication dans le registre 1, l'instruction de la ligne 6 pourra être traduite par :

`addi r1, r0, 2` (le nombre 2 est enregistré dans le registre `r1`)
`load r2, 0` (le nombre enregistré en mémoire à l'adresse 0, celle de la variable `longueur`, est enregistré dans le registre `r2`)
`load r3, 1` (le nombre enregistré en mémoire à l'adresse 1, celle de la variable `largeur`, est enregistré dans le registre `r3`)
`add r2, r2, r3` (le résultat de l'addition des nombres contenus dans les registres `r2` et `r3` est enregistré dans le registre `r2`)
`mult r1, r1, r2` (le résultat de la multiplication des nombres contenus dans les registres `r1` et `r2` est enregistré dans le registre `r1`)
`store r1, 2` (le nombre contenu dans le registre r1 est enregistré en mémoire à l'adresse 2, celle de la variable `perimetre`)

– L'instruction de la ligne 5 demande l'affichage à l'écran de la valeur de la variable `perimetre`. Le compilateur la traduira en langage machine par :

`load r1, 2` (le nombre enregistré en mémoire à l'adresse 2, celle de la variable `perimetre`, est enregistré dans le registre `r1`)

`write` (le nombre enregistré dans le registre `r1` est affiché à l'écran)

– Le point qui termine la ligne 6 demande l'arrêt du programme. Le compilateur le traduira en langage machine par :

```
stop
```

La traduction complète en langage machine du programme *Périmètre* est donc la suivante :

```
read
store r1, 0
read
store r1, 1
addi r1, r0, 2
load r2, 0
load r3, 5
add r2, r2, r3
mult r1, r1, r2
store r1, 2
load r1, 2
write
stop
```

Son exécution pas à pas sur la pitchoun-machine est présentée sur la figure 1.6.

1.3 Anatomie d'un programme C

Un programme C manipule des valeurs classées par types : le type des nombres entiers ou le type des nombres réels, par exemple. Une valeur peut être affectée à une variable. Une variable est une case de la mémoire qui contient la dernière valeur affectée à cette variable. Une variable peut être nommée.

Un programme C est composé d'un ensemble de fonctions. Ces fonctions s'appellent les unes les autres en se transmettant des valeurs. La définition d'une fonction spécifie son nom, le type des valeurs qui peuvent lui être transmises, le type de la valeur qu'elle retourne et son corps qui est une suite d'instructions dont l'exécution, retourne une valeur à la fonction appelante et éventuellement produit des effets de bord tels que des entrées/sorties. Tout programme C doit comporter au moins une fonction : la fonction `main` (fonction principale). L'exécution d'un programme C commence par l'appel de cette fonction.

instruction	registres	mémoire	périphériques
`read`	r1 8 r0 0		saisie de 8
`store r1, 0`	r1 8 r0 0	0 8	
`read`	r1 5 r0 0		saisie de 5
`store r1, 1`	r1 5 r0 0	1 5 0 8	
`addi r1, r0, 2`	r1 2 r0 0		
`load r2, 0`	r2 8 r1 2 r0 0	1 5 0 8	
`load r3, 1`	r3 5 r2 8 r1 2 r0 0	1 5 0 8	
`add r2, r2, r3`	r2 13 r1 2 r0 0		
`mult r1, r1, r2`	r1 26 r0 0		
`store r1, 2`	r1 26 r0 0	2 26 1 5 0 8	
`load r1, 2`	r1 26 r0 0	2 26 1 5 0 8	
`write`	r1 26 r0 0		écriture de 26
`stop`			

Figure 1.6. *Exécution du programme Périmètre*

Un programme C peut appeler des fonctions prédéfinies telles que les fonctions qui permettent de lire ou d'écrire des données ou les fonctions mathématiques usuelles. Ces fonctions constituent la bibliothèque standard de C.

Un programme C est exécuté sous le contrôle du système d'exploitation qui est le logiciel qui assure la gestion de toutes les tâches soumises à l'ordinateur, par exemple : UNIX, Windows de Microsoft ou MacOS d'Apple.

Un programme C communique avec l'extérieur par l'intermédiaire des unités d'entrées-sorties, qui peuvent être un clavier, un écran, une imprimante ou un disque pour ne citer que

les plus classiques. On dit qu'un programme écrit sur une unité de sortie (affichage à l'écran, impression sur papier ou enregistrement sur un disque, …) et qu'il lit sur une unité d'entrée (données saisies au clavier ou enregistrées sur un disque, …).

Exemple 1.1. Le programme suivant que nous appellerons *Maximum* est écrit en C. Il calcule le plus grand des nombres 5 et 19 et affiche ce nombre à l'écran.

```
(1)  /*
(2)   * Calcul du maximum de deux nombres entiers
(3)   */

(4)  #include <stdio.h>

(5)  int max(int x, int y)
(6)     {
(7)     if (x > y)
(8)        return x;
(9)     else
(10)       return y;
(11)    }

(12) int main(void)
(13)    {
(14)    int m;
(15)    m = max(5, 19);
(16)    printf("Le maximum de 5 et 19 est %d.\n", m);
(17)    return 0;
(18)    }
```

Ce programme est construit de la façon suivante :

- Les lignes 1 à 3 contiennent un commentaire qui indique ce qu'effectue le programme.
- La ligne 4 indique que le programme appellera une fonction de la bibliothèque standard : la fonction `printf` pour afficher la valeur du maximum à l'écran (ligne 16).
- Les lignes 5 à 11 constituent la définition de la fonction `max`. La ligne 5 est l'en-tête de cette fonction. Elle indique que cette fonction : retourne un nombre entier (type `int`), a pour nom `max`, et est appelée avec deux nombres entiers qui seront affectées aux variables `x` et `y` de type `int` (nombre entier).
- Le corps de la fonction `max` est le bloc qui commence à la ligne 6 et se termine à la ligne 11. L'instruction `if` qui commence à la ligne 7 et se termine à la ligne 10 demande la comparaison des valeurs des variables `x` et `y`. Si la valeur de `x` est supérieure à celle de `y`, alors (instruction `return` de la ligne 8) la valeur de `x` sera retournée à la fonction qui a appelé la fonction `max`. Sinon (ligne 9), c'est la valeur de `y` qui sera retournée (instruction `return` de la ligne 10).
- Les lignes 12 à 17 constituent la définition de la fonction `main`. Le corps de cette fonction est le bloc qui commence à la ligne 13 et se termine à la ligne 18.
- La ligne 14 contient la définition d'une variable `m` de type `int` (nombre entier) à laquelle sera affecté le maximum calculé.
- L'instruction de la ligne 15 appelle la fonction `max` pour calculer le maximum des nombres 7 et 19 qui lui sont transmis. La valeur retournée par cet appel sera affectée à la variable `m` (opérateur =).

- L'instruction de la ligne 16 appelle la fonction `printf` pour afficher à l'écran la chaîne de caractères : « Le maximum de 5 et 19 est *valeur de la variable* `m`. ».
- L'instruction de la ligne 18 demande que la valeur 0 soit retournée au système d'exploitation pour lui indiquer que le programme s'est bien déroulé.

L'exécution de ce programme se déroule de la façon suivante :

1. La fonction `main` est appelée par le système d'exploitation.
2. La fonction `max` est appelée avec les valeurs 5 et 19 (ligne 15) qui sont affectées respectivement aux variables `x` et `y` de la fonction `max`.
3. La condition `x > y` est testée (ligne 7) : elle est fausse car la valeur 5 de `x` n'est pas supérieure à la valeur 19 de `y` ; l'instruction (ligne 10) introduite par le mot-clé `else` (ligne 9) est donc exécutée et la valeur 19 est retournée à la fonction `main`.
4. La valeur 19 est affectée à la variable `m` (ligne 15).
5. La chaîne de caractères : « Le maximum de 5 et 19 est 19. » est affichée (ligne 16).
6. L'exécution de l'instruction `return 0;` (ligne 17) rend la main au système d'exploitation en lui indiquant (retour de la valeur 0) que le programme s'est bien déroulé. □

1.4 Cycle de vie d'un programme

La vie d'un programme comporte quatre grandes étapes qui s'enchaînent cycliquement, jusqu'à l'abandon de ce programme :

1. **Conception du programme.** Il s'agit d'analyser l'application à programmer pour mettre en évidence les données à manipuler et les opérations à réaliser, choisir les meilleurs algorithmes pour réaliser ces opérations, décider d'une présentation des résultats adaptée aux utilisateurs du programme, etc. C'est une étape cruciale mais difficile lorsque l'application est complexe. Heureusement, il existe maintenant des méthodes et des outils sophistiqués de conception de programmes.
2. **Ecriture du programme**. Il s'agit de traduire le résultat de la conception en un programme écrit dans le langage de programmation choisi. C'est ce que nous allons apprendre à faire dans ce cours avec le langage C.
3. **Compilation.** Pour qu'un programme puisse être exécuté par un ordinateur, il faut le traduire dans le langage machine de cet ordinateur. C'est le rôle du compilateur, qui lit le programme, vérifie s'il est lexicalement, syntaxiquement et sémantiquement correct et si c'est le cas, le traduit en langage machine produisant le code exécutable du programme. Si le programme n'est pas correct, le compilateur signale les erreurs en les localisant aussi précisément que possible dans le texte du programme. Il faut alors retourner à l'étape 2.
4. **Exécution.** Le code exécutable produit à l'issue de la compilation, est soumis au système d'exploitation qui l'exécute. Dans la plupart des cas, un programme est fait pour être exécuté plusieurs fois. Tant qu'aucune modification n'a été apportée à ce programme, il n'est pas nécessaire de le recompiler avant chaque exécution. On conserve le code exécutable et on le soumet au système d'exploitation chaque fois que nécessaire. Il se peut, que l'exécution du programme déclenche des erreurs non détectées à la compilation ou ne satisfasse pas ses utilisateurs, à cause d'une mauvaise écriture ou d'une mauvaise conception du programme. Il faut alors retourner à l'étape 2 ou à l'étape 1.

1.5 Compilation et exécution d'un programme C monofichier

Ce n'est qu'au chapitre 5 que nous apprendrons à organiser et écrire un programme C complet. En attendant, et afin de pouvoir faire les exercices proposés dans les chapitres 1 à 4, voici la marche à suivre pour compiler et exécuter, dans un environnement Unix[2] et avec le compilateur GCC[3], un programme dont le texte est enregistré dans un fichier unique.

1. Saisir le texte du programme sous un éditeur de texte et le sauver dans le fichier *prog*`.c` du répertoire courant (*prog* est le nom du programme).
2. Compiler le programme en lançant la commande :

   `gcc -Wall` *prog*`.c -o` *prog*

   Si le compilateur détecte des erreurs, elles seront affichées et il faudra les corriger. S'il n'y pas d'erreurs, le fichier exécutable *prog* est créé. L'option `-Wall`, bien que facultative, est recommandée car elle génère une liste très complète d'avertissements (« warnings ») sur des incorrections éventuelles ou des oublis dans le texte du programme qui pourraient provoquer des problèmes lors de l'exécution.
3. Exécuter le programme en lançant la commande :

   prog

   Le système d'exploitation appellera alors la fonction `main`. Cet appel déclenchera l'exécution du programme.

Dans le cas où l'option `-o` *prog* n'est pas présente, le fichier exécutable généré aura pour nom `a.out` et l'exécution du programme sera lancée par la commande `a.out`.

Par exemple, si le programme est le programme *Maximum* du paragraphe 1.3 dont le texte est enregistré dans le fichier source `maximum.c`, sa compilation sera lancée par la commande :

```
gcc -Wall maximum.c -o maximum
```

et son exécution par la commande :

```
maximum
```

Il est conseillé de regrouper les programmes correspondant aux exercices dans un même répertoire et de donner au fichier source un nom composé de la façon suivante : `exo`*c*`.`*n*`.c` où *c* est le numéro du chapitre et *n* le numéro de l'exercice dans le chapitre.

Exercices

Exercice 1.1. Saisir et exécuter le programme *Maximum*.

[2] Par exemple Linux ou bien Cygwin qui est un environnement « à la UNIX » pour Windows.

[3] GCC (GNU Compiler Collection) est une collection de compilateurs développée par le projet GNU : un projet de système d'exploitation à la UNIX composé uniquement de logiciels libres.

2
Données

Un programme C manipule des données. Une donnée a un type et une valeur, elle peut avoir un nom et elle peut être constante ou variable. La valeur d'une donnée variable, ou plus simplement d'une variable, peut changer au cours de l'exécution d'un programme alors que celle d'une donnée constante, ou plus simplement d'une constante, ne le peut pas. Une variable est une case de la mémoire qui est identifiée par son adresse et qui contient la valeur de cette variable.

Supposons, par exemple, que l'on veuille écrire un programme qui calcule le volume d'une sphère dont le rayon est saisi au clavier de son poste de travail par l'utilisateur du programme. Le nombre π est une constante. Le rayon et le volume de la sphère sont des variables. Il y aura donc dans la mémoire une case nommée *rayon* qui contiendra le rayon de la sphère une fois qu'il aura été saisi par l'utilisateur et une case nommée *volume* qui contiendra le volume de cette sphère une fois qu'il aura été calculé.

Une valeur peut être :

- un nombre entier ;
- un nombre flottant (une approximation d'un nombre réel) ;
- une structure composée d'une suite de variables qui peuvent être de types différents (par exemple, une structure représentant un point d'un plan, composée du nom, de l'abscisse et de l'ordonnée de ce point) ;
- une union qui est une variable dont les valeurs peuvent être de types différents (par exemple, une union dont la valeur pourra être un nombre entier ou un nombre flottant) ;
- l'adresse d'une variable ou d'une fonction.

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

Il est de plus possible de définir des tableaux qui, en C, ne sont pas des variables mais des suites de variables de même type rangées de façon contigüe dans la mémoire. Par exemple, le tableau des nombres d'habitants des dix plus grandes villes de France.

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 zéro et le booléen « vrai » par toute valeur différente de zéro. Un caractère est codé par un nombre entier (conformément au code ASCII[1], en général). Une chaîne de caractères est représentée comme un tableau de caractères (c.-à-d. comme un tableau de nombres entiers).

Dans la suite de ce chapitre, nous étudierons : la représentation en mémoire des nombres et plus particulièrement celle des nombres flottants (paragraphe 2.1) ; les types numériques entiers et flottants (paragraphe 2.2) ; les identificateurs qui servent à nommer les entités d'un programme (paragraphe 2.3) ; les constantes littérales qui désignent des valeurs (paragraphe 2.4) ; la définition des variables de type entier ou flottant (paragraphe 2.5). Nous expliquerons

[1] Un code international à 128 ou 256 (dans ses versions étendues) positions qui permet de coder les caractères courants des langues latines : chiffres, lettres minuscules et majuscules, caractères de ponctuation, …

ensuite, comment est organisée la mémoire d'un programme et nous proposerons une représentation graphique des variables contenues dans cette mémoire (paragraphe 2.6).

2.1 Représentation des nombres

Sur un ordinateur, un nombre est représenté dans un espace mémoire de taille finie mesurée en nombre de bits 8, 16, 32 ou 64 bits en général.

Si l'on dispose d'une taille de t bits, on peut représenter tous les entiers positifs ou nuls compris entre 0 et $2^t - 1$ ou bien tous les entiers négatifs, positifs ou nul compris entre $-2^{t-1} + 1$ et $2^{t-1} - 1$.

Par contre, pour les nombres réels c'est plus complexe, car il faut tenir compte de la partie entière et de la partie décimale qui peut être de longueur infinie. Si le nombre de bits dont on dispose pour représenter un nombre réel est fini, on ne pourra pas représenter tous les nombres réels qui appartiennent à un intervalle donné. Dans un ordinateur, un nombre réel est donc représenté par le nombre décimal qui en est le plus proche et qui constitue donc une approximation de ce nombre réel.

Deux représentations sont possibles pour représenter un nombre décimal : en virgule fixe ou en virgule flottante. En voici les principes.

Dans la représentation en virgule fixe, on fixe le nombre de chiffres après la virgule. Un nombre décimal est représenté par nombre entier n. Si b est la base et k le nombre de chiffres après la virgule, le nombre décimal représenté par l'entier n sera $n \times b^{-k}$. Par exemple, si k = 3, le nombre 176,48 sera représenté par le nombre 17648. Le problème posé par cette représentation est que la partie décimale devra être tronquée si le nombre de chiffres après la virgule est supérieur à k. Par exemple, le nombre 6,4817 sera représenté par 6481 et donc tronqué car $6481 \times 10^{-3} = 6{,}481$. Remarquons que bien que les nombres 6,4817 et 176,48 aient le même nombre de chiffres significatifs, il a fallu tronquer le premier mais pas le second.

Pour résoudre ce problème, une autre représentation a été proposée : la représentation en virgule flottante qui consiste à introduire un exposant. La représentation d'un nombre décimal en virgule flottante est un doublet (m, e) où m est la mantisse et e est l'exposant. Si b est la base, le nombre représenté par le doublet (m, e) sera égal à $m \times b^e$. Par exemple, si b = 10, le nombre 176,48 sera représenté par le doublet (17648 ; −2) et le nombre 6,4817 sera représenté par le doublet (64817 ; −4). C'est l'exposant qui détermine la place de la virgule : d'où le nom de cette représentation.

Dans la représentation en virgule flottante, la taille de l'exposant détermine l'intervalle auquel doivent appartenir les nombres décimaux représentables et la taille de la mantisse détermine la précision avec laquelle ces nombres peuvent être représentés c.-à-d. leur nombre de chiffres significatifs).

Dans la plupart des architectures actuelles d'ordinateurs, les nombres décimaux sont représentés en virgule flottante selon la norme IEEE 754. Cette norme définit deux formats principaux :

- le format simple précision pour des nombres codés sur 32 bits, qui permet de représenter des nombres dans l'intervalle 10^{-37} à 10^{37} avec une précision de 9 chiffres décimaux ;
- le format double précision pour des nombres codés sur 64 bits, qui permet de représenter des nombres dans l'intervalle 10^{-307} à 10^{307} avec une précision de 16 chiffres décimaux.

	Nom	Extension minimale
Entiers	`signed char`	−127 à 127
	`unsigned char`	0 à 255
	`char`	`signed char` ou `unsigned char` selon les implantations
	[`signed`] `short` [`int`]	−32 767 à 32 767
	`unsigned short` [`int`]	0 à 65 535
	[`signed`] `int`	−32 767 à 32 767
	`unsigned int`	0 à 65 535
	[`signed`] `long` [`int`]	−2 147 483 647 à 2 147 483 647
	`unsigned long` [`int`]	0 à 4 294 967 295

	Nom	Précision minimale	Extension minimale
Flottants	`float`	6 chiffres décimaux	10^{-37} à 10^{+37}
	`double`	10 chiffres décimaux	10^{-37} à 10^{+37}
	`long double`	10 chiffres décimaux	10^{-37} à 10^{+37}

Figure 2.1. *Types numériques de C*

2.2 Types numériques

Le tableau de la figure 2.1 présente les types numériques du langage C.

On distingue les types entiers et les types flottants. Les types entiers sont le type `char`, les types entiers signés `signed char`, `short int`, `int`, `long int` et les types entiers non signés `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`. Les types flottants sont `float`, `double` et `long double`.

On notera que les noms des types entiers ont plusieurs synonymes (la paire de crochets autour d'un mot indique que ce mot est facultatif). Par exemple, `short` (le plus employé), `signed short` et `signed short int` sont des synonymes de `short int`.

Le type `char` est destiné à la manipulation des caractères du jeu de caractères de la machine utilisée (en général, ceux du code ASCII ou d'un code ASCII étendu). Le code ASCII de base prend en compte tous les caractères nécessaires à l'écriture de l'anglais : il leur attribue un code numérique de 0 à 127 (7 bits). Les codes ASCII étendus profitent du 8^{e} bit pour attribuer un code numérique aux caractères des langues européennes non pris en compte par le code ASCII de base, les caractères accentués entre autres. Parmi ceux-ci, le code ISO-8859-15 prend en compte tous les caractères de l'allemand, du français, de l'italien, etc.

Attention ! Dans ce cours, nous n'utiliserons que les caractères du code ASCII de base dans les textes des programmes et dans les fichiers lus par ces programmes. Les caractères accentués ne seront donc pas utilisés. □

Le type `int` est destiné à la représentation des entiers dont la taille est manipulée la plus efficacement par la machine utilisée (en général, 2 ou 4 octets).

L'intérêt des types non signés est de permettre la représentation dans le même espace mémoire de nombres positifs ou nuls deux fois plus grands que s'ils pouvaient aussi être négatifs.

Les types `float`, `double` et `long double` sont destinés à la représentation des nombres en virgule flottante simple précision, double précision et double précision étendue. La représentation d'un nombre flottant est caractérisée par son signe, sa mantisse et son exposant :

$$signe \times mantisse \times 2^{exposant}$$

L'extension et la précision de ces types sont dépendantes de chaque implantation, mais les valeurs minimales indiquées dans le tableau de la figure 2.1 ainsi que le respect des contraintes suivantes sont garanties par la norme ANSI :

- *extension*(`signed char`) ⊆ *extension*(`signed short int`) ⊆ *extension*(`int`) ⊆ *extension*(`signed long int`)
- *extension*(`float`) ⊆ *extension*(`double`) ⊆ *extension*(`long double`)
- *précision*(`float`) ≤ *précision*(`double`) ≤ *précision*(`long double`)
- *extension*(`signed` *T*) ⊆ *extension*(`unsigned` *T*) où *T* ∈ {`char`, `short int`, `int`, `long int`}
- La place mémoire occupée par un entier de type `signed` *T* est la même que celle occupée par un entier de type `unsigned` *T* et la représentation d'un même entier dans chacun de ces deux types est la même.

Les opérations de changement de type (voir chapitre 3, paragraphe 3.2) s'appuient sur le respect de ces contraintes. La 1ère contrainte garantit notamment qu'un entier de type `short int` est convertible en type `int` ou `long int`. Les 2e et 3e contraintes garantissent notamment qu'un flottant de type `float` est convertible en type `double` ou `long double`. Enfin, la 4e contrainte garantit que la conversion d'un type signé en un type non signé se fait sans travail.

L'extension et la précision de ces types pour une implantation particulière est définie par un ensemble de constantes symboliques enregistrées dans les fichiers d'en-têtes `limits.h` et `float.h` de la bibliothèque standard, fichiers que nous apprendrons à utiliser au chapitre 5. Les constantes `INT_MIN` et `INT_MAX`, par exemple, désignent les valeurs minimum et maximum d'un entier de type `int`. Une bonne précaution consiste à paramétrer, dans les tests de débordement, les bornes minimales et maximales par ces constantes symboliques.

Attention ! Pour assurer la portabilité d'un programme, il faudra choisir un type numérique par rapport à son extension et à sa précision minimales et non par rapport à celles de ce type dans l'implantation de C sur laquelle est développé le programme. Notamment, il ne faut utiliser le type `char` que pour la manipulation des caractères et le type `int` que pour des entiers compris entre −32 767 à 32 767. □

2.3 Identificateurs

Les identificateurs servent à nommer les constantes, les variables, les structures, les unions, les champs d'une structure ou d'une union, les tableaux et les fonctions.

Un identificateur est une suite contiguë de caractères dont chacun peut être une lettre, un chiffre ou le caractère _ (blanc souligné). Par convention, les identificateurs commençant par un blanc souligné sont réservés à la bibliothèque standard. Tous les identificateurs d'un programme utilisateur devront donc commencer par une lettre.

Un identificateur peut avoir de 1 à 31 caractères. Une lettre majuscule est un caractère différent de la même lettre en minuscule.[2]

Par exemple :

```
x    y1   ma_variable
```

sont des identificateurs valides.

Les mots suivants dits « mots réservés » qui jouent un rôle spécifique dans la syntaxe de C, ne peuvent pas être utilisés comme identificateurs :

```
auto break case char const continue default do double else enum extern
float for goto if int long register return short signed sizeof static
struct switch typedef union unsigned void volatile while
```

2.4 Constantes

Les valeurs sont des abstractions, mais il faut pouvoir les représenter (les écrire) dans le texte d'un programme. Dans un langage de programmation, on appelle constante la représentation d'une valeur. En C, on distingue : les constantes littérales, les constantes symboliques et les expressions constantes.

2.4.1 **Constantes littérales**

Les nombres positifs, les caractères et les chaînes de caractères ont une représentation littérale : une suite de caractères.

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

```
0    1515
```

sont des constantes littérales de type entier qui représentent respectivement les entiers 0 et 1515. La convention que nous avons adoptée d'employer une fonte particulière pour écrire un extrait de texte de programme C, permet de bien faire la distinction entre une constante : `1515`, et la valeur qu'elle représente : 1515.

Une constante littérale de type entier est la représentation d'une valeur du plus petit des types `int`, `long`, `unsigned long` qui contient cette valeur. Par exemple, si le type `int` est codé sur 2 octets, les constantes littérales `17` et `12519` représentent des valeurs de type `int`.

Une constante littérale de type flottant est formée :

[2] **Attention !** Cette règle n'est valable que pour les identificateurs internes, ceux qui sont définis dans le même fichier source que celui dans lequel ils sont utilisés. Elle est plus restrictive pour les identificateurs externes qui peuvent être limités à 6 caractères sans distinction entre minuscule et majuscule pour une même lettre. La notion d'identificateur externe sera étudiée au chapitre 5.

- 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 :

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

sont des constantes littérales qui représentent respectivement les nombres :

273,15 0,15 2000

Une constante littérale de type flottant est la représentation d'une valeur de type `double`.

Une constante littérale de type caractère est notée en plaçant ce caractère entre apostrophes (`'`). Par exemple : `'A'`, `'9'`, `';'`.

Une constante littérale de type chaîne de caractères est notée en plaçant cette chaîne de caractères entre guillemets (`"`). Par exemple :

```
"Bonjour"
```

Dans une constante littérale de type caractère ou chaîne de caractères, tout caractère imprimable ou non, peut être représenté par un caractère ou un nombre octal (en base 8) précédé du caractère d'échappement « `\` » (anti-slash). Entre autres :

`\n`	nouvelle ligne (LF),
`\r`	retour chariot (CR),
`\t`	tabulation,
`\'`	apostrophe (pour la distinguer du délimiteur de constante caractère),
`\"`	guillemet (pour la distinguer du délimiteur de constante chaîne de caractères),
`\\`	anti-slash (pour le distinguer du caractère d'échappement),
`\0`	caractère nul (caractère ayant le code ASCII 0).

Par exemple, la constante littérale `"Il a dit : \"Bonjour\""` représente la chaîne de caractères : « Il a dit : "Bonjour" ».

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

Nous verrons au chapitre 9 consacré à la manipulation des chaînes de caractères ce que représente exactement une constante littérale de type chaîne de caractères.

2.4.2 **Constantes symboliques**

Une constante symbolique est un nom donné à une valeur. Il y a deux façons de donner un nom à une valeur : les types énumérés que nous n'étudierons pas dans ce cours et les définitions de macros qui le seront au chapitre 5.

Par exemple :

```
#define PI 3.14
```

définit une constante symbolique `PI` qui désigne le nombre flottant 3.14 : une valeur approchée de π.

2.4.3 **Expressions constantes**

Une expression constante est une expression construite à partir de constantes et d'opérateurs (les opérateurs arithmétiques, par exemple). Une expression constante a pour particularité de ne pas dépendre de l'exécution du programme, puisqu'elle ne contient pas de variables. Sa valeur peut donc être calculée lors de la compilation du programme. Par exemple, si `M` est une constante symbolique qui représente l'entier 20 et `N` est une constante symbolique qui représente l'entier 30, alors :

```
N + M + 1
```

est une expression constante qui a pour valeur l'entier 51.

2.5 Variables

Nous nous restreindrons pour le moment à la définition de variables numériques (de type entier ou flottant). La forme la plus simple d'une définition de variable numérique est la suivante :

T var_1 = exp_1, …, var_n = exp_n;

où :

- *T* est l'un des types numériques du tableau de la figure 2.1 ;
- var_1, …, var_n sont les noms des variables définies (des identificateurs) ;
- exp_1, …, exp_n sont des expressions dont les valeurs constituent les valeurs initiales respectives des variables définies.

Cette définition déclare *n* variables de type *T*, nommées var_1, …, var_n, et les crée. La déclaration d'une variable consiste à lui associer un type et la création d'une variable consiste à lui attribuer une case de la mémoire et à y enregistrer sa valeur initiale, si elle est indiquée.

L'indication d'une valeur initiale est facultative, il suffit d'omettre le terme « = *exp* ».

L'expression d'une valeur initiale ainsi que la valeur initiale affectée à une variable lorsque cette valeur n'est pas spécifiée dans la définition de cette variable, diffèrent selon que cette variable est globale ou locale comme nous le verrons aux chapitres 4 et 5.

Par exemple, les définitions :

```
int i, j, k;
float longueur = 10.5, largeur = 6.34;
char une_lettre;
```

définissent respectivement :

- trois variables `i`, `j` et `k` de type `int` ;
- deux variables `longueur` et `largeur` de type `float` et de valeurs initiales respectives 10,5 et 6,34 ;
- une variable `une_lettre` de type `char`.

On peut indiquer au compilateur qu'une variable ne changera pas de valeur au cours de l'exécution d'un programme, en préfixant sa définition par le mot-clé `const`. En ce cas, sa valeur devra être fournie dans la définition de cette variable. Cette indication pourra être utilisée par C pour contrôler que le programme ne modifie pas cette valeur.

Par exemple, la définition :

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

signale au compilateur que la variable `pi` a la valeur 3.14 et n'en changera pas.

2.6 Mémoire d'un programme

La mémoire d'un programme contient les informations dont il a besoin pour son exécution. Elle est constituée de 4 parties :

- le code exécutable du programme qui est constitué de la suite des instructions machine à exécuter. Il n'est pas modifié lors de l'exécution du programme.
- la mémoire statique qui contient les variables globales.
- la pile, qui contient les variables locales (celles qui sont définies dans le corps des fonctions). La pile évolue par l'ajout d'une variable à son sommet ou par la suppression de la variable placée à son sommet.
- le tas, qui contient des variables allouées dynamiquement au cours de l'exécution du programme. Lors de l'exécution d'un programme le tas évolue par l'allocation d'une variable dans un emplacement disponible au moment de cette allocation ou par la suppression d'une variable allouée dynamiquement et la restitution de l'emplacement qu'elle occupait.

Dans ce cours, nous utiliserons la visualisation graphique suivante des variables et de la pile.

- une variable *var*, de type *T* et de valeur *v* sera représentée par :

  T var | *v* |

  où le rectangle représente la case mémoire dans laquelle est placée la valeur de la variable (le type pourra être omis) ;
- un extrait de la pile ou de la mémoire statique apparaitra comme une suite de variables rangées dans l'ordre de leur déclaration, de bas en haut ou éventuellement de gauche à droite dans le cas des tableaux et le fond de la pile sera marqué par un double trait :

  T var | *v* |

  T var | *v* |

Dans un programme C, il peut y avoir plusieurs variables de même nom mais nous verrons que les règles de visibilité font qu'en un point du texte d'un programme, il ne peut pas y avoir deux variables visibles de même nom. En conséquence, la valeur d'une variable dont le nom est donné est celle qui est liée à l'unique variable visible ayant ce nom. Dans notre représentation graphique, le nom d'une variable invisible en un point d'un programme sera grisé :

T var | *v* |

Attention ! Cette visualisation graphique n'est qu'un modèle sur lequel nous nous appuierons pour faire comprendre comment se déroule l'exécution d'un programme C. C'est une vision

très schématique de la réalité. De plus, dans cette représentation, le type et le nom des variables sont indiqués afin de faire le lien avec le texte du programme, alors que dans le code exécutable d'un programme ce type et ce nom ont disparu. □

Exemple 2.1. Considérons, par exemple, le programme C suivant qui définit trois variables `x`, `y` et `s` (ligne 3), demande à l'utilisateur de saisir la valeur des variables `x` et `y` (lignes 4 et 5), affecte à la variable `s` la somme des valeurs de `x` et de `y` (ligne 6) et enfin affiche la valeur de `s` (ligne 7).

```
(1) int main(void)
(2)     {
(3)     int x, y, s;
(4)     saisir la valeur de x;
(5)     saisir la valeur de y;
(6)     s = x + y;
(7)     afficher la valeur de s;
(8)     return 0;
(9)     }
```

Quand le bloc qui constitue le corps de la fonction `main` est exécuté (lignes 2 à 8) les variables qui y sont définies (`x`, `y` et `s` dans cet exemple) sont empilées dans cet ordre. Quand la valeur de `x` a été saisie par l'utilisateur, elle est lue et affectée dans la case associée à `x`. Il en est de même pour `y`. Enfin, la valeur de l'expression `x + y` est calculée et rangée dans la case associée à `s`.

L'état de la pile immédiatement après l'exécution de l'instruction 6 sera le suivant, si les valeurs saisies pour `x` et `y` sont 3 et 7 :

`int s`	10
`int y`	7
`int x`	3

□

Exercices

Exercice 2.1. Ecrire en C les constantes littérales représentant :

- le nombre entier 1024 ;
- le nombre réel 3,141592 avec et sans partie décimale ;
- le nombre réel $1{,}602\times10^{-19}$ (valeur approchée de la charge de l'électron) ;
- les booléens vrai et faux ;
- le caractère Z.

Exercice 2.2. Définir :

- une variable `compteur` de type `int` initialisée à 60 ;
- une variable de nom `e` et de type `float` dont la valeur est fixée à $1{,}602.10^{-19}$.

Exercice 2.3. Soit le programme C :

```
(1)  #include <stdio.h>

(2)  int main(void)
(3)     {
(4)     int v;
(5)     float m, ec;
(6)     m = 1e6;
(7)     v = 30;
(8)     ec = (m * v * v) / 2;
(9)     printf("m = %e v = %d ec = %e\n", m, v, ec);
(10)    return 0;
(11)    }
```

1. Dessiner la pile immédiatement après l'exécution de l'instruction 8.
2. Exécuter le programme.

3
Expressions et opérateurs

Une expression est formée à l'aide de constantes, de noms de variables et d'opérateurs. Une expression est destinée à être évaluée. Par exemple, dans un état du programme où la variable `x` a la valeur 8 :

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

sont des expressions qui ont pour valeurs respectives 12, 8, 17 et 1 (« vrai »).

En C, toute expression a obligatoirement un type et une valeur. Elle peut avoir une adresse et un effet de bord qui change l'état de la mémoire ou déclenche des entrées-sorties.

Si *e* est une expression nous noterons :

- *type*(*e*) son type ;
- *val*(*e*) sa valeur ;
- *adr*(*e*) son adresse.

Une expression peut être mise entre parenthèses. Si *exp* est une expression, alors `(`*exp*`)` est une expression équivalente à *exp*.

Une expression peut être :

- soit une expression simple : une constante ou un nom de variable ;
- soit une expression composée à l'aide d'opérateurs.

Dans la suite de ce chapitre nous étudierons : les expressions simples (paragraphes 3.1) ; les expressions composées à l'aide d'opérateurs (paragraphe 3.2) ; les règles de priorité et d'associativité de ces opérateurs dont l'application permet d'améliorer la lisibilité des expressions en diminuant le nombre de parenthèses nécessaires (paragraphe 3.3) ; l'ordre d'évaluation de leurs opérandes auquel il faut veiller lorsque ces opérandes ont un effet de bord (paragraphe 3.4). Dans la plupart des programmes, il est nécessaire de lire ou d'écrire des données. Nous terminerons ce chapitre en présentant une version simplifiée des fonctions qui permettent de le faire (paragraphe 3.6).

3.1 Expressions simples

Une expression simple est soit une constante, soit un nom de variable.

Une constante *c* est une expression qui a les propriétés suivantes :

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

Par exemple :

– `125` est une expression de type `int` qui a pour valeur le nombre 125 ;
– `3.14` est une expression de type `double` qui a pour valeur le nombre 3,14.

Un nom de variable *n*, est une expression qui a les propriétés suivantes :
– *type*(*n*) = type spécifié dans la déclaration de *n* ;
– *val*(*n*) = valeur de la variable *n* ;
– *adr*(*n*) = adresse de la variable *n*.

Attention ! Un même nom peut être associé à des constantes ou à des variables différentes dans un même programme. La conséquence est que lorsqu'un nom apparaît dans une expression, il faut pouvoir déterminer de façon univoque quelle est la valeur ou la variable associée à ce nom. Cela se fait en appliquant les règles de visibilité des déclarations, que nous étudierons au chapitre 5, qui ont pour conséquence qu'il ne peut pas y avoir deux entités (type, constante ou variable) de même nom qui sont visibles en un même point du texte d'un programme. Ces règles peuvent se résumer de la façon suivante : un nom dans une expression se rapporte à sa déclaration la plus récente dans l'ordre de lecture du programme. □

3.2 Opérateurs

Dans ce chapitre nous étudierons :
– les opérateurs arithmétiques ;
– les opérateurs booléens ;
– la conversion de type ;
– l'affectation.

L'opérateur d'appel de fonction sera étudié aux chapitres 5 et 8. Les opérateurs de manipulation de structures et de tableaux seront étudiés au chapitre 7 et ceux de manipulation des adresses le seront au chapitre 8. Nous n'étudierons, dans ce cours, ni les opérateurs de traitement des bits, ni l'opérateur d'évaluation conditionnelle.

Les opérateurs arithmétiques ou de comparaison peuvent s'appliquer à des opérandes de types différents, mais pour que l'opération puisse être effectuée, il faut au préalable convertir ces opérandes en un même type que nous appellerons le type commun. Intuitivement, il est clair que le type commun doit être le plus grand, au sens de son extension et de sa précision, des types des opérandes. Mais le fait que C impose que ce type ne soit pas plus petit que le type `int` et le fait que le type `int` oscille entre le type `short` et le type `long` au gré des implantations, rend un peu plus compliqué le choix du type commun. Ce choix s'appuie sur l'ordre suivant des types numériques :

`signed char` = `unsigned char` = `char` = `short` = `unsigned short` < `int` < `unsigned int` < `long` < `unsigned long` < `float` < `double` < `long double`

Soit T_1 et T_2 le type des opérandes et T le type commun recherché. Choisir pour T le plus grand des types `int`, T_1 et T_2. Si le type obtenu pour T est `int` et que l'extension de T_1 ou celle de T_2 n'est pas incluse dans celle du type `int`, choisir pour T le type `unsigned int`. Si le type T obtenu est `long` et que l'extension de T_1 ou celle de T_2 n'est pas incluse dans celle du type `long`, choisir pour T le type `unsigned long`.

Ces conversions ont pour conséquence que lorsque les valeurs des opérandes des opérateurs arithmétiques ou de comparaison sont des entiers, ceux-ci son tout d'abord convertis en type `int` ou `unsigned int`. C'est ce que l'on appelle la « promotion entière ».

3.2.1 **Moins unaire**

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

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

`-`*exp*

est une expression qui a les propriétés suivantes :

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

3.2.2 **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 (où *op* est l'un des opérateurs `+ - * / %`) :

– *type*(exp_1 *op* exp_2) = type commun de *type*(exp_1) et de *type*(exp_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 type commun

Si exp_1 et exp_2 sont de type entier, l'opérateur `/` effectue la division entière de leurs valeurs.

Par exemple :

– l'expression `((4.0 + 2.25) * 2.0)` est de type `double` (flottant) et a la valeur 12,5 ;
– l'expression `15 / 2` (division entière) est de type `int` et a la valeur 7 ;
– l'expression `15.0 / 2` est de type `double` (flottant) et a la valeur 7,5 ;
– l'expression `15 % 2` est de type `int` et a la valeur 1.

Attention ! Comme nous venons de le voir, lorsque ses opérandes sont des entiers, l'opérateur `/` calcule le quotient de la division de ces deux entiers. Supposons que l'on veuille calculer la moyenne arithmétique des deux entiers 4 et 7. On ne pourra pas l'obtenir en évaluant l'expression `(4 + 7) / 2`, car on obtiendra l'entier 5 et non le flottant 5,5 attendu. Pour l'obtenir, il faudra forcer la division flottante en écrivant l'un des opérandes sous forme d'une constante littérale de type flottant ou en convertissant sa valeur en un flottant à l'aide de l'opérateur de changement de type (3.2.6 ci-dessous). Par exemple, l'expression :

```
(4 + 7) / 2.0
```

a la valeur 5,5. □

3.2.3 **Comparateurs**

Les comparateurs `==` `!=` `<` `<=` `>` `>=` testent respectivement l'égalité, la différence, l'infériorité, l'infériorité ou l'égalité, la supériorité, la supériorité ou l'égalité, de deux nombres. Rappelons qu'en C, le booléen « faux » est représenté par 0 et le booléen « vrai » par toute valeur différente de 0.

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 (où *op* est l'un des comparateurs `==`, `!=`, `<`, `<=`, `>` ou `>=`) :

- $type(exp_1\ op\ exp_2)$ = `int`
- $val(exp_1\ op\ exp_2) = v_1\ op\ v_2$, où v_1 et v_2 sont le résultat de la conversion de $val(exp_1)$ et de $val(exp_2)$ en type commun
- $v_1\ op\ v_2$ est égal à 1 si la comparaison est vraie, 0 sinon

Supposons, par exemple, que la variable `mois` contienne le rang d'un mois dans une année. L'expression `mois == 4` a la valeur 1 (vrai) pour le mois d'avril et 0 (faux) pour les autres mois. L'expression `mois <= 2` a la valeur 1 pour les mois de janvier et de février et 0 pour les autres mois.

Attention ! L'opérateur d'égalité est `==` et non `=` qui est, comme nous le verrons au paragraphe 3.3.7, l'opérateur d'affectation. □

3.2.4 **Négation**

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

Si *exp* est une expression, alors :

`!`*exp*

est une expression qui a les propriétés suivantes :

- $type(exp)$ = `int`
- $val(\texttt{!}exp)$ = si $val(exp) = 0$ alors 1 sinon 0

3.2.5 **Conjonction et disjonction.**

Les opérateurs `||` et `&&` calculent respectivement la disjonction et la conjonction 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 (où *op* est l'un des opérateurs `||` ou `&&`) :

- $type(exp_1\ op\ exp_2)$ = `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

Il est important de noter que 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 `&&`.

Exemple 3.1. Supposons que la variable `mois` contienne le rang d'un mois dans une année (1 à 12) et la variable `jour`, le rang d'un jour dans un mois (1 à 31). Les expressions suivantes indiquent respectivement si le couple (*val*(`mois`), *val*(`jour`)) représente ou non, un jour d'hiver, un jour de printemps, un jour d'été ou un jour d'automne de l'année 2006 :

```
((mois == 12) && (jour >= 22)) || (mois < 3)
|| ((mois == 3) && (jour < 20))

((mois == 3) && (jour >= 20)) || ((mois > 3) && (mois < 6))
|| ((mois == 6) && (jour < 21))

((mois == 6) && (jour >= 21)) || ((mois > 6) && (mois < 9))
|| ((mois == 9) && (jour < 23))

((mois == 9) && (jour >= 23)) || ((mois > 9) && (mois < 12))
|| ((mois == 12) && (jour < 21))
```

Nous verrons ci-dessous (paragraphe 3.3) que l'application des règles de priorité et d'associativité des opérateurs permettent d'alléger ces expressions en supprimant certaines paires de parenthèses. □

3.2.6 **Changement de type**

L'opérateur `(`*T*`)`, où *T* est un nom de type, convertit une valeur en une valeur de type *T*.

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

`(`*T*`)` *exp*

où *T* est un type numérique ou un type d'adresse est une expression qui a les propriétés suivantes :

– *type*(`(`*T*`)` *exp*) = *T*

– *val*(`(`*T*`)` *exp*) = *val*(*exp*) convertie en *T*

Une conversion est réalisée conformément aux règles suivantes qui s'appuient sur l'ordre des types présenté dans l'introduction de ce paragraphe :

- La conversion d'une valeur de type entier en une valeur de type entier de taille supérieure ou égale produit la même valeur.
- La conversion d'une valeur *v* de type entier T_1 en une valeur de type entier T_2 de taille inférieure à celle de T_1 produit *v* si *v* appartient à l'extension de T_2. Elle produit une valeur tronquée sinon.
- La conversion d'une valeur *v* de type entier en une valeur de type flottant *T* produit le flottant de type *T* le plus proche de *v* si *v* appartient à l'extension de *T*. Sinon, le résultat de la conversion est imprévisible.
- La conversion d'une valeur *v* de type flottant en une valeur de type entier *T* produit la partie entière de *v* si celle-ci appartient à l'extension de *T*. Sinon, le résultat de la conversion est imprévisible.
- La conversion d'une valeur de type flottant T_1 en une valeur de type flottant T_2 de précision supérieure ou égale à celle de T_1 produit la même valeur.
- La conversion d'une valeur *v* de type flottant T_1 en une valeur de type flottant T_2 de précision inférieurs à celle de T_1 produit le flottant de type T_2 le plus proche de *v* si *v* appartient à l'extension de T_2. Sinon, le résultat de la conversion est imprévisible.
- La conversion d'une adresse de type T_1 en une adresse de type T_2, ne modifie pas cette adresse.

Par exemple, si :

- `c` est une variable de type `unsigned char` et de valeur 32 ;
- `i1`, `i2` et `i3` sont des variables de type `int` et de valeurs respectives 32, 36 et 549 ;
- `f` est une variable de type `float` de valeur 67,893.

et que la valeur maximum d'un nombre de type `unsigned char` est 255 ($2^8 - 1$), alors :

- l'expression `(int) c` a le type `int` et la valeur 32 ;
- l'expression `(unsigned char) i2` a le type `unsigned char` et la valeur 36 ;
- l'expression `(unsigned char) i3` a le type `char` et la valeur 37 (549 modulo 256) ;
- l'expression `(float) i3` a le type `float` et la valeur 549 ;
- l'expression `(int) f` a le type `int` et la valeur 67 ;
- l'expression `(i2 + (float i3)) / 2` a le type `float` et la valeur 292,5 (la conversion de `i3` en flottant a forcé l'addition puis la division à être des opérations sur les flottants).

Des exemples de changement de type d'une adresse seront donnés au chapitre 8 sur les pointeurs.

Attention ! Il n'est pas possible de changer le type d'une structure ou d'une union. □

3.2.7 **Affectation**

L'opérateur = affecte une valeur à une variable.

Le fait qu'en C l'affectation soit un opérateur et non comme dans d'autres langages de programmation une instruction, oblige à identifier un type particulier d'expression : les valeurs gauches qui sont les expressions qui peuvent être placées à gauche de l'opérateur d'affectation (=). On comprendra facilement que ce sont celles qui désignent des variables. Une valeur gauche a une adresse qui est celle de la variable qu'elle désigne.

Par opposition, les expressions qui peuvent être placées à droite du signe d'affectation, en réalité toutes les expressions, sont appelées valeurs droites. Les valeurs gauches sont aussi des valeurs droites.

Considérons, par exemple, l'expression `x = 2 * x`. La première occurrence de `x` est une valeur gauche et la seconde est une valeur droite. Ces deux occurrences de `x` ne sont pas évaluées de la même façon. La première a pour valeur l'adresse de `x`, c'est-à-dire l'adresse de la case mémoire où sera rangée la valeur de l'expression placée à droite de l'opérateur `=`. La seconde a pour valeur la valeur de `x`.

Si exp_1 est une valeur gauche et exp_2 est une expression, alors :

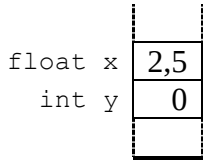
exp_1 `=` exp_2

est une expression qui a les propriétés suivantes :

- $type(exp_1 = exp_2) = type(exp_1)$
- $val(exp_1 = exp_2) = val(exp_2)$ convertie en $type(exp_1)$
- $adr(exp_1 = exp_2) = adr(exp_1)$,
- elle a un effet de bord : $val(exp_2)$ est enregistrée dans la case mémoire qui contient la valeur de exp_1 et remplace cette valeur.

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

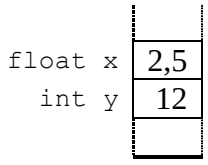
Supposons, par exemple, que la pile soit dans l'état :



L'évaluation de l'expression :

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

retourne la valeur 12 de type `int` et fait passer la pile dans le nouvel état :



Remarquons que l'expression à droite de l'opérateur `=` a la valeur 12,5 de type flottant mais qu'elle est convertie en la valeur 12 de type entier afin de pouvoir être affectée à `y` qui est de type entier.

Signalons quelques abréviations classiques, obtenues avec les opérateurs `++ -- += -= *= /= %=`.

Soit *exp* une expression numérique :

- l'expression :

  exp`++`

  a le même type que *exp*, le même effet de bord que *exp* `=` *exp* `+ 1` et a pour valeur $val(exp)$;

Opérateurs	Priorité	Associativité
`() [] . ->`	15	de gauche à droite
`! ++ -- -`*unaire* `*`*unaire* `(`*type*`) sizeof`	14	de droite à gauche
`*`*binaire* `/ %`	13	de gauche à droite
`+ -`*binaire*	12	de gauche à droite
`< <= > >=`	10	de gauche à droite
`== !=`	9	de gauche à droite
`&&`	5	de gauche à droite
`||`	4	de gauche à droite
`= += -= *= /= %=`	2	de droite à gauche

Figure 3.1. *Priorité et associativité des principaux opérateurs de C*

– l'expression :

  `++`*exp*

  a le même type que *exp*, le même effet de bord que *exp* `=` *exp* `+ 1` et a pour valeur val(*exp*) + 1.

Par exemple, si `x` a la valeur 5 :

– après l'évaluation de l'expression `y = x++`, `x` aura la valeur 6 et `y` aura la valeur 5 ;

– après l'évaluation de l'expression `y = ++x`, `x` aura la valeur 6 et `y` aura la valeur 6.

L'opérateur `--` a un comportement analogue.

Soit exp_1 une expression qui est une valeur gauche et exp_2 une expression, l'expression :

exp_1 *op*`=` exp_2

où *op* est l'un des opérateurs `+ - * / %` est équivalente à exp_1 `=` exp_1 *op* exp_2.

Par exemple, si `x` a la valeur 5, après l'évaluation de l'expression `x += 7`, `x` aura la valeur 12.

Ces opérateurs devront être utilisés avec précaution, c.-à-d. en étant bien conscient de leurs effets de bord. Par exemple, lorsque l'on écrit `y = ++x`, on n'affecte pas seulement la valeur de `x + 1` à `y` mais l'on modifie aussi la valeur de `x` en l'incrémentant de 1.

3.3 Priorité et associativité des opérateurs

La priorité et l'associativité des opérateurs sont utilisées lors de l'évaluation d'expressions dans lesquelles l'ordre des opérations n'a pas été explicitement indiqué par des paires de parenthèses.

Le tableau de la figure 3.1 indique la priorité et l'associativité des opérateurs qui ont été ou seront étudiés dans ce cours.

Les règles suivantes appliquées à une expression parcourue de gauche à droite, permettent de rétablir les paires de parenthèses rendues implicites par l'application des règles de priorité et d'associativité. Inversement, elles permettent de supprimer les paires de parenthèses rendues inutiles par l'application de ces mêmes règles.

Si *e*, *f* et *g* sont des expressions simples ou parenthésées et si op_1 et op_2 sont des opérateurs infixes (placés entre leurs opérandes), on a :

- *e* op_1 *f* op_2 *g* ≡ (*e* op_1 *f*) op_2 *g*, 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 de gauche à droite
- *e* op_1 *f* op_2 *g* ≡ *e* op_1 (*f* op_2 *g*), sinon

Si *e* et *f* sont des expressions simples ou parenthésées, si op_1 est un opérateur préfixe (placé devant son opérande) et si op_2 est un opérateur infixe, on a :

- op_1 *e* op_2 *e* ≡ (op_1 *e*) op_2 *f*, 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 de gauche à droite
- op_1 *e* op_2 *f* ≡ op_1 (*e* op_2 *f*), sinon

Par exemple :

- `6 * 3 + 4` ≡ `(6 * 3) + 4`, car la priorité de `*` est supérieure à celle de `+`
- `6 + 3 * 4` ≡ `6 + (3 * 4)`, car la priorité de `+` n'est pas supérieure à celle de `*`
- `!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 `||` n'est pas supérieure à celle de `>`
- `3 + 4 - 5` ≡ `(3 + 4) - 5`, car `+` et `-` ont la même priorité et `+` est associatif de gauche à droite
- `x = y = 5` ≡ `x = (y = 5)`, car `=` est associatif de droite à gauche
- la première expression de l'exemple 3.1 peut s'écrire de façon plus lisible de la façon suivante :

```
(mois == 12 && jour >= 22) || mois < 3 || (mois == 3 && jour < 20)
```

Un conseil cependant : en cas de doute, il est plus sûr de spécifier l'ordre des opérations d'une expression en la parenthésant.

3.4 Ordre d'évaluation des opérandes

L'ordre d'évaluation des opérandes n'est pas spécifié par la norme ANSI de C, excepté pour les opérateurs `||` et `&&`. Il faut en tenir compte lorsque l'un des opérandes est dépendant de l'autre, ce qui peut se produire en utilisant des opérateurs à effet de bord tels que `=`, `++` ou `+=`.

Par exemple, à la question : « Quelle est la valeur de l'expression `x != x++` ? », on est fortement tenté de répondre 1 (vrai) en pensant que quelque soit *x*, la valeur de *x* + 1 est toujours différente de celle de *x*. Or, cette expression pourra avoir la valeur 1 (vrai) ou la valeur 0 (faux) selon que l'opérande gauche (`x`) est évalué avant ou après l'opérande droit (`x++`). Supposons, par exemple, que `x` ait la valeur 5. Voyons tout d'abord le premier cas. On évalue `x` et l'on trouve 5. Puis on évalue `x++` et l'on trouve 5 aussi. La valeur de l'expression `x != x++` est donc 0 car 5 n'est pas différent de 5. Voyons maintenant le second cas. On évalue `x++` et l'on trouve 5. Puis on évalue `x` et l'on trouve 6, car `x` a été incrémenté de 1 lors de l'évaluation de `x++`. La valeur de l'expression `x != x++` est donc 1 car 5 est différent de 6.

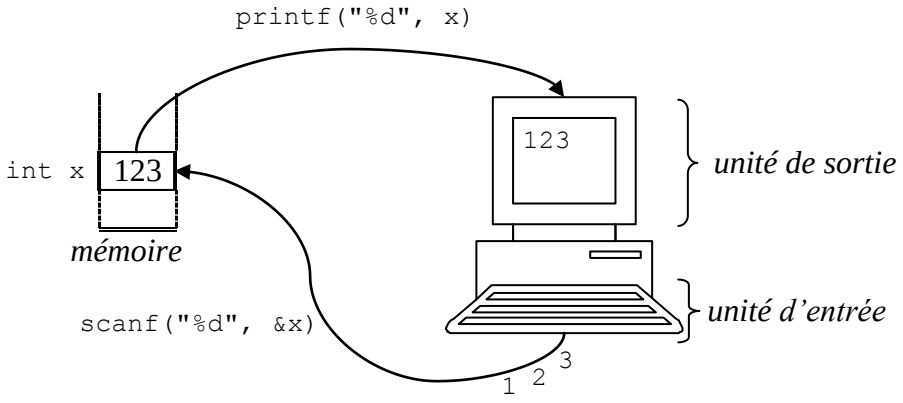


Figure 3.2. *Saisie et affichage*

3.5 Saisie de données et affichage de résultats

Les fonctions d'entrées-sorties seront étudiées au chapitre 6. En attendant, et afin de pouvoir écrire des programmes qui lisent des données saisies au clavier ou affichent des résultats à l'écran, on fera appel aux fonctions `printf` et `scanf` définies dans la bibliothèque standard de C, fournie avec toute implantation de ce langage.

Comme le montre la figure 3.2, la fonction `printf` écrit des caractères sur l'unité de sortie standard, qui par défaut, est l'écran du poste de travail. La fonction `scanf` lit des caractères sur l'unité d'entrée standard qui par défaut est le clavier du poste de travail. Les caractères lus sont donc ceux saisis au clavier par l'utilisateur.

Pour que ces fonctions soient reconnues, il faut que la directive :

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

soit placée en tête du programme.

L'affichage des valeurs d'une suite d'expressions e_1, ..., e_n ($n \geq 0$) au sein d'une chaîne de caractères qui les nomme ou les commente, est obtenue par l'appel :

`printf("...%`*conv*$_1$`...%`*conv*$_n$`...", `e_1`, ..., `e_n`)`

Les valeurs des expressions e_1, ..., e_n sont converties en une suite de caractères selon les spécifications de conversion `%`*conv*$_1$, ..., `%`*conv*$_n$ puis substituées à ces spécifications dans la chaîne de caractères (le format) `"...%`*conv*$_1$`...%`*conv*$_n$`..."`. La chaîne de caractères obtenue est affichée.

La lecture de la valeur d'une variable *var* est obtenue par l'appel :

`scanf("%`*conv*`", &`*var*`)`

Les caractères saisis par l'utilisateur sont lus après que l'utilisateur ait tapé sur la touche « Entrée », ils sont ensuite convertis en un nombre conformément à la spécification de conversion `%`*conv*. Ce nombre est affecté à la variable *var* dont l'adresse est calculée par l'opérateur `&` qu'il ne faudra surtout pas oublier.

Trois spécifications de conversion seront utilisées : `%c` pour convertir un caractère en un nombre de type `char` ou inversement, `%d` pour convertir une suite de caractères en un nombre

de type `int` ou inversement et `%f` pour convertir une suite de caractères en un nombre de type `float` ou inversement.

Par exemple :

– l'évaluation de l'appel `printf("%d + %d = %d\n", x, y, x + y)` aura pour effet de bord d'afficher « 8 + 4 = 12 » puis passera à la ligne, si la valeur de la variable `x` est 8 et celle de la variable `y` est 4 ;
– l'évaluation de l'appel `printf("Entrer un prix ? ")` aura pour effet de bord d'afficher « Entrer un prix ? » ;
– l'évaluation de l'appel `scanf("%f", &prix)` aura pour effet de bord de lire le nombre flottant saisi au clavier et de l'affecter à la variable `prix`.

Exercices

Exercice 3.1.

Soit `n` et `r` deux variables de type `int`.

1. Ecrire une expression qui a la valeur 1 si la valeur de `n` est paire et 0 sinon.
2. Ecrire une expression qui a pour effet d'affecter à la variable `r` la valeur 1 si la valeur de `n` est paire et 0 sinon.

Exercice 3.2. On suppose que les variables `c`, `a`, `b`, `n`, `x` et `y` sont définies de la façon suivante :

```
char c = 'Z';
int a = 1, b = 0, n = 12;
float x = 1.75;
float y;
```

1. Placer des parenthèses dans les expressions suivantes pour mettre en évidence l'ordre dans lequel les opérations sont réalisées :

```
2 * x + n
x >= 12 && x < 14
a || b && c
!a && b
(int) x + n
(float) n + x
y = x = 3.14
```

2. Donner le type, la valeur, et l'effet de bord de chacune des expressions suivantes :

```
c + 32
2 * x + n
x >= 12 && x < 14
a || (b && c)
!a && b
(int) x + n
(float) n + x
```

```
y = x = 3.14
n++
```

Exercice 3.3.

1. Définir cinq variables `a`, `b`, `r1`, `r2` et `r3` de type `int`.
2. Ecrire les expressions qui affectent respectivement à `r1`, `r2` et `r3` :
   - la valeur de « `a` implique `b` » ;
   - la valeur de « `a` équivalent à `b` » ;
   - la valeur de « `a` ou exclusif `b` ».

Exercice 3.4. On choisit de coder une date par un entier composé de la façon suivante :

$a \times 10000 + m \times 100 + j$

où a est l'année ($a \geq 1$), m le mois ($1 \leq m \leq 12$) et j le jour ($1 \leq j \leq 31$). Par exemple, la date 14 juillet 1789, sera codée par le nombre 17890714.

Il s'agit de terminer l'écriture d'un programme qui lit une date codée saisie au clavier (dont on supposera qu'elle est valide), la décode puis affiche le numéro du jour, le numéro du mois et l'année. La date codée sera affectée à une variable `d`. Le jour, le mois et l'année seront affectées aux variables `j`, `m` et `a` de type `int`.

Le programme est le suivant :

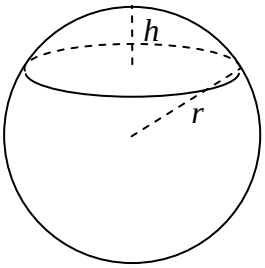
```
#include <stdio.h>

int main(void)
   {
   définition de j, m et a;
   définition de d;
   printf("Entrez une date codée ? ");
   scanf("%d", &d);
   calcul du numéro du jour et affectation du résultat à la variable j;
   calcul du numéro du mois et affectation du résultat à la variable m;
   calcul de l'année et affectation du résultat à la variable a;
   printf("jour = %d mois = %d annee = %d\n", j, m, a);
   return 0;
   }
```

1. Terminer l'écriture du programme. Le type de la variable `d` sera choisi en supposant que les extensions des types sont les extensions minimales fixées par la norme ANSI, indiquées dans le tableau de la figure 2.1 du chapitre 2.
2. Tester ce programme en décodant des dates variées.

Exercice 3.5. Il s'agit de terminer l'écriture d'un programme qui calcule puis affiche l'aire et le volume de la calotte polaire de la Terre (celle comprise entre le plan du cercle polaire arctique (antarctique) et le pôle N (S) géographique.

Le volume et l'aire d'une calotte sphérique peuvent être calculés en utilisant les formules de la figure 3.3. Le rayon de la Terre est de 6378 km et la distance entre un pôle et le plan du cercle polaire du même hémisphère est de 526 km environ. L'aire et le volume calculés seront affichés en nombres entiers de km^2.

Le programme est le suivant :



$$volume = \frac{1}{3}\pi h^2(3r - h)$$
$$aire = 2\pi rh$$

Figure 3.3. *Calotte sphérique*

```
#include <stdio.h>

int main(void)
   {
   définition d'une variable pi de type double à valeur constante égale à 3,14;
   définition de 2 variables h, r de type int;
   définition de 2 variables aire et volume de type float;
   affectation de la valeur de h;
   affectation de la valeur de r;
   calcul de l'aire de la calotte polaire et affectation du résultat à la variable aire;
   calcul du volume de la calotte polaire et affectation du résultat à la variable volume;
   printf("aire = %.0f, volume = %.0f\n", aire, volume);
   return 0;
   }
```

La spécification de conversion `%.0f` permet de ne pas afficher les chiffres après le point décimal et d'arrondir à l'entier inférieur ou supérieur selon que la partie décimale est ou non supérieure à 0,5. Cette spécification a été choisie car le rayon et la hauteur étant des valeurs approximatives données en nombre entier de kilomètres, on veut obtenir l'aire en nombre entier de km^2 et le volume en nombre entier de km^3.

1. Terminer l'écriture du programme et l'exécuter.
2. Expliquer pourquoi les variables `aire` et `volume` on été déclarées de type `double` plutôt que de type `float`. Pour cela, on se reportera au tableau des propriétés des types numériques présenté sur la figure 2.1 du chapitre 2.

4
Instructions

Une instruction est un ordre donné à l'ordinateur de réaliser une suite d'actions dont chacune a pour effet de changer l'état de la mémoire ou le déroulement du programme ou bien de communiquer avec les unités périphériques (clavier, écran, imprimante...).

En programmation impérative, un algorithme se traduit par une suite d'instructions. Par exemple, le calcul de x^n (où x est un nombre réel et n est un nombre entier positif ou nul) peut être réalisé par une suite de n multiplications en exécutant les instructions suivantes d'un langage de programmation virtuel :

soit x et p des variables réelles et n une variable entière
lire la valeur de x
lire la valeur de n qui devra être positive ou nulle
affecter la valeur de 1 à p
pour i = 1 à n **faire**
  affecter la valeur de $p \times x$ à p
fait
écrire la valeur de p

On lit les valeurs de x et de n. On initialise p à 1. On répète (valeur de n) fois l'instruction « **affecter** la valeur de $p \times x$ à p ». La valeur de p est alors égale à (valeur de x)$^{(\text{valeur de } n)}$. On écrit cette valeur.

Les instructions offertes par C sont les suivantes :

- instruction vide ;
- instruction « expression » ;
- bloc d'instructions ;
- instruction conditionnelle ;
- instruction de choix multiple ;
- instruction d'itération ;
- instruction de rupture de séquence ;
- instruction de retour d'un appel de fonction.

La suite de ce chapitre est consacrée à l'étude de ces instructions. Pour chaque instruction, nous décrirons sa syntaxe et l'action qu'elle réalise.

La façon de présenter une instruction dans le texte d'un programme n'est pas imposée. Une instruction peut être écrite indifféremment sur une ligne ou plusieurs lignes et indentée à l'aide de tabulations ou d'espaces. Cependant, pour faciliter la lecture d'un programme, il est de bonne pratique d'adopter pour chaque type d'instruction, une présentation qui mette en évidence ses différentes composantes. Prenons, par exemple, l'instruction conditionnelle :

`if (`*condition*`)` *instruction*$_1$ `else` *instruction*$_2$

qui exécute l'instruction 1, si la condition est vraie et l'instruction 2, si la condition est fausse. Les instructions 1 et 2 pouvant être complexes (composées d'autres instructions), il est classique de présenter l'instruction conditionnelle de la façon suivante, qui met en évidence ses trois composantes, la condition et les deux alternatives :

```
if (condition)
    instruction1
else
    instruction2
```

Nous proposerons donc, pour chaque instruction, en même temps que sa syntaxe, une façon de la présenter. Cette présentation sera utilisée dans tous les exemples de ce cours.

4.1 Instruction vide

Une instruction vide a la forme suivante :

```
;
```

Elle ne réalise aucune action.

4.2 Instruction « expression »

Une instruction « expression » a la forme suivante :

```
exp;
```

Elle réalise l'effet de bord de l'expression *exp*. Par exemple :

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

est une instruction qui déclenche l'action : « affecter 16 à x ». Par contre, l'instruction :

```
12 + 4;
```

ne déclenche aucune action. Elle est équivalente à l'instruction vide.

4.3 Bloc d'instructions

Un bloc d'instructions ou plus simplement un bloc regroupe une suite d'instructions au sein d'une instruction unique.

Un bloc est composé d'une suite éventuellement vide de déclarations suivie d'une suite éventuellement vide d'instructions. Un bloc a la forme suivante :

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

où $decl_1$, …, $decl_m$ sont des déclarations de type, de variables ou de tableaux et $inst_1$, …, $inst_n$ sont des instructions.

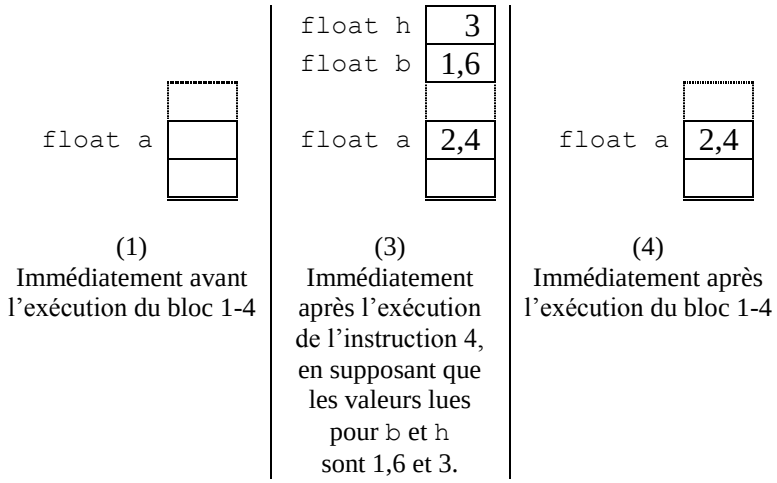


Figure 4.1. *Exécution du bloc d'instructions de l'exemple 4.1*

Les blocs peuvent être imbriqués. Cette situation se rencontre dans les blocs qui contiennent des instructions conditionnelles, de choix multiple ou d'itération, car ces instructions sont elles-mêmes composées d'instructions qui peuvent contenir des blocs.

Parmi les déclarations d'un bloc, les déclarations de variables qui ne sont pas précédées du mot-clé `extern` sont des définitions de variables. Les variables qui sont définies dans un bloc sont dites locales à ce bloc. Elles sont créées sur la pile au début de l'exécution de ce bloc et ôtées de la pile à la fin de son exécution. La durée de vie d'une variable locale à un bloc est donc celle de l'exécution de ce bloc.

Les variables qui sont utilisées dans un bloc et qui n'y sont pas définies doivent avoir été définies dans une autre partie du programme, soit globalement à l'extérieur des fonctions du programme, soit dans un bloc englobant. Nous expliquerons cela de façon plus détaillée au chapitre 5.

En termes de notre modèle de mémoire d'un programme (voir paragraphe 2.7), l'exécution d'un bloc se déroule de la façon suivante :

1. Les variables locales sont empilées sur la pile dans l'ordre de leur définition.
2. Les instructions sont exécutées séquentiellement jusqu'à la sortie du bloc qui est déclenchée :
   - soit parce que la fin du bloc a été atteinte ;
   - soit par l'exécution d'une instruction `break` ou `return`.
3. A la sortie du bloc, le sommet de la pile est ramené à la position qu'il occupait immédiatement avant l'exécution du bloc.

Exemple 4.1. L'instruction suivante est un bloc :

```
(1) {
(2) float b, h;
(3) scanf("%f", b);
(4) scanf("%f", h);
(5) a = h * b / 2;
(6) }
```

Dans ce bloc, les variables `b` et `h` sont des variables locales et la variable `a` est une variable non locale dont nous supposerons qu'elle est de type `float`. L'évolution de la pile lors de l'exécution de ce bloc est représentée sur la figure 4.1. □

Une entité (type, constante ou variable) déclarée et donc nommée dans un bloc *B* est visible (c.-à-d. peut être utilisée) dans *B* et dans tout bloc englobé par *B* qui ne déclare une entité de même nom.

La valeur initiale d'une variable dans un bloc est calculée lors de l'exécution de ce bloc. L'expression de cette valeur initiale peut donc comporter des noms de constantes ou de variables visibles dans ce bloc.

4.4 Instruction conditionnelle

Une instruction conditionnelle permet de choisir l'instruction à exécuter parmi deux possibles, en fonction de la valeur d'une expression booléenne : la condition. Si la condition est vraie alors, c'est la première instruction qui est exécutée, sinon, c'est la seconde. Par exemple : « si *p* est âgé de 7 à 77 ans alors afficher "*p* peut lire Tintin !" sinon afficher "*p* ne peut pas lire Tintin !" ». Le cas sinon peut être omis, lorsqu'il n'y a aucune instruction à exécuter si la condition n'est pas vérifiée.

En C, l'instruction conditionnelle est l'instruction `if`. Elle a l'une des deux formes suivantes :

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

ou

```
if (exp)
   inst
```

où *inst*, $inst_1$ et $inst_2$ sont des instructions. La première réalise l'action :

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

et la seconde l'action :

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

Par exemple, l'instruction :

```
if (x > y)
   max = x;
else
   max = y;
```

affecte à `max` la plus grande des valeurs de `x` et `y`. Le même effet aurait pu être obtenu de la façon suivante :

```
max = y;
if (x > y)
   max = x;
```

Les instructions `if` peuvent être imbriquées. En ce cas, 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 et non celle du `if` externe. Si c'est l'inverse qui est souhaité, il faut insérer le if interne dans un bloc :

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

Et d'ailleurs, dans le premier cas également, il est conseillé d'insérer le `if` interne dans un bloc :

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

Exemple 4.2. En supposant que la valeur de la variable `mois` est le rang (1 à 12) d'un mois de l'année et que la valeur de la variable `jour` est le rang (1 à 31) d'un jour d'un mois, l'instruction conditionnelle suivante affichera si ce jour là, en 2006, c'était l'hiver, le printemps, l'été ou l'automne.

```
if (mois < 3 || (mois == 3 && jour < 22))
   printf("C'etait l'hiver !");
else
   if (mois < 6 || (mois == 6 && jour < 20))
      printf("C'etait le printemps !");
   else
      if (mois < 9 || (mois == 9 && jour < 21))
         printf("C'etait l'ete !");
      else
         if (mois < 12 || (mois == 12 && jour < 23))
            printf("C'etait l'automne !");
         else
            printf("C'etait l'hiver !");
```

Par exemple, si `mois` a la valeur 8 et `jour` la valeur 5, l'exécution de cette instruction affichera « `C'etait l'ete !` ». □

4.5 Instruction de choix multiple

Une instruction de choix multiple permet de choisir une suite d'instructions à réaliser parmi un ensemble de suites d'instructions possibles, en fonction de la valeur d'une expression que nous appellerons « expression de choix ».

En C, l'instruction de choix multiple est l'instruction `switch` qui a la forme suivante :

`switch (`*exp-choix*`)`
`{`
…
inst-switch$_1$
…
inst-switch$_n$
`}`

où :

– *exp-choix* est une expression de type entier ;
– le bloc constitue le corps de l'instruction ;
– chaque instruction *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 entière 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
  exécuter l'instruction qui suit immédiatement cette étiquette puis les suivantes
sinon
  si dans le corps il existe une étiquette `default:` alors
    exécuter l'instruction qui suit immédiatement cette étiquette puis les suivantes
  sinon
    ne rien faire
  fin si
fin si

Pour éviter d'enchaîner l'exécution de deux cas exclusifs consécutifs, il faut terminer le premier par une instruction `break;` qui a pour effet d'abandonner l'exécution de l'instruction `switch` et d'exécuter l'instruction qui la suit immédiatement.

Exemple 4.3. En supposant que `n` ait pour valeur un nombre entier, le bloc suivant affiche si ce nombre est pair ou impair :

```
switch(n % 2)
   {
   case 0:
      printf("%d est pair !", n);
      break;
```

```
    case 1:
       printf("%d est impair !", n);
       break;
    }
```

Par exemple, si la valeur de `n` est 5, « `5 est impair !` » sera affiché.

□

Il est possible de définir des cas à plusieurs étiquettes.

Exemple 4.4. L'instruction suivante affichera ce que l'on fait le jour de la semaine dont le numéro (1 à 7) est affecté à la variable `jour` :

```
switch (jour)
   {
   case 1:
   case 2:
   case 4:
   case 5:
      printf("On travaille !");
      break;
   case 3:
      printf("On s'occupe des enfants !");
      break;
   case 6:
   case 7:
      printf("On se repose !");
      break;
   }
```

L'instruction `switch` peut être simplifiée en utilisant le cas `default` pour les jours où l'on travaille. Ce qui donne :

```
switch (jour)
   {
   case 3:
      printf("On s'occupe des enfants !");
      break;
   case 6:
   case 7:
      printf("On se repose !");
      break;
   default:
      printf("On travaille !");
      break;
   }
```

□

Attention ! Dans une expression `switch`, un cas doit être associé à chacune des valeurs que peut prendre l'expression de choix. Lorsque parmi ces valeurs, il en existe qui ne correspondent pas à une opération valide, alors, on leur associera un cas de traitement d'erreur : le cas `default` en général. □

Exemple 4.5. Le bloc suivant calcule puis affiche l'aire d'un carré ou d'un disque selon le choix de l'utilisateur.

```
(1)  {
(2)  char type;
(3)  float aire;
(4)  printf("Saisir c pour carre, d pour disque ? ");
(5)  scanf("%c", &type);
(6)  switch(type)
(7)     {
(8)     case 'c':
(9)        {
(10)       float c;
(11)       printf("Saisir le cote ? ");
(12)       scanf("%f", &c);
(13)       aire = c * c;
(14)       }
(15)       break;
(16)    case 'd':
(17)       {
(18)       float r;
(19)       printf("Saisir le rayon ? ");
(20)       scanf("%f", &r);
(21)       aire = 3.14 * r * r;
(22)       }
(23)       break;
(24)    default:
(25)       printf("Saisie incorrecte !");
(26)       exit(1);
(27)    }
(28) printf("aire = %f", aire);
(29) }
```

Si l'utilisateur saisit le caractère « c », c'est l'aire d'un carré dont la longueur du côté lui sera demandée, qui sera calculée et affichée. S'il saisit le caractère « d » c'est l'aire d'un disque dont la longueur du rayon, lui sera demandé qui sera calculée et affichée. S'il saisit un autre caractère, un message d'erreur sera affiché, le programme s'arrêtera et retournera la valeur 1 (erreur) au système d'exploitation. Cet arrêt est provoqué par l'appel `exit(1)` (ligne 26) à la fonction `exit` définie dans la bibliothèque standard.

Cet exemple illustre de plus l'imbrication des blocs ainsi que la visibilité des variables locales. Les variables `type` et `aire` définies dans le bloc externe 1-29 sont visibles dans ce bloc et dans les blocs 7-27, 9-14, 17-22 qu'il englobe. La variable `c` définie dans le bloc 9-14 n'est visible que dans ce bloc. La variable `r` définie dans le bloc 17-22 n'est visible que dans ce bloc.

L'évolution de la pile lors de l'exécution du bloc 1-29 est montrée sur la figure 4.2. On constate que la place occupée dans la pile par le bloc 9-14 a été libérée après la sortie de ce bloc et que le bloc 17-22 qui n'a pas été exécuté, n'a pas occupé de place dans cette pile. □

4.6 Instructions d'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. Une instruction d'itération est aussi appelée une boucle, car lors de son exécution, le programme exécute plusieurs fois la même instruction. Trois instructions d'itération sont disponibles en C : `while`, `do while` et `for`.

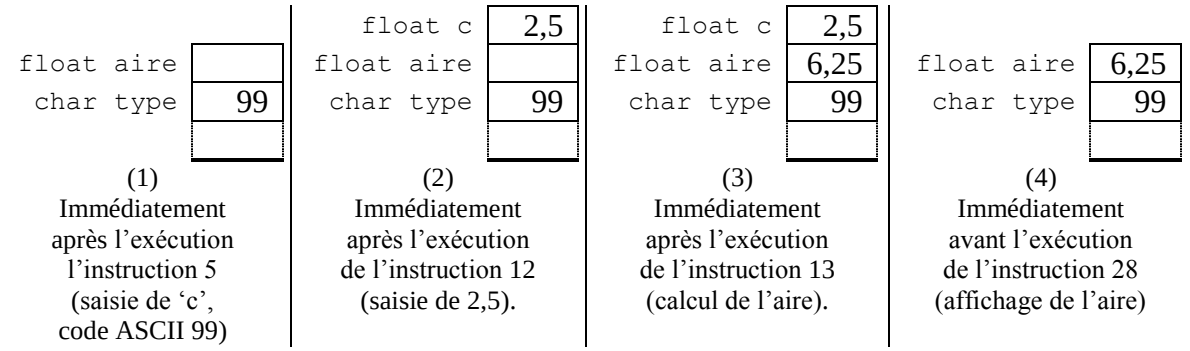


Figure 4.2. *Exécution du bloc de l'exemple 4.5*

4.6.1 **while**

L'instruction d'itération `while` a la forme suivante :

```
while (exp)
   inst
```

où *exp* est une expression booléenne : le test de continuation et *inst* est une instruction à répéter : le corps de la boucle.

L'action réalisée par cette instruction est la suivante :

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

Le corps de la boucle n'est jamais exécuté si le test de continuation est faux avant le premier passage dans la boucle.

Exemple 4.6. Le bloc suivant calcule le plus petit carré supérieur à un nombre entier saisi par l'utilisateur. Le résultat est affiché sous la forme : « Le plus petit carré supérieur à ? est ? ».

```
{
int n, i;
printf("Entrer un nombre entier ? ");
scanf("%d", &n);
i = 0;
while (i * i <= n)
   i = i + 1;
printf("Le plus petit carre superieur a %d est %d.", n, i * i);
}
```

Le nombre saisi est affecté à `n`. La variable `i` est initialisée à 0 (le nombre dont le carré est le plus petit possible). Tant que la valeur de `i * i` (c.-à-d. le carré de la valeur de `i`) est inférieure ou égale à celle de `n`, on incrémente `i de 1`. A la sortie de la boucle, la valeur de `i * i` est le carré recherché.

Par exemple, si le nombre entré est 50, l'exécution du bloc produira l'affichage :

```
Le plus petit carre superieur a 50 est 64 !
```

□

4.6.2 *do while*

L'instruction d'itération `do while` a la forme suivante :

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

où, 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*) ≠ 0 alors
aller à boucle
fin-si

Le corps de la boucle est exécuté au moins une fois, contrairement à l'instruction `while` où il peut ne jamais l'être.

Exemple 4.7. Le bloc suivant affiche le nombre de chiffres d'un nombre entier saisi par l'utilisateur :

```
{
int n, nb_chiffres;
printf("Entrer un nombre entier ? ");
scanf("%d", &n);
nb_chiffres = 0;
do
    {
    n = n / 10;
    nb_chiffres = nb_chiffres + 1;
    }
while (n != 0);
printf("Nombre de chiffres = %d", nb_chiffres);
}
```

L'algorithme utilisé est basé sur la règle suivante : « le nombre de chiffres d'un nombre entier *n* est égal au nombre de divisions réalisées au cours de l'opération suivante : diviser *n* par 10, si le quotient *q* est égal à 0 arrêter, sinon, répéter l'opération avec *n* = *q* ». Par exemple, si *n* = 172, on divise 172 par 10 et on obtient le quotient 17, puis on divise 17 par 10 et on obtient le quotient 1, puis on divise 1 par 10 et on obtient le quotient 0. Trois divisions ayant été réalisées, on en déduit que le nombre 172 a trois chiffres. Si *n* = 0, une division sera réalisée : le nombre 0 a bien un chiffre : le chiffre 0.

Le nombre dont on veut calculer le nombre de chiffres est lu et affecté à `n`. La variable `nb_chiffres` a pour valeur le nombre de divisions réalisées et donc le nombre de chiffres déjà comptés. Elle est initialisée à zéro. Le nombre de chiffres recherché est égal à la valeur de `nb_chiffres` à la sortie de la boucle : on l'affiche.

En initialisant `nb_chiffres` dans sa définition et en utilisant les opérateurs `/=`, `++` (voir paragraphe 3.3.7), ce bloc peut s'écrire de façon plus compacte :

```
{
int n, nb_chiffres = 0;
printf("Entrer un nombre entier ? ");
scanf("%d", &n);
```

```
do
    {
    n /= 10;
    nb_chiffres += 1;
    }
while (n != 0);
printf("Nombre de chiffres = %d", nb_chiffres);
}
```

Cette seconde écriture est-elle plus lisible ? Les experts de C la justifieront en disant que l'expression `nb_chiffres++` est une traduction plus fidèle de « incrémenter le nombre de chiffres » que l'expression `nb_chiffres = nb_chiffres + 1` et que l'expression `n /= 10` est une traduction plus fidèle de « changer la valeur de `n` en la divisant par 10 » que l'expression `n = n / 10`. Affaire de style donc !

Dans tous les cas, le choix de l'une ou de l'autre de ces deux écritures ne doit pas être guidé par des questions de performances car il faut faire confiance au compilateur pour générer un code exécutable optimal quelque soit l'écriture choisie. □

4.6.3 *for*

L'instruction d'itération `for` a la forme suivante :

`for (`exp_1`;` exp_2`;` exp_3`)`
    inst

où exp_1, exp_2, exp_3 sont des expressions et *inst* est une instruction. Elle est équivalente, par définition, à :

exp_1`;`
`while (`exp_2`)`
    `{`
    inst
    exp_3`;`
    `}`

Exemple 4.8. Le bloc suivant calcule puis affiche la somme des carrés des entiers de 1 à *n* où *n* est un entier > 0, saisi par l'utilisateur :

```
{
int n, i, s;
printf("Entrer un nombre entier > 0 ? ");
scanf("%d", &n);
if (n <= 0)
   {
   printf("Saisie incorrecte !");
   exit(1);
   }
s = 0;
for (i = 1; i <= n; i++)
   s = s + i * i;
printf("Somme des carres des entiers de 1 à %d = %d", n, s);
}
```

□

Dans une instruction `for (`exp_1`;` exp_2`;` exp_3`)`, 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 qui répète indéfiniment l'instruction *inst*. Elle utilisée lorsque l'arrêt de la boucle est provoquée par une instruction de rupture de séquence (`break`, par exemple), placée dans le corps de la boucle.

4.6.4 Choisir entre *for* et *while*

Puisque les instructions `for` et `while` sont équivalentes, on peut se demander quand choisir l'une et quand choisir l'autre. Ce choix peut être guidé par des considérations de lisibilité :

- Dans le cas où le nombre d'itérations dépend du calcul fait dans le corps de la boucle, on utilisera les instructions `while` ou `do while` plutôt que l'instruction `for`. C'est ce que nous avons fait dans les exemples 4.6 et 4.7.
- Dans le cas où le nombre d'itérations dépend d'un paramètre dont les valeurs initiale et finale ainsi que l'incrément sont connus avant l'exécution de la boucle, on utilisera plutôt l'instruction `for`. C'est ce que nous avons fait dans l'exemple 4.8.

Reprenons le calcul de la somme des carrés des entiers de 1 à *n* de l'exemple 4.8. En mathématiques, cette somme se noterait :

$$\sum_{i=1}^{n} i^2$$

On obtiendra une écriture en C plus proche de cette notation en utilisant l'instruction `for` :

```
s = 0;
for (i = 1; i <= n; i++)
   s += i * i;
```

plutôt que l'instruction `while` :

```
i = 1;
s = 0;
while (i <= n)
   {
   s += i * i;
   i++;
   }
```

4.7 Rupture de séquence

L'instruction :

```
break;
```

provoque l'abandon d'une instruction `while`, `do while`, `for` ou `switch` dans laquelle elle est insérée et l'exécution de l'instruction qui suit immédiatement l'instruction abandonnée. Nous avons montré son utilisation à l'intérieur d'une instruction `switch` au paragraphe 4.5.

4.8 Retour d'un appel de fonction

Les instructions :

```
return;
```

`return` *exp*`;`

où *exp* est une expression, sont destinées à être placées dans le corps d'une fonction. L'exécution de l'instruction `return;` par une fonction appelée redonne la main à la fonction appelante en lui retournant la valeur *val*(*exp*) si l'expression *exp* est spécifiée. Si la fonction appelée est la fonction `main`, l'exécution de l'instruction `return` provoque la fin du programme.

Nous étudierons cette instruction plus en détail au chapitre 6 consacré aux fonctions et aux programmes.

Exercices

Exercice 4.1. Soit le bloc suivant (un peu ésotérique !) :

```
(1)  {
(2)  int x = 4, y = 3, m;
(3)  if (calcul_flottant == 1)
(4)     {
(5)     float m;
(6)     m = (x + y) / 2.0;
(7)     printf("%f", m);
(8)     }
(9)  else
(10)    {
(11)    m = (x + y) / 2;
(12)    printf("%d", m);
(13)    }
(14) }
```

1. Quelles sont les variables locales aux blocs 1-14, 4-8 et 10-13.
2. A quelle définition se rapporte les variables `m`, `x` et `y` dans chacune des instructions 6, 7, 11 et 12.
3. En supposant que la variable `calcul_flottant` a la valeur 1, dessiner l'état de la pile immédiatement après l'exécution de l'instruction 6.

Exercice 4.2. Il s'agit de terminer l'écriture d'un programme qui fait avancer une horloge de 1 seconde. Ce programme est le suivant :

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

int main(void)
   {
   définitions de h, m et s
   printf("h (0 <= h < 24) ? ");
   scanf("%d", &h);
   if (h < 0 || h >= 24)
      {
      printf("Erreur de saisie !");
      exit(1);
      }
```

```
printf("m (0 <= m < 60) ? ");
scanf("%d", &m);
if (m < 0 || m >= 60)
   {
   printf("Erreur de saisie !");
   exit(1);
   }
printf("s (0 <= s < 60) ? ");
scanf("%d", &s);
if (s < 0 || s >= 60)
   {
   printf("Erreur de saisie !");
   exit(1);
   }
?
printf("%dh %dm %ds\n", h, m, s);
return 0;
}
```

L'heure qu'il est, est représentée par trois variables `h`, `m` et `s` de type `int`, où `h` est le nombre d'heures, `m` est le nombre de minutes et `s` est le nombre de secondes.

1. Terminer l'écriture de ce programme en remplaçant le **?** par une instruction `if` qui met l'heure à jour après qu'une seconde se soit écoulée. Par exemple, s'il est 15h 12m 59s on aura {`h` = 15, `m` = 12, `s` = 59} et une seconde plus tard {`h` = 15, `m` = 13, `s` = 0}. On supposera que le passage au jour suivant remet l'heure à {`h` = 0, `m` = 0, `s` = 0}.
2. Tester le programme pour des heures variées.

Exercice 4.3. Dans le programme suivant :

```
#include <stdio.h>

int main(void)
   {
   int i, s;
   ?
   printf("s = %d\n", s);
   return EXIT_SUCCESS
   }
```

remplacer le **?** par une instruction qui affecte à `s` la somme des entiers pairs de 2 à 100. Exprimer cette instruction de trois façons différentes en utilisant successivement les instructions `while`, `do while` et `for`

Exercice 4.4. Il s'agit de terminer l'écriture d'un programme qui compte le nombre de points d'une équipe à la suite d'une partie de belote à quatre. Les cartes seront décrites par deux caractères : le premier pour son type (Sept, Huit, Neuf, Dix, Valet, Dame, Roi, As) et le second pour son enseigne (Cœur, Carreau, Trèfle, Pique). Le type d'une carte est identifié par l'un des caractères suivants : 7 pour le Sept, 8 pour le Huit, 9 pour le Neuf, D pour le Dix, V pour le Valet, Q (Queen) pour la Dame, R pour le Roi et A pour l'As. L'enseigne d'une carte est décrite par l'un des caractères suivants : C pour un Cœur, D (Diamond) pour un Carreau, T pour un trèfle, et P pour un Pique.

On rappelle la valeur en nombre de points des cartes à la belote :

– à l'atout : Valet = 20, Neuf = 14, As = 11, Dix = 10, Roi = 4, Dame = 3, Huit = Sept = 0 ;
– hors atout : As = 11, Dix = 10, Roi = 4, Dame = 3, Valet = 2, Neuf = Huit = Sept = 0.

Il ne sera pas tenu compte des points supplémentaires tels que belote et rebelote, dix de der ou capot.

Ce programme est le suivant :

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

int main(void)
   {
   char atout, type, enseigne;
   int nb_cartes, points;
   Lecture de l'atout
   Lecture du nombre de cartes gagnées par l'équipe
   Calcul du nombre de points gagnés par l'équipe
   printf("Nombre de points = %d", points);
   return 0;
   }
```

Il demande quel est l'atout et le nombre de cartes gagnées par l'équipe, nombre qui doit être compris entre 0 et 32 et multiple de 4. Il calcule ensuite le nombre de points gagnés par l'équipe en ajoutant la valeur en nombre de points de chaque carte dont la description aura été demandée à l'utilisateur, puis affiche ce nombre de points.

La saisie de l'atout sera réalisée par les deux instructions suivantes :

```
printf("Atout (C|D|T|P) ? ");
scanf("%c", &atout);
```

La saisie du nombre de cartes sera réalisée par les deux instructions suivantes :

```
printf("Nombre de cartes ? ");
scanf("%d", &nb_cartes);
```

La saisie de la description d'une carte sera réalisée par les deux instructions suivantes :

```
printf("Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? ");
scanf(" %c%c", &type, &enseigne);
```

(l'espace en début du format de lecture a pour effet d'absorber le dernier caractère « nouvelle ligne » saisi par l'utilisateur, comme nous l'expliquerons au chapitre 6).

En cas de saisie incorrecte (type ou enseigne d'une carte inconnu, nombre de cartes non compris entre 0 et 32 et non multiple de 4), un message sera affiché et l'exécution sera interrompue par l'instruction `exit(1);`.

1. Compléter le programme en utilisant une instruction `for` pour balayer les cartes gagnées et une instruction `switch` pour calculer la valeur en nombre de points d'une carte.
2. Tester le programme pour différents ensembles de cartes gagnées et vérifier la détection des erreurs.

Voici, par exemple, une exécution du programme dans le cas où l'atout est Carreau et l'équipe a gagné 8 cartes : le Valet de Carreau, le Sept de Carreau, le Neuf de Carreau, le Huit de Carreau, l'As de Trèfle, le Neuf de Trèfle, le Roi de Trèfle et le Sept de Trèfle :

```
Atout (C|D|T|P) ? D
Nombre de cartes ? 8
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? VD
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? 7D
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? 9D
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? 8D
```

```
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? AT
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? 9T
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? RT
Carte (7|8|9|D|V|Q|R|A)(C|D|T|P) ? 7T
Nombre de points = 49
```

5
Fonctions et programmes

Tout langage de programmation offre le moyen de découper un programme en unités qui peuvent s'appeler les unes les autres en se transmettant des valeurs et qui peuvent partager les mêmes variables. En C, ces unités sont les fonctions et les variables partagées sont appelées variables globales, par opposition aux variables locales qui ne sont utilisables que dans les blocs où elles sont définies. Par convention, l'exécution d'un programme commence par l'appel de la fonction `main`, qui doit donc être présente dans tout programme C : c'est elle qui pilote l'exécution du programme.

Une fonction est définie par son nom, par le type des valeurs qu'elle reçoit de la fonction appelante, par le type de la valeur qu'elle lui retourne et par le bloc d'instructions qui permet de calculer cette valeur.

Les fonctions d'un programme C peuvent être réparties dans plusieurs fichiers. Chacun de ces fichiers peut être vu comme un module qui prend en charge une des composantes de l'application gérée par le programme ou bien comme une bibliothèque qui rassemble des fonctions d'intérêt général (des fonctions mathématiques, par exemple) qui pourront être utilisées soit par les autres modules du programme, soit par d'autres programmes. De plus, ces fichiers peuvent être compilés séparément, ce qui permet lors de la modification d'un programme de ne recompiler que les fichiers touchés par cette modification.

Dans la suite de ce chapitre, nous étudierons : la définition et l'appel des fonctions (paragraphe 5.1) ; les règles de structuration d'un programme composé de plusieurs fichiers compilés ou non (paragraphes 5.2 et 5.3) ; l'utilisation du compilateur GCC et de l'outil *make* pour assembler les différents fichiers d'un programme et générer son code exécutable (paragraphe 5.4). Nous terminerons ce chapitre par l'énoncé de quelques règles de bonne pratique pour l'écriture de programmes (paragraphe 5.5).

5.1 Fonctions

5.1.1 **Notion de fonction**

Rappelons qu'en mathématiques, une fonction f à n variables est une correspondance notée $f : E_1 \times \ldots \times E_n \rightarrow F$ qui à un élément $(x_1, ..., x_n)$ du produit cartésien des n ensembles $E_1, \ldots, E_n$ fait correspondre un et un seul élément de l'ensemble F noté $f(x_1, ..., x_n)$. $E_1 \times \ldots \times E_n$ est l'ensemble de départ de la fonction f, F est son ensemble d'arrivée et $x_1, ..., x_n$ sont ses variables.

Par exemple :

- la fonction à une variable *partie-entière* : $\mathbb{R} \rightarrow \mathbb{Z}$ qui à un nombre réel r fait correspondre sa partie entière *partie-entière*(r), qui est un entier relatif ;
- la fonction à deux variables *min* : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ qui à une paire de nombres entiers relatifs (z_1, z_2) fait correspondre le plus petit de ces deux nombres *min*(z_1, z_2) qui est un entier relatif.

Dans un langage de programmation, les ensembles mis en correspondance sont les types. Une fonction est définie par son nom, par son type de départ, par son type d'arrivée et par son corps : l'expression ou les instructions permettant de calculer l'élément de l'ensemble d'arrivée à partir d'un élément de l'ensemble de départ. On appelle application d'une fonction, l'opération qui réalise ce calcul.

En C, le corps d'une fonction $f : T_1 \times \ldots \times T_n \rightarrow T$ est un bloc dans lequel sont définies n variables locales de type respectifs T_1, ..., T_n, qui constituent les arguments formels de la fonction et qui comporte au moins une instruction qui retourne une valeur de type T. L'application de cette fonction est une expression $f(e_1, \ldots, e_n)$, dite appel de fonction, dans laquelle e_1, …, e_n sont des expressions de types respectifs T_1, ..., T_n dont les valeurs constituent les arguments effectifs de l'appel. La valeur de cet appel est celle qui est retournée par l'exécution du corps de la fonction, après avoir affecté les arguments effectifs aux arguments formels.

Exemple 5.1. La fonction *min* citée ci-dessus peut être définie en C de la façon suivante :

```
(1) int min(int z1, int z2)
(2)     {
(3)     if (z1 < z2)
(4)        return z1;
(5)     else
(6)        return z2;
(7)     }
```

La ligne 1 déclare une fonction qui retourne une valeur de type `int`, qui a pour nom `min` et dont les arguments formels sont `z1` de type `int` et `z2` de type `int`. Son corps est le bloc 2-7. L'instruction `return` *e;* retourne la valeur de l'expression *e*.

La valeur de l'appel `min(10 + 9, 5)` est 5. En effet, les arguments effectifs de cet appel sont 19 et 5. Ils sont affectés aux arguments formels `z1` et `z2`. Puis le corps est exécuté. Il retourne 5 puisque c'est l'instruction `return z1;` qui est exécutée et que `z1` a la valeur 5. □

5.1.2 **Définition d'une fonction**

La définition d'une fonction a la forme suivante :

T f(*decl*$_1$, …, *decl*$_n$)
bloc

où :

- la première ligne est l'en-tête de la fonction (on dit aussi son prototype) qui déclare le nom et le type de cette fonction ;
- *f* est le nom de la fonction : un identificateur ;
- les *decl*$_i$ (*i* de 1 à *n*) sont les déclarations des arguments formels de la fonction, c.-à-d. 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 le bloc d'instructions qui constitue le corps de la fonction.

Attention ! Il doit y avoir une déclaration par argument formel. Il n'est pas possible de regrouper dans la même déclaration plusieurs arguments formels. □

Les arguments formels sont des variables locales au corps de la fonction.

Dans le corps de la fonction on doit trouver au moins une instruction :

`return` *exp*`;`

dont l'exécution a pour effet de retourner à la fonction appelante la valeur *val*(*exp*) que nous appellerons valeur de retour.

Exemple 5.2. La fonction qui calcule x^n où x est un réel et n un entier positif ou nul, peut être définie de la façon suivante :

```
(1)  float puissance(float x, int n)
(2)     {
(3)     float p;
(4)     int i;
(5)     if (n < 0)
(6)        {
(7)        printf("Appel de puissance incorrect !");
(8)        exit(1);
(9)        }
(10)    p = 1;
(11)    for (i = 1; i <= n; i++)
(12)       p = p * x;
(13)    return p;
(14)    }
```

La ligne 1 constitue l'en-tête de la fonction et le bloc d'instructions 2-14, son corps. Cette fonction a pour nom `puissance`, pour arguments formels les variables `x` et `n` de types respectifs `float` et `int` et pour type d'arrivée `float`. L'algorithme utilisé a été expliqué dans l'introduction du chapitre 4. □

5.1.3 **Appel d'une fonction**

Un appel de fonction est une expression ayant la forme suivante :

f`(`*exp*$_1$`, …, `*exp*$_n$`)`

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

L'application d'une fonction est une expression au même titre que les expressions arithmétiques ou booléennes étudiées au chapitre 3.

Si *exp*$_1$, ..., *exp*$_1$, sont des expressions et que *f* est le nom d'une fonction de type $T_1 \times ... \times T_n \rightarrow T$, alors :

- *type*(*f*`(`*exp*$_1$`, . . . , `*exp*$_n$`)`) = *T*
- *val*(*f*`(`*exp*$_1$`, . . . , `*exp*$_n$`)`) = valeur retournée par l'exécution du corps de la fonction *f*

La valeur de l'appel *f*(*exp*$_1$, ..., *exp*$_n$) où *f* est le nom d'une fonction de type $T_1 \times ... \times T_n \rightarrow T$ dont les arguments formels sont les variables x_1, …, x_n et dont le corps est un bloc d'instructions *b*, a pour valeur la valeur de type *T* obtenue de la façon suivante :

1. Les valeurs des arguments effectifs sont calculées :

   $arg_1 = val(exp_1)$, …, $arg_n = val(exp_n)$
2. Les arguments formels x_1, …, x_n sont empilés. Ils sont considérés comme des variables locales au corps de la fonction.
3. Les instructions suivantes sont exécutées :

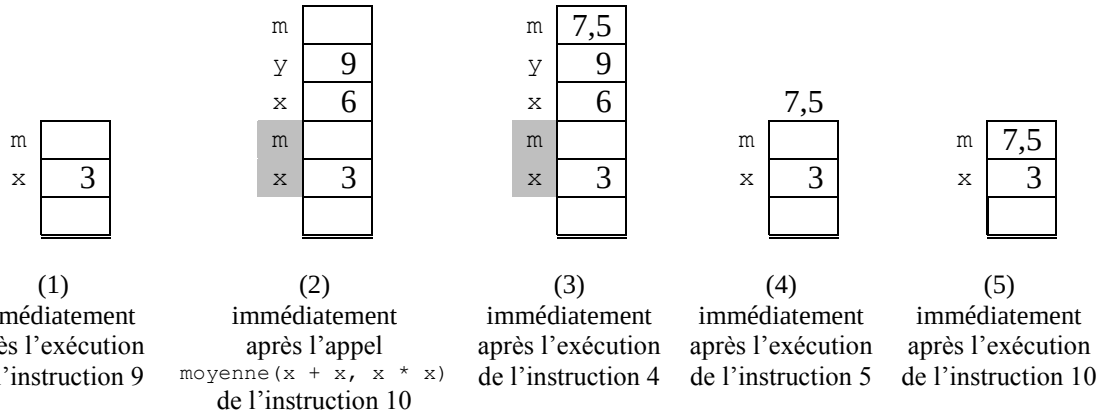


Figure 5.1. *Exécution du bloc de l'exemple 5.3*

x_1 = arg_1;
...
x_n = arg_n;

(si un argument effectif arg_i n'est pas de type T_i, il est donc converti en T_i si cette conversion est possible).

4. Le bloc *b* est exécuté retournant la valeur de l'appel $f(exp_1,\ldots, exp_n)$,
5. Les arguments formels sont dépilés.

Remarquons :

– que les valeurs des arguments sont recopiées dans les variables x_1, …, x_n. On dit que les arguments sont passés par valeur ;

– que les variables locales au bloc appelant ne sont pas visibles dans la fonction appelée puisque le bloc constituant le corps de la fonction appelante n'est pas imbriqué dans celui qui constitue le corps de la fonction appelée.

Exemple 5.3. Soit la fonction :

```
(1) float moyenne(float x, float y)
(2)     {
(3)     float m;
(4)     m = (x + y) / 2;
(5)     return m;
(6)     }
```

appelée dans le bloc :

```
(7) {
(8) float x, m;
(9) x = 3;
(10) m = moyenne(x + x, x * x);
(11) }
```

L'évolution de la pile lors de l'exécution de ce bloc est montrée sur la figure 5.1.

Remarquons notamment ce qui se passe lors de l'appel d'une fonction et lors du retour de cet appel. Lorsqu'une fonction est appelée dans un bloc, les valeurs des variables locales à ce bloc sont préservées dans la pile mais ces variables sont invisibles dans le bloc constituant le corps de la fonction appelée, puisque ce bloc n'est pas imbriqué dans le bloc appelant (états 2

et 3). Ces variables redeviennent visibles lors du retour dans le bloc appelant (état 4). Par contre, les variables locales à la fonction appelée sont ôtées de la pile puisqu'elles ne sont plus utiles (passage de l'état 3 à l'état 4). Seul est préservée la valeur de retour (état 4). □

5.1.4 **Fonctions sans arguments ou sans valeur de retour**

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

T f`(void)`

où `void` est le type vide. C'est le cas, par exemple, de la fonction suivante qui lit un entier dans le fichier d'entrée standard :

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

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

`void` *f*`(`*decl*$_1$`, …,` *decl*$_n$`)`

où `void` est le type vide. En ce cas, bien entendu, les instructions `return` présentes dans le corps de cette fonction ne spécifieront pas d'expression de la valeur de retour. Elles seront de la forme `return;`. Une fonction sans valeur de retour n'a de sens que si son appel produit un effet de bord. C'est le cas, par exemple, de la fonction suivante qui affiche un entier à l'écran :

```
void ecrire_entier(int i)
    {
    printf("%d", i);
    return;
    }
```

Lorsqu'une instruction `return;` est la dernière du corps d'une fonction, elle peut être omise. La définition de la fonction `ecrire_entier` aurait pu aussi s'écrire :

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

car la dernière instruction d'une fonction sans valeur de retour peut-être omise si c'est une instruction `return ;`.

Les deux fonctions précédentes peuvent être combinées pour écrire l'entier lu :

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

5.1.5 **Lancement et sortie d'un programme : les fonctions `main` et `exit`**

La fonction `main` est un peu particulière. Elle est appelée par le système d'exploitation pour lancer l'exécution du programme et elle lui retourne un code d'erreur (un entier) indiquant si le programme s'est ou non déroulé correctement. Elle ne peut être appelée ni par elle-même, ni dans le corps d'une autre fonction du programme.

La fonction `main` retourne un entier de type `int` et peut avoir des d'arguments qui lui sont transmis par le système d'exploitation lors du lancement du programme. Lorsqu'elle n'a pas d'arguments son en-tête doit être :

```
int main(void)
```

Lorsqu'elle a des arguments, son en-tête doit être :

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

Si le programme est lancé par la commande :

n a_1 ... a_k

où *n* est le nom de cette commande et a_1 ... a_k sont ses arguments la variable `argc` aura la valeur *n* + 1, la variable `argv[0]` pointera sur la chaîne de caractères *n*, la variable `argv[1]` sur la chaîne de caractères a_1 et la variable `argv[`*k*`]` sur la chaîne de caractères a_k. Tout cela sera plus clair lorsque nous aurons étudié les tableaux, les pointeurs et les chaînes de caractères (voir chapitres 7, 8 et 9).

L'entier retourné par la fonction `main` est, par convention, égal à 0 si le programme s'est déroulé correctement, différent de 0 sinon (1, en général). En conséquence, l'appel de la fonction `main` doit se terminer par l'exécution de l'instruction `return` *exp;* où *exp* est une expression qui a pour valeur le code d'erreur.

Il peut arriver que le déroulement d'un programme doive être interrompu à cause d'une situation exceptionnelle : données incorrectes, mémoire insuffisante, demande d'accès à une ressource inconnue, etc. Dans un tel cas, la sortie du programme peut être provoquée en utilisant la fonction `exit` d'en-tête :

```
void exit(int code)
```

déclarée dans le fichier d'en-têtes `stdlib.h`. L'appel `exit(`*code d'erreur*`)` a pour effet de sortir du programme et de rendre la main au système d'exploitation en lui transmettant le code d'erreur.

La signification du code d'erreur retourné par la fonction `main` ou la fonction `exit` dépend de l'implantation. Pour s'affranchir de cette dépendance, on pourra utiliser les constantes `EXIT_SUCCESS` (le programme s'est déroulé correctement) ou `EXIT_FAILURE` (une erreur s'est produite) définies dans le fichier d'en-têtes `stdlib.h`.

Exemple 5.4. Le programme ci-dessous qui sera plus facile à comprendre après avoir étudié les tableaux, les pointeurs et les chaînes de caractères (voir chapitre 7, 8 et 9) calcule puis affiche la somme des paramètres de la commande qui la lance :

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

int main(int argc, char *argv[])
   {
   int i, s = 0;
   if (argc == 0)
      {
      printf("La liste des arguments est vide !")
      exit(EXIT_FAILURE);
      }
   for (i = 1; i < argc; i++)
      s += atoi(argv[i]);
   printf("Somme des arguments = %d\n", s);
   return EXIT_SUCCESS;
   }
```

(la fonction `atoi` convertit la chaîne de caractères qu'elle reçoit en argument en un nombre entier : celui dont cette chaîne est la représentation littérale).

Supposons que ce programme soit enregistré dans le fichier `somme-arguments.c` par la commande :

```
gcc –Wall somme-arguments.c –o somme
```

l'exécution de la commande

```
somme 1 3 5 7 9
```

affichera :

```
Somme des parametres = 25
```

□

5.2 Structure d'un programme C

Le texte d'un programme C est contenu dans un ou plusieurs fichiers dits « fichiers sources ».

Pour les programmes simples, tels que ceux qui sont proposés dans ce cours à titre d'exercices, un seul fichier source est suffisant. Mais dès qu'un programme devient volumineux (un programme peut contenir des dizaines, voire des centaines de milliers d'instructions !), il faut le découper en plusieurs fichiers sources que l'on peut voir comme des modules. Cela a plusieurs avantages :

- **découpage fonctionnel d'un programme** en plusieurs modules ;
- **réutilisation et partage** : les fonctions utiles à plusieurs modules d'un même programme ou de programmes différents peuvent être définies une fois pour toutes et partagées par ces modules ;
- **facilité de maintenance**, car chaque module peut-être compilé séparément, ce qui permet, en cas de modifications du programme, que seuls les fichiers source touchés par ces modifications soient recompilés.

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

- des directives au préprocesseur ;
- des déclarations de types ;
- des déclarations externes de variables ou de fonctions ;
- des définitions de variables ;
- des définitions de fonctions.

Les fonctions définies dans un fichier source peuvent utiliser des variables globales ou des fonctions définies dans d'autres fichiers sources ainsi que des fonctions d'utilité générale dont le code objet est enregistré dans une bibliothèque dont notamment la bibliothèque standard livrée avec toute implantation de C. Nous la décrirons succinctement au paragraphe 5.4.

Les éléments d'un fichier source doivent être ordonnés conformément à des règles que nous expliquerons plus en détail dans la suite de ce chapitre, mais qui peuvent se résumer dans le principe suivant : chaque fois que le compilateur lit un nom dans le texte d'un fichier source, il doit avoir connaissance de l'entité (type, constante ou variable) désignée par ce nom et de son type. Cette entité doit donc avoir été déclarée. Lorsque les fonctions d'un fichier source utilisent des entités définies dans d'autres fichiers sources ou dans une bibliothèque, il est nécessaire de déclarer ces entités au début de ce fichier. La pratique est de regrouper les déclarations d'entités utilisables dans plusieurs fichiers sources dans des fichiers dits :

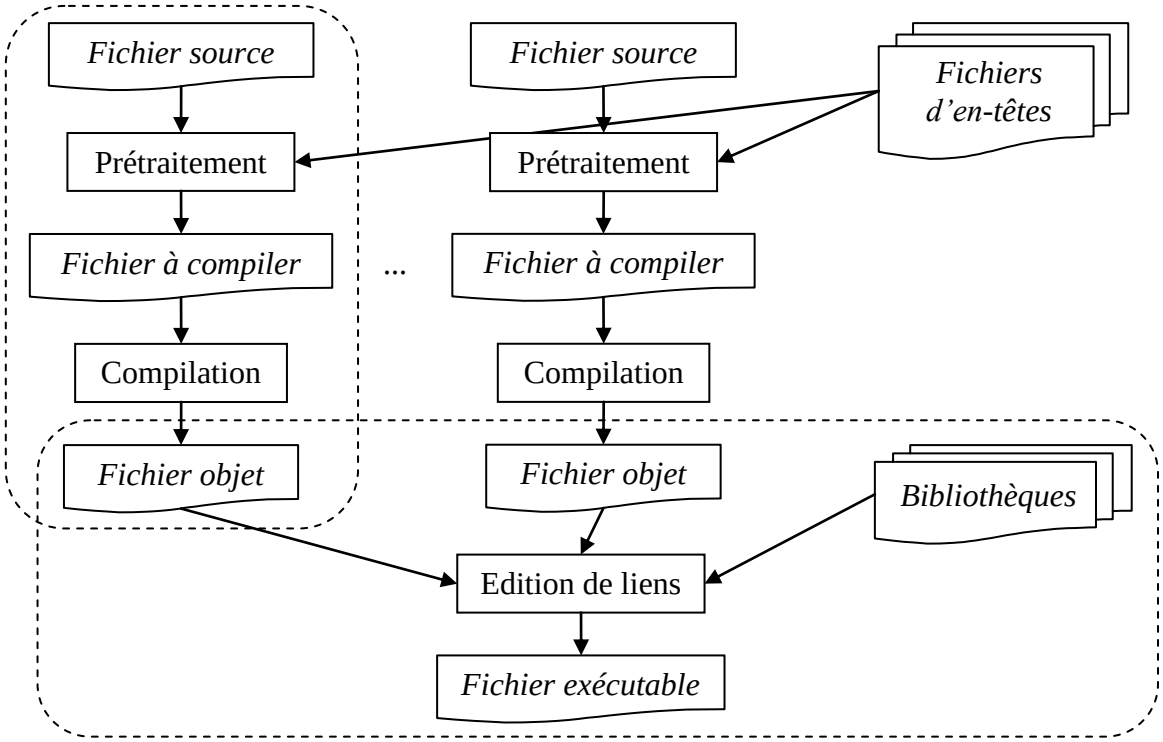


Figure 5.2. *Génération du code exécutable d'un programme*

« fichiers d'en-têtes ». Le contenu d'un fichier d'en-têtes devra être inclus dans tout fichier source utilisant une entité déclarée dans ce fichier d'en-têtes.

Les déclarations des fonctions de la bibliothèque standard sont regroupées dans un ensemble de fichiers d'en-têtes livrés avec chaque implantation de C. Par contre, c'est au programmeur de créer les fichiers d'en-têtes déclarant les entités définies dans ses propres programmes.

La génération du code exécutable d'un programme se déroule en deux phases comme le montre la figure 5.2.

1. **Prétraitement et compilation des fichiers sources.** Chaque fichier source est soumis au préprocesseur qui traite les directives qui lui sont adressées dont notamment l'inclusion des fichiers d'en-têtes et produit un fichier qui est soumis au compilateur. Le compilateur produit un « fichier objet » qui contient les codes exécutables des fonctions définies dans le fichier source ainsi que des appels vers des fonctions dont le code exécutable est contenu dans une bibliothèque mais dont l'adresse en mémoire n'est pas encore connue.
2. **Edition de liens.** L'éditeur de liens rassemble les codes exécutables de toutes les fonctions appelées par le programme, résout les appels laissés en suspens dans la phase de compilation et génère le code exécutable du programme qui est enregistré dans un fichier dit « fichier exécutable ». Le nom de ce fichier sert de commande pour lancer l'exécution du programme.

5.2.1 **Directives au préprocesseur**

Le préprocesseur est un outil bien pratique. Il offre au programmeur plus de souplesse pour écrire un fichier source que ne le permettrait la stricte observance de la syntaxe de C. Les trois principales actions du préprocesseur sont la substitution de macros, l'inclusion de fichiers et la compilation conditionnelle.

Une action du préprocesseur est commandée par une directive occupant une ligne du fichier source et qui doit commencer par le caractère `#`.

Le préprocesseur lit séquentiellement chaque ligne du fichier source. Si cette ligne commence par le caractère `#`, il exécute la directive qui suit ce caractère, sinon, il recopie la ligne lue dans le fichier à compiler.

Nous ne détaillerons dans ce paragraphe que la substitution de macros et l'inclusion de fichiers d'en-têtes qui sont utilisées dans la quasi-totalité des programmes C. Un exemple de compilation conditionnelle sera donné au paragraphe 5.4 dans le cas de l'écriture d'un fichier d'en-têtes.

Substitution de macros

Une macro est une association entre un nom (un identificateur) et un morceau de texte d'un programme C. Une définition de macro a la forme suivante:

`#define` *nom texte*

On adopte souvent la convention d'écrire les noms de macro en majuscules pour les distinguer des noms de variables ou de fonctions que l'on écrit en général en minuscules.

Après avoir lu cette directive, le préprocesseur substituera le texte *texte* à chaque occurrence de l'identificateur *nom*. Cette substitution ne s'applique pas à l'intérieur d'une chaîne de caractères, d'un commentaire ou d'un identificateur.

Exemple 5.5. Si le fichier source est :

```
#define PI 3.14

int main(void)
   {
   float r, p;
   r = 6325;
   p = 2 * PI * r;
   return 0;
   }
```

le préprocesseur produira le fichier à compiler :

```
int main(void)
   {
   float r, p;
   r = 6325;
   p = 2 * 3.14 * r;
   return 0;
   }
```

□

Attention ! Il ne faut pas confondre une définition de macro et une affectation de variable. Si par confusion avec une instruction d'affectation, on avait écrit :

```
#define PI = 3.14
```

au lieu de

```
#define PI 3.14
```

le préprocesseur aurait remplacé l'expression `2 * PI * r` par l'expression `2 * = 3.14 * r` qui est syntaxiquement incorrecte. □

L'utilisation la plus classique des macros est celle de l'exemple 5.5 : la définition de constantes symboliques. L'utilisation de constantes symboliques a deux avantages. Le premier est d'améliorer la lisibilité des programmes. Par exemple, si dans l'instruction `switch` de l'exemple 4.4 du chapitre 4, les numéros de jour de la semaine avaient été désignés par les constantes symboliques définies par :

```
#define LUNDI 1
#define MARDI 2
#define MERCREDI 3
#define JEUDI 4
#define VENDREDI 5
#define SAMEDI 6
#define DIMANCHE 7
```

cette instruction aurait pu s'écrire de façon plus lisible :

```
switch (jour)
   {
   case LUNDI:
   case MARDI:
   case JEUDI:
   case VENDREDI:
      printf("On travaille !");
      break;
   case MERCREDI:
      printf("On s'occupe des enfants !");
      break;
   case SAMEDI:
   case DIMANCHE:
      printf("On se repose !");
      break;
   }
```

Le second avantage est d'assurer une certaine forme de paramétrage. Par exemple, si un programme utilise plusieurs fois le nombre π, le fait de donner le nom `PI` à une valeur approchée de π, permet de changer cette valeur en un seul endroit du fichier source : la définition de la macro `PI`, au lieu d'avoir à changer chacune de ses occurrences.

Inclusion de fichiers d'en-têtes

Après avoir lu la directive :

`#include` *nom de fichier*

le préprocesseur écrit dans le fichier à compiler le texte contenu dans le fichier dont le nom est spécifié.

Cette directive est principalement utilisée pour inclure dans un fichier source le contenu d'un fichier d'en-têtes. Elle a alors l'une des deux formes suivantes :

`#include <`*nom module*`.h>`

`#include "[`*chemin d'accès/*`]`*nom module*`.h"`

La première forme est à utiliser pour l'inclusion d'un fichier qui appartient à un répertoire connu du préprocesseur. C'est le cas notamment des fichiers d'en-têtes associés à la bibliothèque standard. Si ce n'est pas le cas, il faut utiliser la seconde forme. Le nom de fichier fourni doit être un nom complet précisant le chemin d'accès sauf si ce fichier réside dans le répertoire courant, auquel cas son nom suffit.

5.2.2 **Règles de visibilité**

Les noms (identificateurs) apparaissant dans un fichier à compiler sont classés en trois catégories :

1. les noms de variables, de tableaux et de fonctions, les noms de valeurs introduites par un type énuméré et les noms de type introduits par une déclaration `typedef` ;
2. les noms d'énumération, de structure et d'union ;
3. les noms des champs de chaque type structure ou de chaque type union.

(Les tableaux, structures et unions seront étudiées au chapitre 7.)

La liaison entre un nom et l'entité qu'il désigne est faite au travers de la déclaration de cette entité. Par exemple, la déclaration :

```
int longueur;
```

lie le nom `longueur` à une variable de type `int`. Le compilateur lit les fichiers qui lui sont soumis séquentiellement. Chaque fois qu'il lit un nom, il doit avoir lu au préalable la déclaration de l'entité désignée par ce nom, afin de connaitre ses propriétés. Par exemple, si le compilateur lit le nom `longueur` dans l'instruction `longueur = 5;` il doit savoir si `longueur` est bien le nom d'une variable numérique et quel est son type, ce qui nécessite d'avoir lu la déclaration de cette variable.

Plusieurs entités différentes peuvent avoir le même nom. Cela ne pose pas de problème quand ces entités appartiennent à des catégories différentes, car elles sont syntaxiquement différentiables. Par exemple, la syntaxe de C est telle qu'un nom d'énumération ne peut pas être confondu avec un nom de variable. Mais quand deux entités de même nom appartiennent à la même catégorie, par exemple un nom de variable et un nom de fonction, il est nécessaire d'avoir des règles permettant d'associer sans ambiguïté un nom à sa déclaration. Ce sont les règles de visibilité. Ces règles sont telles, qu'un nom lu en un point d'un programme est lié à une et d'une seule entité. On dit alors que cette entité est visible en ce point.

En C les règles de visibilité sont les suivantes :

- Les entités déclarées globalement, c.-à-d. en dehors du corps des fonctions, 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 entité locale de même catégorie et de même nom.
- Les entités déclarées dans un bloc sont visibles dans ce bloc ainsi que dans tout bloc englobé ne déclarant pas une entité de même catégorie et de même nom. La déclaration d'une entité dans un bloc masque les entités de même catégorie et de même nom déclarées globalement ou dans un bloc englobant.
- Les arguments formels d'une fonction sont assimilés à des entités locales au bloc qui constitue le corps de cette fonction et sont donc considérés comme étant déclarés dans ce bloc.
- Il ne peut pas y avoir deux entités globales de même catégorie et de même nom, ni deux entités de même catégorie et de même nom déclarées dans un même bloc.

Ces règles sont un peu abstraites. Le fichier source suivant, dont c'est d'ailleurs le seul intérêt, en illustre certaines :

```
(1)  float x = 5.5, y = 9.1;

(2)  float min(float x, float y)
(3)     {
(4)     if (x < y)
(3)        return x;
(4)     else
(5)        return y;
(6)     }

(7)  int main(void)
(8)     {
(9)     float m;
(10)    m = min(x, y);
(11)    return 0;
(12)    }
```

Les variables `x` et `y` définies à la ligne 1 sont des variables globales. Elles sont visibles dans le corps de la fonction `main` (bloc 8-12) mais pas dans celui de la fonction `min` (bloc 2-6) car elles sont masquées par les arguments formels `x` et `y` qui sont assimilés à des variables locales au bloc 3-6. Ces deux variables sont visibles dans tout ce bloc. La variable `m` définie à la ligne 9 est locale au bloc 8-12, elle est visible dans tout ce bloc. Après l'exécution de l'instruction 10, la variable `m` aura la valeur 5,5.

5.2.3 **Déclarations externes**

Tout d'abord, il faut bien faire la différence entre déclaration et définition :

- la déclaration d'une variable, d'un tableau ou d'une fonction indique au compilateur le nom et le type de cette entité ;
- la définition d'une variable, d'un tableau ou d'une fonction déclare cette entité, la crée en lui affectant un emplacement en mémoire et l'initialise.

Par exemple, la définition :

```
int x = 2;
```

déclare une variable `x` de type `int` et crée cette variable en lui affectant une case mémoire et en y enregistrant la valeur 2 (valeur initiale) et la définition :

```
int est_impair(int x)
    {
    return x % 2;
    }
```

déclare une fonction `est_impair` de type `int` → `int` et crée cette fonction en enregistrant son code en mémoire.

La déclaration d'une variable, d'un tableau ou d'une fonction peut être séparée de sa définition. Une telle déclaration est appelée déclaration externe. Le rôle d'une déclaration externe est d'indiquer au compilateur le type d'une variable, d'un tableau ou d'une fonction dont il lira le nom avant de lire sa définition, soit parce que cette définition apparaît plus loin dans le même fichier source, soit parce qu'elle apparaît dans un autre fichier source.

La déclaration externe d'une variable ou d'un tableau est constituée du mot-clé `extern` suivi de la déclaration de cette variable ou de ce tableau à l'exception des valeurs initiales. Par exemple :

```
extern int nombre_maximum_places;
```

est la déclaration externe d'une variable de nom `nombre_maximum_places` et de type `int`.

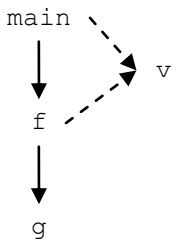
La déclaration externe d'une fonction est constituée facultativement du mot-clé `extern`, suivi de l'en-tête de cette fonction dans lequel les noms des arguments formels peuvent être omis, suivie d'un point-virgule. Par exemple :

```
int est_pair(int);
```

est la déclaration externe d'une fonction de nom `est_pair` ayant un argument de type `int` et retournant une valeur de type `int`.

Donc, une fonction, une variable ou un tableau qui est utilisé dans un fichier source F_1 et définie dans un autre fichier source F_2, doit faire l'objet dans F_1 d'une déclaration externe précédent sa première utilisation.

Les déclarations externes de fonctions peuvent aussi s'avérer nécessaire, dans le fichier source dans lequel elles sont définies, selon l'imbrication de leurs appels. Soit, par exemple, un fichier source composé d'une fonction `main`, de deux fonctions `f` et `g` et d'une variable globale `v`. Supposons que la fonction `main` appelle la fonction `f` qui appelle la fonction `g` et que la variable globale `v` est utilisée par les fonctions `main` et `f`, comme l'illustre le graphe suivant :



La déclaration de `v` et les définitions de `main`, `f` et `g` devront être écrites dans le fichier source dans un ordre tel que la déclaration de `g` précède celle de `f` qui précède celle de `main` et tel que la déclaration de `v` précède les définitions de `f` et de `main`. Le fichier source suivant respecte ces contraintes :

```
... v;                  définition de v
... g(...)              définition et donc déclaration de g
    {                   |
    ...                 |
    }                   |
... f(...)              définition et donc déclaration de f
    {                   |
    ...v...             | utilisation de v qui a déjà été déclarée
    ...g(...)...        | appel de g qui a déjà été déclarée
    }                   |
```

```
int main(void)          définition de main
    {                   |
    ...v...             | utilisation de v qui a déjà été déclarée
    ...f(...)...        | appel de f qui a déjà été déclarée
    }                   |
```

En C, les fonctions peuvent être récursives, c'est-à-dire s'appeler elles-mêmes :

- soit directement : un appel à *f* est placé dans le corps d'une fonction *f* ;
- soit indirectement : un appel à *f* est placé dans le corps d'une fonction *g* qui est appelée directement ou indirectement par une fonction *f* (les fonctions *f* et *g* sont dites **mutuellement récursives**).

Nous étudierons les fonctions récursives au chapitre 10. Nous ne nous préoccupons ici que de l'ordre dans lequel les déclarer.

Lorsqu'une fonction *f* est directement récursive, la règle énoncée ci-dessus est respectée puisque, la définition d'une fonction étant constituée de son en-tête (sa déclaration) suivie de son corps, lorsque le compilateur lit l'appel de *f*, il a obligatoirement déjà lu la déclaration de *f*, comme le montre le schéma suivant :

```
...f(...)               définition et donc déclaration de f
    {                   |
    ...                 |
    ...f(...)...;       | appel de f qui a déjà été déclarée
    ...                 |
    }                   |
```

La situation est un peu plus compliquée dans le cas des fonctions mutuellement récursives. Par exemple, une fonction *f* qui appelle une fonction *g* qui elle-même appelle *f*. En ce cas quelque soit l'ordre dans lequel sont rangées les définitions de *f* et de *g*, un des appels à *f* ou à *g* ne sera pas précédé de sa déclaration, comme le montre le schéma suivant :

```
... f(...)              définition de f
    {                   |
    ...                 |
    ...g(...)...;       | appel de g qui n'a pas encore été déclarée
    ...                 |
    }                   |
... g(...)
    {
    ...
    ...f(...)...;
    ...
    }
```

La solution consiste à séparer les déclarations de `f` et de `g` de leurs définitions comme le montre le schéma suivant :

```
... f(...);             déclaration externe de f
... g(...);             déclaration externe de g
```

```
... f(...)              définition de f
    {                   |
    ...g(...)...        | appel de g qui a déjà été déclarée
    }                   |
    g(...)              définition de g
    {                   |
    ...                 |
    ...f(...)...;       | appel de f qui a déjà été déclarée
    ...                 |
    }                   |
```

Attention ! Lorsque le compilateur lit dans une instruction un nom de fonction qui n'a pas été déclarée préalablement, il considère que sa valeur de retour a été implicitement déclarée de type `int`. Si c'est le type qui était prévu cela ne pose pas de problème. Sinon, des erreurs se produiront soit dans la suite de la compilation, soit lors de l'exécution. Il est donc fortement recommandé de ne pas utiliser cette possibilité de déclaration par défaut. □

5.2.4 La bibliothèque standard

La bibliothèque standard contient les codes objet des fonctions utilitaires distribués avec chaque implantation de C : fonctions d'entrées-sorties, fonctions mathématiques, fonctions de manipulation de chaînes de caractères, etc.

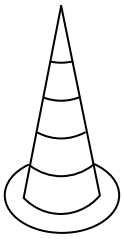
Associée à cette bibliothèque, un ensemble de fichiers d'en-têtes est également distribué. Ces fichiers contiennent les déclarations des fonctions de la bibliothèque standard, des déclarations de types ou de constantes associés à ces fonctions, etc.

Les fichiers d'en-têtes suivant contiennent les déclarations des fonctions de la bibliothèque standard et des définitions de constantes qui sont utilisées dans les exemples et les exercices de ce cours :

- `ctype.h` contient les déclarations des fonctions qui testent à quelle catégorie appartient un caractère : alphabétique, numérique, alphanumérique, etc. et celles des fonctions de conversion d'une lettre en minuscule ou en majuscule ;
- `float.h` contient le plus petit et le plus grand nombre représentables ainsi que le nombre de chiffres significatifs pour chaque type numérique flottant de C ;
- `limits.h` contient notamment les valeurs minimale et maximale des instances de chaque type numérique entier de C ;
- `math.h` contient les déclarations des principales fonctions mathématiques : exponentielles, logarithmiques, trigonométriques, hyperboliques, etc. ;
- `stddef.h` contient notamment la définition de la macro `NULL` : un pointeur qui ne pointe sur rien et dont nous étudierons l'utilisation au chapitre 8 consacré aux pointeurs ;
- `stdio.h` contient les déclarations des fonctions d'entrées-sorties dont les fonctions `printf` et `scanf` que nous avons déjà rencontrées et celles que nous étudierons au chapitre 6 ;
- `stdlib.h` contient les déclarations de diverses fonctions utilitaires : conversions, allocation dynamique de mémoire que nous étudierons au chapitre 8, sortie d'un programme dont la fonction `exit` (voir ci-dessus paragraphe 5.1.5), génération de nombres aléatoires, tri, recherche dichotomique, etc. ;
- `string.h` contient les déclarations des fonctions de manipulation des chaînes de caractères dont celles que nous étudierons au chapitre 9.

5.3 Un programme complet

Il s'agit d'écrire un programme qui calcule l'aire d'une balise de signalisation routière constituée d'un cône de révolution posé sur un anneau de façon à ce que le cercle de base du cône et le cercle intérieur de l'anneau coïncident, comme le montre la figure suivante :



Soit h la hauteur du cône, r_c le rayon du cercle de base du cône qui est égal au rayon intérieur de l'anneau et r_a le rayon extérieur de l'anneau. L'aire de cette balise est égale à l'aire latérale d'un cône de révolution de rayon r_c et de hauteur h augmentée de l'aire d'un l'anneau de rayon intérieur r_c et de rayon extérieur r_a. Cette dernière est égale à l'aire d'un disque de rayon r_a moins l'aire d'un disque de rayon r_c.

La connaissance de l'aire de cette balise pourrait permettre, par exemple, de calculer la quantité de PVC pour la fabriquer.

Ce programme, que nous appellerons *Balise*, sera constitué[1] :

- de la définition d'une constante de valeur 3,14 : une valeur approchée de π ;
- de la définition d'une fonction `aire_laterale_cone` qui calcule l'aire latérale d'un cône de révolution de rayon r et de hauteur h par la formule $\pi r\sqrt{r^2+h^2}$ (où $\sqrt{r^2+h^2}$ est l'apothème du cône) ;
- de la définition d'une fonction `aire_disque` qui calcule l'aire d'un disque de rayon r par la formule πr^2 ;
- de la définition de la fonction `main` qui demande à l'utilisateur la hauteur du cône, le rayon de son cercle de base et le rayon extérieur de l'anneau puis calcule et affiche l'aire de cette balise.

Nous en donnerons deux versions :

- une version mono-fichier dans laquelle la valeur approchée de π est affectée à une variable globale `pi` dont la valeur ne doit pas être modifiée ;
- une version multi-fichiers dans laquelle la valeur approchée de π est désignée par la constante symbolique `PI` et dans laquelle les définitions des fonctions `aire_laterale_cone` et `aire_disque` sont enregistrées dans un fichier à part, afin de pouvoir être réutilisées dans d'autres programmes.

[1] Ce programme pourrait bien entendu se réduire à la fonction `main` dans laquelle l'aire recherchée serait directement calculée. C'est le besoin d'un exemple simple pour illustrer ce chapitre qui justifie une telle décomposition.

Version mono-fichier

Le texte du programme, qui est enregistré dans le fichier source `balise-tout-en-un.c`, est le suivant :

```
(1)  #include <stdio.h>
(2)  #include <math.h>

(3)  const float pi = 3.14;

(4)  float aire_laterale_cone(float r, float h)
(5)     {
(6)     return pi * r * sqrt(pow(r, 2) + pow(h, 2));
(7)     }

(8)  float aire_disque(float r)
(9)     {
(10)    return pi * pow(r, 2);
(11)    }

(12) int main(void)
(13)    {
(14)    float h, rc, ra, aire_balise;
(15)    printf("Hauteur du cone (en cm) ? ");
(16)    scanf("%f", &h);
(17)    printf("Rayon du cercle de base du cone (en cm) ? ");
(18)    scanf("%f", &rc);
(19)    printf("Rayon exterieur de l'anneau (en cm) ? ");
(20)    scanf("%f", &ra);
(21)    aire_balise = aire_disque(ra) - aire_disque(rc)
                    + aire_laterale_cone(h, rc);
(22)    printf("Aire de la balise = %.0f cm2\n", aire_balise);
(23)    return 0;
(24)    }
```

Commentons-le :

- La ligne 1 est une directive demandant l'inclusion du fichier d'en-têtes des fonctions d'entrées-sorties. Ceci permet de faire appel aux fonctions `printf` et `scanf` dans la fonction `main` (lignes 15 à 20 et 22).
- La ligne 2 est une directive demandant l'inclusion du fichier d'en-têtes des fonctions mathématiques. Ceci permet de faire appel aux fonctions `pow` (puissance) et `sqrt` (racine carré) dans le corps des fonctions `aire_laterale_cone` et `aire_disque` (lignes 6 et 10).
- La ligne 3 est la définition d'une variable globale `pi` dont le valeur est fixée à 3,14. Cette variable est visible dans les corps des fonctions `aire_laterale_cone`, `aire_disque` et `main` qui suivent.
- Les lignes 4 à 7 définissent la fonction `aire_laterale_cone` qui retourne l'aire latérale d'un cône de révolution de rayon du cercle de base `r` et de hauteur `h`. Cette fonction est visible dans son corps et dans ceux des fonctions `aire_disque` et `main`.
- Les lignes 8 à 11 définissent la fonction `aire_disque` qui retourne l'aire d'un disque de rayon `r`. Cette fonction est visible dans son corps et dans celui de la fonction `main`.
- Les lignes 12 à 24 définissent la fonction `main`. La ligne 14 définit quatre variables locales `h`, `rc`, `ra` et `aire_balise` qui ne sont visibles que dans le corps de la fonction `main`. Les

instructions 15 à 20 demandent à l'utilisateur de saisir la hauteur du cône, le rayon de son cercle de base, et le rayon extérieur de l'anneau, mesurés en cm, lisent ces valeurs et les affectent aux variables `h`, `rc` et `ra`. L'instruction 21 calcule l'aire de la balise et l'affecte à la variable locale `aire_balise`. L'instruction 22 affiche cette aire en cm^2.

Version multi-fichiers

Afin de pouvoir être réutilisées dans d'autres programmes, les définitions des fonctions `aire_laterale_cone` et `aire_disque` sont enregistrées dans un fichier à part. Ceci implique de créer un fichier d'en-têtes contenant les déclarations de ces deux fonctions, qui devra être inclus dans tout fichier contenant des appels à ces fonctions.

De plus, au lieu d'être affectée à une variable, la valeur approchée de π sera désignée par la constante symbolique `PI`.

Le programme *Balise* est donc découpé en trois fichiers :

1. le fichier `aires.h` :

```
#ifndef AIRES
#define AIRES
#define PI 3.14
float aire_laterale_cone(float r, float h);
float aire_disque(float);
#endif
```

   qui contient la définition de la constante symbolique `PI` ainsi que les déclarations des fonctions `aire_laterale_cone` et `aire_disque` (la signification des deux premières lignes sera expliquée ci-dessous) ;

2. le fichier `aires.c` :

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

float aire_laterale_cone(float r, float h)
   {
   return PI * r * sqrt(pow(r, 2) + pow(h, 2));
   }

float aire_disque(float r)
   {
   return PI * pow(r, 2);
   }
```

   qui contient les définitions des fonctions `aire_laterale_cone` et `aire_disque`, précédées de l'inclusion des fichiers d'en-têtes `math.h` et `aires.h` qui contiennent les déclarations des fonctions `pow` et `sqrt` et la définition de la constante `PI`, utilisées dans les corps des fonctions `aire_laterale_cone` et `aire_disque` ;

3. le fichier `balise.c` :

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

```
int main(void)
   {
   float h, rc, ra, aire_balise;
   printf("Hauteur du cone (en cm) ? ");
   scanf("%f", &h);
   printf("Rayon du cercle de base du cone (en cm) ? ");
   scanf("%f", &rc);
   printf("Rayon exterieur de l'anneau (en cm) ? ");
   scanf("%f", &ra);
   aire_balise = aire_disque(ra) - aire_disque(rc)
                 + aire_laterale_cone(h, rc);
   printf("Aire de la balise = %.0f cm2\n", aire_balise);
   return 0;
   }
```

qui contient la définition de la fonction `main` précédée de la directive d'inclusion des fichiers d'en-têtes `stdio.h` et `aires.h` qui contiennent les déclarations des fonctions `scanf`, `printf`, `aire_laterale_cone` et `aire_disque`, utilisées dans le corps de la fonction `main`.

Il se peut que l'inclusion d'un même fichier d'en-têtes soit demandée dans plusieurs fichiers sources d'un même programme. Il faut empêcher que ce fichier soit inclus plusieurs fois afin d'éviter les erreurs dues aux définitions multiples d'une même entité. Cela peut être fait en utilisant la directive de compilation conditionnelle :

```
#ifndef m
```

Cette directive indique au préprocesseur que si la macro *m* a été définie, il ne doit pas recopier dans le fichier à compiler les lignes qui suivent cette directive jusqu'à celle, comprise, qui précède la directive :

```
#endif
```

Chaque fichier d'en-têtes sera donc composé de la façon suivante :

(1) `#ifndef` *m*
(2) `#define` *m*
contenu du fichier d'en-têtes
(*n*) `#endif`

Si la macro *m* n'a pas été définie (ligne 1) alors elle l'est (ligne 2) puis le contenu du fichier d'en-têtes est lu et traité jusqu'à la rencontre de la directive `#endif` (ligne *n*) de même niveau que la directive `#ifndef` de la ligne 1. Si la macro *m* a déjà été définie, alors les lignes 2 à *n* – 1 ne sont pas traitées par le préprocesseur. Ainsi, le contenu d'un fichier d'en-têtes n'est traité que par la première directive d'inclusion de ce fichier et ignoré par les suivantes. C'est ainsi qu'a été composé le fichier `aires.h`.

5.4 Génération du code exécutable d'un programme

Maintenant que nous savons écrire un programme, voyons comment générer son code exécutable en utilisant le compilateur GCC. Nous illustrerons cette procédure sur la version multi-fichiers du programme *Balise*.

5.4.1 Compilation des fichiers sources

Comme nous l'avons dit ci-dessus, la compilation d'un fichier source consiste à générer un fichier objet qui contient les codes objets des fonctions définies dans ce fichier. Le code objet

d'une fonction contient du code exécutable ainsi que des appels vers des fonctions définies dans d'autres fichiers sources ou dans une bibliothèque (par exemple, les fonctions d'entrées-sorties) mais dont l'adresse en mémoire n'est pas encore connue.

La commande UNIX :

```
gcc -Wall -c fic.c
```

lance la compilation du fichier source *fic*`.c`. Les erreurs éventuelles sont détectées et des avertissements (« warnings ») sont émis en cas d'incorrections ou d'oublis qui pourraient entraîner des problèmes lors de l'exécution. Si aucune erreur n'est détectée, le fichier objet *fic*`.o` est généré.

L'intérêt de l'option `-Wall` est de fournir une liste très complète d'avertissements concernant des erreurs ou des oublis qui, bien que n'empêchant pas la compilation, peuvent être la source d'une exécution incorrecte du programme. C'est pourquoi cette option est particulièrement conseillée.

Par exemple, la commande :

```
gcc -Wall -c aires.c
```

génère le fichier objet `aires.o` à condition bien entendu que des erreurs n'aient pas été détectées.

Notons que la possibilité de compiler séparément les fichiers sources permet, en cas de mise à jour d'un programme, de ne pas avoir à recompiler les fichiers qui ne sont pas touchés par cette mise à jour.

5.4.2 **Génération du code exécutable**

Le code exécutable d'un programme C est obtenu en liant (édition de liens) les codes objets des fonctions auxquelles ce programme fait appel.

Soit fic_1, …, fic_n les fichiers source ou objet des modules d'un programme. La commande :

`gcc` [`-Wall`] fic_1 … fic_m [`-l`bib_1] … [`-l`bib_n] `-o` *fic*

compile parmi les fichiers fic_1, …, fic_m et avec l'option `-Wall` ceux qui sont des fichiers sources (`.c`), réalise l'édition de liens et génère, si aucune erreur n'est détectée, le fichier exécutable *fic* (ou `a.out` si l'option `-o` n'est pas présente). L'exécution du programme pourra alors être lancée par la commande *fic* (ou `a.out`).

Lorsqu'un programme utilise des fonctions de bibliothèque, l'éditeur de liens doit connaitre le nom du fichier qui contient les codes objet de ces fonctions. Pour les fonctions de la bibliothèque standard, autres que les fonctions mathématiques, il n'est pas nécessaire de fournir ce nom : l'éditeur de liens le connait. Pour les autres fonctions de bibliothèque, dont notamment les fonctions mathématiques de la bibliothèque standard, il faut indiquer le nom du fichier qui contient leur code objet. Cela se fait de la façon suivante. Un fichier de bibliothèque est stocké dans un répertoire (en général `/usr/lib`) connu de l'éditeur de liens et a un nom de la forme `lib`*nom* (le préfixe `lib` est toujours présent). Si un programme utilise une fonction d'une bibliothèque nommée `lib`*nom*, il faut ajouter l'option `-l`*nom* à la commande de génération de son code exécutable.

Le fichier de bibliothèque qui contient le code objet des fonctions mathématiques de la bibliothèque standard a pour nom `libm`. Si un programme fait appel à certaines de ces

fonctions, il faut ajouter l'option `-lm` dans la commande de génération de son code exécutable.

Par exemple, la commande :

```
gcc -Wall aires.o balise.c -lm -o balise
```

génère le fichier exécutable `balise` à partir du fichier objet `aires.o` (généré par la compilation du fichier objet `aires.c`), du fichier source `balise.c` et de la bibliothèque mathématique, nécessaire car les fonction `pow` (puissance) et `sqrt` (racine carrée) sont utilisées dans le fichier source `balise.c`.

L'exécution de ce programme sera lancée par la commande `balise`. Par exemple :

```
balise↵
Hauteur du cone (en cm) ? 50↵
Rayon du cercle de base du cone (en cm) ? 10↵
Rayon exterieur de l'anneau (en cm) ? 14↵
Aire de la balise = 8307 cm2
```

Les caractères saisis au clavier et renvoyés en écho sur l'écran sont écrits en gras. La saisie de la touche « Entrée » est symbolisée par le caractère « ↵ ».

5.4.3 L'outil *make*

Lorsqu'un fichier composant un programme est mis à jour, il faut régénérer le code exécutable de ce programme. Mais ceci n'implique pas obligatoirement de recompiler tous les fichiers composant ce programme. Seuls ceux qui dépendent du fichier mis à jour doivent l'être. Pour faciliter la tâche du programmeur, UNIX offre une commande appelée `make` qui permet de décrire les dépendances entre les fichiers composant un programme ainsi que les commandes à exécuter pour remettre à jour ces fichiers lorsque l'un des fichiers dont ils dépendent a lui-même été mis à jour.

Les dépendances entre les fichiers composant un programme sont décrites dans un fichier dit fichier `makefile` par un ensemble de règles de la forme :

fic : fic_1 … fic_n
    commandes (*précédées chacune d'une tabulation*)

et dont la signification est la suivante. Le fichier *fic* dépend des fichiers fic_1, …, fic_n. Pour que le fichier *fic* soit à jour, il faut que les fichiers fic_1, …, fic_n le soient aussi, c.-à-d. qu'ils aient une date de dernière mise à jour inférieure à celle du fichier *fic*. Si ce n'est pas le cas, alors les commandes spécifiées devront être exécutées pour mettre à jour le fichier *fic*.

Pour établir ces règles, il est pratique de tracer le graphe de dépendances des fichiers composant le programme. Dans ce graphe, chaque nœud est associé à un fichier et une arête entre un nœud origine fic_1 et un nœud extrémité fic_2 indique que le fichier fic_1 dépend du fichier fic_2.

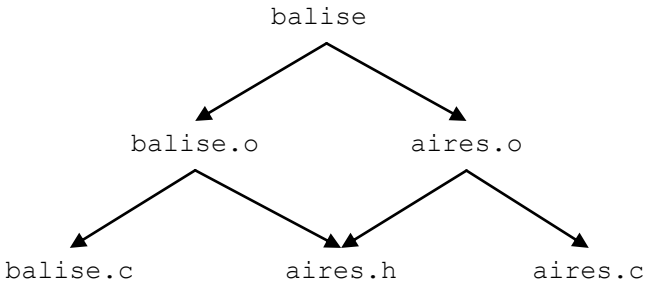


Figure 5.3. *Graphe de dépendances des fichiers du programme Balise*

Par exemple, le graphe de dépendances entre les fichiers composant le programme *Balise* est présenté sur le figure 5.3. A partir de ce graphe, on peut établir les règles permettant de générer le fichier exécutable `balise`. Ces règles sont les suivantes :

```
balise: balise.o aires.o
   gcc aires.o balise.o -o balise
balise.o: aires.h balise.c
   gcc –Wall -c balise.c
aires.o: aires.h aires.c
   gcc –Wall -c aires.c
```

La génération du fichier exécutable d'un programme dont les règles de construction sont contenues dans le fichier *fic* est lancée par la commande :

`make –f` *fic*

Par exemple, si les règles ci-dessus ont été enregistrées dans le fichier `balise.mk`, la génération du fichier exécutable `balise` sera réalisée par la commande :

```
make -f balise.mk
```

Pour comprendre le fonctionnement de la commande `make`, supposons que les fichiers `aires.h`, `aires.c`, `balise.c`, `aires.o`, `balise.o` et `balise` aient été créés aux dates respectives $d_1 < d_2 < d_3 < d_4 < d_5 < d_6$. Supposons maintenant qu'une mise à jour du fichier `balise.c` soit effectuée à la date $d_7 > d_6$. Il faut donc régénérer le fichier exécutable `balise`. On lance donc la commande :

```
make –f balise.mk
```

qui applique les règles du fichier `balise.mk` de la façon suivante :

1. Le fichier `balise` est à jour si les fichiers `aires.o` et `balise.o` le sont.
2. Le fichier `balise.o` n'est pas à jour car sa date de dernière mise à jour (d_5) est inférieure à celle (d_7) du fichier `balise.c` dont il dépend. Il doit donc être recompilé et sa date de dernière mise à jour devient $d_8 > d_7$.
3. Le fichier `aires.o` est à jour car sa date de dernière mise à jour (d_4) est supérieure à celles (d_1 et d_2) des fichiers `aires.h` et `aires.c` dont il dépend. Il est donc inutile de le recompiler.
4. Le fichier `balise` n'est pas à jour car sa date de dernière mise à jour (d_6) est inférieure à celle (d_8) du fichier `balise.o` dont il dépend. Il est donc régénéré.

5.5 Ecriture d'un programme : règles de bonne pratique

Un programme est un objet qui évolue au cours du temps, soit par des corrections au cours de sa mise au point, soit par des mises à jour destinées à l'améliorer. Il est donc nécessaire que son auteur ou que d'autres programmeurs puissent le relire et le comprendre facilement.

Par ailleurs, un programme doit être sûr. Son exécution ne doit pas s'interrompre brutalement sans qu'aucun message clair sur la raison de cette interruption ne soit affiché. Elle ne doit pas non plus boucler indéfiniment. Elle ne doit pas, enfin, provoquer l'écriture d'informations dans des zones de la mémoire non prévues à cet effet.

Voici quelques règles de bonne pratique pour l'écriture d'un programme.

Choix des noms

Les noms choisis doivent permettre d'identifier facilement les objets désignés (constantes, variables, fonctions…). Lorsqu'un nom est composé, ses différentes parties pourront être séparées, soit par un blanc souligné, soit en mettant en majuscule la première lettre de chaque nom composant. Par exemple, une variable désignant un prix total pourra être nommée soit `prix_total`, soit `prixTotal`.

Etant donné que ce sont les fonctions qui structurent un programme, il ne faut pas hésiter à leur donner un nom significatif quitte à ce qu'il soit long. Par exemple, `hauteur_pyramide` est bien préférable à un nom abrégé tel que `haut_pyram` ou `hpyr`. Cela s'applique aussi aux noms des variables globales.

Concernant les variables locales, il faut trouver un compromis entre des noms trop courts et non significatifs et des noms trop longs fastidieux à manipuler. Par exemple, les variables désignant une valeur minimum et une valeur maximum pourront avoir pour nom `min` et `max`, plutôt que `m1` et `m2`, qui ne sont pas significatifs ou `minimum` et `maximum` qui sont un peu longs à écrire.

Indentation

La disposition du texte d'un programme C est libre. Des espaces, des tabulations et des retours à la ligne peuvent être placés à tout endroit où cela ne coupe pas un identificateur. Cependant, il est fortement conseillé, pour la lisibilité d'un programme, de l'indenter afin de mettre en valeur ses différents constituants. On se convaincra facilement qu'il est beaucoup lisible d'écrire :

```
if (temp > 30)
   printf("Allumer la climatisation !")
else
   if (temp < 20)
      printf("Eteindre la climatisation !")
```

que :

```
if (temp > 30) printf("Allumer la climatisation !") else if (temp < 20)
printf("Eteindre la climatisation !")
```

Plusieurs styles d'indentation 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.

Commentaires

Un commentaire est un texte libre qui apporte les informations nécessaires à la compréhension d'un programme. En C 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 commentaires :

```
/* Ceci est un commentaire. */

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

Tout programme doit être commenté, mais il faut veiller à ce que les commentaires apportent des informations utiles à la compréhension. Inutile, par exemple, de faire précéder l'instruction `x = 2;` du commentaire :

```
/* On affecte la valeur 2 à la variable x */
```

Utilisation de constantes symboliques

Il est conseillé de désigner les constantes dont dépend un programme (valeurs minimum ou maximum d'une variable, valeur approchée d'un nombre irrationnel tel que π, nombre d'éléments d'un tableau, etc.) par un nom au moyen de la macro `define`. Comme nous l'avons indiqué ci-dessus au paragraphe 5.2.1, cette pratique améliore la lisibilité du programme et apporte une certaine forme de paramétrage : plusieurs versions d'un même programme peuvent être obtenues en changeant uniquement les valeurs associées aux constantes symboliques.

Ecriture de programmes sûrs

Il faut :

- vérifier que les données lues, qu'elles soient saisies par l'utilisateur ou enregistrées dans un fichier, sont conformes à celles attendues par le programme ;
- vérifier que les valeurs des arguments transmis à une fonction ou que la valeur retournée par cette fonction soient conformes à celles attendues ;
- vérifier qu'il n'y aura pas de débordement de tableau, car, comme nous le verrons au chapitre 7, C ne contrôle pas ce débordement ;
- vérifier que le programme ne bouclera pas indéfiniment, ce qui implique de faire attention à l'écriture des boucles `while`, `do while` ou `for` (voir chapitre 4, paragraphe 4.6) et des fonctions récursives (voir chapitre 10) ;

Une fois une erreur détectée, le programme doit la signaler, soit par un code d'erreur retourné par la fonction dont l'appel a produit cette erreur, soit par un message envoyé à l'utilisateur ou au processus qui a lancé le programme. Si cette erreur peut-être récupérée, il faut le faire, sinon, il faut interrompre le programme. Par exemple, si l'utilisateur saisit une donnée incorrecte, il pourra lui être demandé de la saisir à nouveau. Les langages de programmation avancés, comme C++, Caml ou Java, intègrent un mécanisme dit de « gestion des exceptions » qui facilitent la récupération des erreurs, mais ce n'est malheureusement pas le cas de C.

Exercices

Exercice 5.1. Ecrire un programme qui calcule les solutions réelles, quand elles existent d'une équation du second degré $ax^2 + bx + c$ dont les paramètres a, b et c seront demandés à l'utilisateur. On utilisera les deux fonctions suivantes de la bibliothèque standard : la fonction `sqrt` qui appliquée à un flottant retourne sa racine carrée et la fonction `pow` qui appliquée à

deux nombres flottants x et y retourne la valeur de x^y. Les déclarations de ces deux fonctions se trouvent dans le fichier d'en-têtes `math.h`. Il faudra de plus, lors de l'édition de liens, lier la bibliothèque mathématique. On compilera donc le programme par la commande :

```
gcc –Wall exo5.1.c –lm –o exo5.1
```

On pourra tester le programme sur les trois équations suivantes :

– l'équation : $-2x^2 + 7x + 15$ qui a deux racines réelles : $-1{,}5$ et 5
– l'équation : $x^2 - 2x + 1$ qui a une racine double : 1 ;
– l'équation : $x^2 + 3x + 3$ qui n'a pas de racines réelles.

Exercice 5.2. Ecrire un programme qui calcule puis affiche les n premiers nombres premiers, n étant demandé à l'utilisateur. Ce programme devra être composé :

– d'une fonction `est_premier` qui appliquée à un nombre entier retourne 1 s'il est premier ou 0 sinon ;
– d'une fonction `premiers_premiers` qui appliquée à un nombre entier positif n, affiche les n premiers nombres premiers calculés en utilisant la fonction `est_premier` ;
– de la fonction `main` qui demande à l'utilisateur de saisir le nombre des premiers nombres premiers qu'il désire voir afficher et les affiche en appelant la fonction `premiers_premiers`.

On pourra tester le programme en calculant les 10 premiers nombres premiers qui sont : 2, 3, 5, 7, 11, 13, 17, 19, 23 et 29.

Exercice 5.3. La commande `gcc` possède une option qui génère le fichier obtenu (*fichier à compiler*) après traitement du fichier source par le préprocesseur. Si *fic*`.c` est le fichier source, la commande :

```
gcc –E fic.c
```

soumet le fichier *fic*`.c` au préprocesseur et affiche le fichier produit.

Appliquer cette commande au fichier source :

```
#define N 10

int main(void)
   {
   int i, s = 0;
   for (i = 1; i < N; i++)
      s = s + i;
   return 0;
   }
```

et expliquer le résultat obtenu.

Exercice 5.4. Ecrire un programme qui calcule le plus grand nombre n tel que $n!$ soit inférieure la valeur maximum d'un nombre de type `unsigned long`. Cette valeur maximum est celle de la constante `LONG_MAX` déclarée dans le fichier d'en-têtes `limits.h`.

Exercice 5.5.

1. Décomposer le programme de l'exercice 5.2 en trois fichiers :
   - `exo5.2.bib.c`, contenant la définition de la fonction `est_premier` ;
   - `exo5.2.bib.h` contenant la déclaration de la fonction `est_premier` ;
   - `exo5.2.prog.c` contenant la fonction `premiers_premiers` et la fonction `main`.

2. Tracer le graphe de dépendances entre ces trois fichiers, les fichiers objet et le fichier exécutable du programme.
3. Construire le fichier *makefile* du programme et l'expérimenter selon le scénario suivant :
   - Lancer la commande `make -f exo5.2.mk`. Quels fichiers ont été générés ?
   - Lancer la commande `make -f exo5.2.mk`. Qu'affiche-t-elle ? Pourquoi ?
   - Noter les dates des fichiers de préfixe `exo5.2`, effacer le fichier `exo5.2.bib.o`, lancer la commande `make -f exo5.2.mk`. Quels fichiers de préfixe `exo5.2` ont été ajoutés ou modifiés (changement de date) et lesquels ne l'ont pas été ? Pourquoi ?
   - Noter les dates des fichiers de préfixe `exo5.2`, faire une modification dans la fonction `main` du fichier `exo5.2.prog.c` et lancer la commande `make -f exo5.2.mk`. Quels fichiers de préfixe `exo5.2` ont été modifiés (changement de date) et lesquels ne l'ont pas été. Pourquoi ?

6
Entrées-sorties

En C, les opérations d'entrées-sorties : saisie au clavier, affichage à l'écran, lecture et écriture sur disque, etc. sont réalisées par des fonctions prédéfinies de la bibliothèque standard de C qui sont déclarées dans le fichier d'en-têtes `stdio.h`. Lorsqu'un fichier source contient des appels à ces fonctions, il faudra donc inclure la directive `#include <stdio.h>` au début de ce fichier source.

Les fonctions d'entrées-sorties 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 texte » et dans le second ils sont dits « fichiers binaires ». Par exemple, le texte source d'un programme est enregistré dans un fichier texte et son code exécutable dans un fichier binaire. Dans ce livre, nous ne traiterons que des fichiers texte.

Ces fichiers sont affectés à des unités d'entrées-sorties qui peuvent être un clavier, un écran, une imprimante, un disque, etc. Il existe trois fichiers particuliers toujours disponibles lors de l'exécution d'un programme : le fichier d'entrée standard qui est généralement affecté au clavier du poste de travail, le fichier de sortie standard qui est généralement affecté à l'écran du poste de travail et le fichier d'erreur standard qui est aussi généralement affecté à l'écran du poste de travail.

Dans la suite de ce chapitre, nous étudierons : la lecture de données saisies au clavier et l'affichage de données à l'écran (paragraphe 6.1) ; la lecture et l'écriture de données dans des fichiers texte (paragraphe 6.2).

6.1 Saisie et affichage de données

Les caractères saisis au clavier sont écrits dans le fichier qui lui est affecté : le fichier d'entrée standard. Un programme qui veut lire ces caractères doit donc les lire dans ce fichier. Ce fichier possède un curseur qui marque la position du prochain caractère à lire, ainsi qu'une marque de fin de fichier. Au début de l'exécution du programme, le fichier d'entrée standard est vide : le curseur est positionné sur la marque de fin de fichier.

Lorsqu'une demande de lecture de caractère est faite par le programme, deux cas peuvent se produire selon la position du curseur :

– si le curseur est positionné sur la marque de fin de fichier (fichier vide ou fin de fichier atteinte), l'exécution du programme est suspendue et la main est donnée à l'utilisateur qui saisit une suite de caractères terminée par la frappe de la touche « Entrée ». Les caractères saisis sont écrits dans l'ordre de leur saisie dans le fichier d'entrée standard à partir de son début y compris le caractère « nouvelle ligne » généré par la frappe de la touche « Entrée » et le curseur est positionné au début du fichier. La main est alors redonnée au programme qui poursuit son exécution.

– sinon, le caractère marqué par le curseur est retourné et le curseur est positionné sur le caractère suivant ou sur la marque de fin de fichier si le caractère lu est le dernier du fichier.

L'affichage de caractères à l'écran est réalisé à partir d'une position marquée par un caractère spécial, clignotant le plus souvent : le curseur de l'écran.

Pour afficher une suite de caractères à l'écran, un programme doit les écrire dans le fichier affecté à l'écran : le fichier de sortie standard. Un caractère écrit dans ce fichier :

– est affiché à l'écran à la position du curseur de l'écran, si c'est un caractère imprimable (lettre, chiffre, signe de ponctuation, etc.) ;
– positionne le curseur de l'écran sur la ligne suivante, si c'est le caractère « nouvelle ligne » ou sur la prochaine position de tabulation si c'est le caractère de tabulation.

En résumé, pour lire les caractères saisis au clavier, il suffit de les lire dans le fichier d'entrée standard et pour afficher des caractères à l'écran, il suffit de les écrire dans le fichier de sortie standard. Nous allons donc étudier les fonctions de lecture et d'écriture dans les fichiers d'entrée et de sortie standards : caractère par caractère (paragraphe 6.1.1) et avec format (paragraphe 6.1.2).

Dans les exemples qui suivent :

– nous supposerons que le fichier d'entrée standard est affecté au clavier et que le fichier de sortie standard est affecté à l'écran ;
– nous écrirons en gras les caractères saisis au clavier et renvoyés en écho sur l'écran et représenterons par le symbole ↵ la frappe de la touche « Entrée ».

Une constante joue un rôle important, c'est la constante `EOF` (« End of File ») définie dans le fichier d'en-têtes `stdio.h`. Elle a la valeur −1 qui a été choisie parce qu'elle n'est pas le code d'un caractère qui est un entier positif ou nul.

6.1.1 **Lecture et écriture d'un caractère**

L'écriture d'un caractère dans le fichier de sortie standard peut être effectuée par l'appel de la fonction `putchar` d'en-tête :

```
int putchar(int c)
```

L'appel `putchar(`*c*`)` a pour effet d'écrire le caractère *c* à la fin du fichier de sortie standard. Le caractère *c* est retourné, sauf si une erreur s'est produite, auquel cas `EOF`[1] est retournée.

La lecture d'un caractère dans le fichier d'entrée standard peut être effectuée par l'appel de la fonction `getchar` d'en-tête :

```
int getchar(void)
```

L'appel `getchar(`*c*`)` a pour effet de lire dans le fichier d'entrée standard le caractère à la position du curseur. Le caractère lu est retourné, sauf si la fin du fichier a été atteinte ou si une erreur s'est produite, auquel cas `EOF` est retournée.

[1] A partir de maintenant et pour ne pas alourdir l'écriture, nous désignerons souvent une valeur *v* par une expression dont la valeur est *v*. Nous écrirons, par exemple, « `EOF` est retournée » au lieu de « la valeur de `EOF` » est retournée.

Exemple 6.1. Le programme suivant lit un caractère dans le fichier d'entrée standard et l'écrit dans le fichier de sortie standard tant que le caractère lu est différent du caractère « nouvelle ligne » :

```
#include <stdio.h>

int main(void)
   {
   int c;
   c = getchar();
   while (c != '\n')
      {
      putchar(c);
      c = getchar();
      }
   return 0;
   }
```

Une écriture classique et plus compacte du corps de la fonction `main` est la suivante :

```
{
int c;
while ((c = getchar()) != '\n')
   putchar(c);
return 0;
}
```

qui peut être paraphrasé par : « tant que le caractère saisi affecté à `c` est différent du caractère 'nouvelle ligne', afficher ce caractère et recommencer ».

Une exécution de ce programme pourra être la suivante :

```
Bonjour↵
Bonjour
```

□

L'exemple 6.2 permet de bien comprendre la lecture de caractères saisis au clavier.

Exemple 6.2. Considérons le programme suivant :

```
#include <stdio.h>

int main(void)
   {
   int c1, c2, c3;
   c1 = getchar();
   c2 = getchar();
   c3 = getchar();
   putchar(c1);
   putchar(c2);
   putchar(c3);
   return 0;
   }
```

Une première exécution possible est la suivante :

```
abc↵
abc
```

Le fichier d'entrée standard étant vide, le premier appel à `getchar` a donné la main à l'utilisateur qui a saisi les caractères « a », « b », « c » et « nouvelle ligne » qui ont été écrits dans ce fichier d'entrée à partir de son début. Cet appel à `getchar` a donc retourné le

caractère « a » qui a été affecté à `c1`. Le deuxième appel à `getchar` a retourné le caractère « b » qui a été affecté à `c2` et le troisième a retourné le caractère « c » qui a été affecté à `c3`. Les trois appels à `putchar` ont donc affiché les caractères « a », « b » et « c ».

Une seconde exécution possible est la suivante :

```
a↵
bc↵
a
b
```

Pour la même raison, le premier appel à `getchar` a donné la main à l'utilisateur qui a saisi les caractères « a » et « nouvelle ligne » qui ont été écrits dans le fichier d'entrée standard à partir de son début. Ce premier appel à `getchar` a donc retourné le caractère « a » qui a été affecté à `c1` et le deuxième a retourné le caractère « nouvelle ligne » qui a été affecté à `c2`. La fin du fichier d'entrée standard a alors été atteinte. Le troisième appel à `getchar` a donc donné la main à l'utilisateur qui a saisi les caractères « b », « c » et « nouvelle ligne » qui ont été écrits dans le fichier standard à partir de son début. Cet appel à `getchar` a donc retourné le caractère « b » qui a été affecté à `c3`. Les trois appels à `putchar` ont donc affiché le caractère « a » suivi d'un retour à la ligne puis le caractère « b ». □

6.1.2 **Ecriture avec format dans le fichier de sortie standard**

Un format d'écriture est une suite de caractères ou de spécifications de conversion qui définit ce qui doit être écrit dans le fichier de sortie standard. Un caractère indique que ce caractère doit être écrit et une spécification de conversion indique qu'une valeur doit être écrite et spécifie la forme sous laquelle elle doit l'être. Par exemple, le format `"Le carré de %d est %d."` indique qu'il faut écrire : « Le carré de » puis un entier, puis « est » puis un entier, puis « . ». Dans ce format `%d` est une spécification de conversion qui spécifie qu'un entier de type `int` doit être écrit comme la suite de ses chiffres, précédée du signe – si cet entier est négatif.

L'écriture avec format dans le fichier de sortie standard est effectuée par la fonction `printf` dont l'appel a la forme suivante :

`printf(`*format*`, `*exp*$_1$`, …, `*exp*$_n$`)`

Le format est une chaîne de caractères qui doit contenir autant de spécifications de conversion qu'il y a de valeurs à écrire. Les valeurs à écrire sont celles des expressions *exp*$_1$, …, *exp*$_n$. Une spécification de conversion spécifie la façon de convertir une valeur en une suite de caractères. La première spécification de conversion correspondant à la première valeur à écrire et ainsi de suite. Cet appel retourne le nombre de valeurs écrites si l'écriture s'est déroulée correctement ou un entier négatif sinon.

Par exemple, en supposant que la variable `x` est de type `int` et a la valeur 12, l'appel :

```
printf("Le carré de %d est %d.", x, x * x);
```

a pour effet d'afficher :

```
Le carré de 12 est 144.
```

Dans ce cours, nous n'utiliserons qu'un sous-ensemble des spécifications de conversion offertes par C, celles qui sont composées :

– du caractère `%` ;

– facultativement suivi du caractère :

  - `-` pour cadrer à gauche du champ (cadrage à droite par défaut) ;
  - `+` pour faire précéder le nombre par son signe si la valeur est un nombre (par défaut, seuls les nombres négatifs sont précédés de leur signe) ;
- facultativement suivi de la largeur du champ (si le nombre de caractères de la valeur à écrire est inférieur à la largeur du champ, cette valeur sera complétée par des espaces à gauche ou à droite selon le type de cadrage, s'il est supérieur la valeur à écrire ne sera pas tronquée) ;
- facultativement suivi d'un point (`.`) suivi d'un nombre qui indique :
  - soit le nombre de chiffres à écrire pour un entier (si le nombre de chiffres à écrire est supérieur au nombre de chiffres de l'entier, celui-ci n'est pas tronqué, s'il est inférieur, l'entier est complété par des zéros à gauche) ;
  - soit le nombre de chiffres après la virgule à écrire pour un flottant (6 par défaut) ;
  - soit le nombre maximum de caractères à écrire pour une chaîne de caractères ;
- obligatoirement suivi d'une ou deux lettres qui indiquent le type de la valeur à écrire et la forme sous laquelle l'écrire :
  - `d` pour écrire un nombre de type `char`, `short` ou `int`, signé ou non signé, sous forme décimale signée,
  - `hd` pour écrire un nombre de type `short`, signé ou non signé, sous forme décimale signée ;
  - `ld` pour écrire un nombre de type `long`, signé ou non signé, sous forme décimale signée ;
  - `f` pour écrire un nombre de type `float` ou `double` en virgule fixe ;
  - `e` pour écrire un nombre de type `float` ou `double` en virgule flottante ;
  - `Lf` pour écrire un nombre de type `long double` en virgule fixe ;
  - `Le` pour écrire un nombre de type `long double` en virgule flottante ;
  - `c` pour écrire un nombre de type `char`, `short` ou `int`, signé ou non signé, sous la forme d'un caractère dont le code est le résultat de la conversion de ce nombre en `unsigned char` ;
  - `s` pour écrire une chaîne de caractères rangée dans un tableau d'éléments de type `char`.

Voyons maintenant comment se déroule l'évaluation d'un appel de la fonction `printf`. Les arguments sont tout d'abord évalués. Le caractère courant du format est le premier du format. L'expression courante est la première, si elle existe, de la liste des expressions. Les actions suivantes sont réalisées tant que la fin du format n'a pas été atteinte et qu'aucune erreur ne s'est produite :

- si le caractère courant du format est différent de `%`, il est écrit dans le fichier de sortie standard, puis on passe au caractère suivant du format ;
- si le caractère courant du format est `%`, la valeur de l'expression courante est convertie en une suite de caractères conformément à la spécification de conversion et écrite dans le fichier de sortie standard, puis on passe au caractère du format qui suit cette spécification de conversion et à l'expression suivante.

Exemple 6.3. L'exécution du bloc :

```
(1) {
(2) int x = 19;
(3) float y = 1982.73;
(4) printf("%s %d %f %e\n", "Bonjour", x, y, y);
(5) printf("x = %d ", x);
(6) printf("y = %.2f\n", y);
```

```
(7)  printf("%+d\n", x);
(8)  printf("%.3d\n", x);
(9)  printf("|%-6s||%6s|\n", "un", "neuf");
(10) }
```

produit l'affichage :

```
(i)   Bonjour 19 1982.729980 1.982730e+03
(ii)  x = 19 y = 1982.73
(iii) +19
(iv)  019
(v)   |un    ||  neuf|
```

Commentons cet affichage :

- l'affichage (i) est produit par l'instruction 4 : la valeur de `y` a été écrite avec 6 chiffres après le point décimal car c'est le nombre de décimales par défaut ;
- l'affichage (ii) est produit par les instructions 5 et 6 : la spécification de conversion `.2f` demande que la valeur de `y` soit écrite avec 2 chiffres après la virgule ;
- l'affichage (iii) est produit par l'instruction 7 : la spécification de conversion `+d` demande que le signe `+` soit écrit devant la valeur de `x` ;
- l'affichage (iv) est produit par l'instruction 8 : la spécification de conversion `.3d` demande que la valeur de `x` soit écrite avec 3 chiffres, ce qui entraîne l'ajout d'un zéro à gauche ;
- l'affichage (v) est produit par l'instruction 9 : les spécifications de conversion `-6s` et `6s` demandent que les chaînes de caractères `"un"` et `"neuf"` soient écrites dans un champ de 6 caractères et cadrées respectivement à gauche et à droite ;
- `\n` (« nouvelle ligne ») à la fin des formats des lignes 4, 6, 7, 8 et 9 entraîne le positionnement du curseur au début de la ligne suivante, par contre l'affichage (ii) produit par les instructions 2 et 3 est réalisé sur la même ligne car le format de la ligne 2 ne se termine pas par `\n`. □

6.1.3 **Lecture avec format dans le fichier d'entrée standard**

Un format de lecture est une suite de caractères ou de spécifications de conversion qui définit ce qui doit être lu dans le fichier d'entrée standard. Un caractère indique que ce caractère doit être lu et une spécification de conversion indique qu'une valeur doit être lue et spécifie la forme sous laquelle elle doit être écrite. Par exemple, le format de lecture `"n = %d"` indique qu'il faut lire « n = » puis un nombre entier composé de son signe (facultatif, si le nombre est positif) suivi de ses chiffres.

La lecture avec format dans le fichier d'entrée standard est réalisée par la fonction `scanf` dont l'appel a la forme suivante :

`scanf(`*format*`, `exp_1`, ..., `exp_n`)`

Le format est une chaîne de caractères qui contient autant de spécifications de conversion qu'il y a de valeurs à lire. Les valeurs lues sont à affecter aux variables dont les adresses sont les valeurs des expressions exp_1, …, exp_n. La première spécification de conversion correspond à la première valeur à lire et ainsi de suite. Une spécification de conversion indique la forme sous laquelle une valeur à lire doit être écrite dans le fichier d'entrée standard. Cet appel retourne le nombre de caractères lus si la lecture s'est déroulée correctement et un entier négatif sinon.

Comme nous le verrons au chapitre 9, il existe un opérateur pour extraire l'adresse d'une variable : c'est l'opérateur `&`. Si *var* est un nom de variable, l'expression `&`*var* a pour valeur l'adresse de la variable *var*. C'est pourquoi l'appel de la fonction `scanf` aura souvent la forme suivante :

`scanf(`*format*`, &`var_1`, ..., &`var_n`)`

Attention ! Une erreur fréquente, qui a des conséquences imprévisibles, est d'oublier l'opérateur `&` devant le nom de la variable dont la valeur est à lire[2]. □

Par exemple, si `n` est une variable de type `int`, l'appel :

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

lit la suite de caractères : « n », « espace », « = », « espace » puis un entier (spécification de conversion `%d`) qui est affecté à la variable `n`.

Dans ce cours, nous n'utiliserons qu'un sous-ensemble des spécifications de conversion offertes par C, celles qui sont composées du caractère `%` suivi d'une ou deux lettres qui indiquent la forme de la valeur à lire et le type de la variable à laquelle elle est affectée :

- `d` pour lire un nombre entier sous forme décimale affecté à une variable de type `int` ;
- `hd` pour lire un nombre entier qui sera affecté à une variable de type `short` ;
- `ld` pour lire un nombre entier qui sera affecté à une variable de type `long` ;
- `f` pour lire un nombre en virgule flottante qui sera affecté à une variable de type `float` ;
- `lf` pour lire un nombre en virgule flottante qui sera affecté à une variable de type `double` ;
- `c` pour un lire un caractère qui sera affecté à une variable de type `char` ;
- `s` pour lire une chaîne de caractères terminée par un caractère « espace », « tabulation » ou « nouvelle ligne » qui ne fait pas partie de cette chaîne, qui sera rangée dans un tableau de variables de type `char`.

Voyons maintenant comment se déroule l'évaluation d'un appel de la fonction `scanf`. Les arguments sont tout d'abord évalués. Le premier caractère du format est lu ainsi que le premier caractère du fichier d'entrée standard qui n'a pas été lu par l'appel précédent d'une fonction de lecture. L'adresse courante est la première de la liste des adresses. Les actions suivantes sont réalisées tant que la fin du format n'est pas atteinte :

- Si le caractère courant du format est différent d'un caractère « espace » ou « tabulation » ou bien du caractère `%` alors ce caractère doit être identique au caractère courant du fichier d'entrée standard. Si c'est le cas, le caractère suivant du format d'entrée est lu ainsi que le caractère suivant du fichier d'entrée standard. Si ce n'est pas le cas, la lecture est interrompue et un entier négatif est retourné.
- Si le caractère courant du format est `%`, alors la spécification de conversion qui suit est lue, puis la suite de caractères du fichier d'entrée standard conforme à cette spécification est lue (les espaces en début de nombre sont sautés) et convertie en une valeur du type de la variable dont l'adresse est l'adresse courante. Si cette conversion réussit, cette valeur est affectée à cette variable. Si cette conversion échoue, la lecture est interrompue et un entier négatif est retourné. Le caractère du format d'entrée qui suit la spécification de conversion

[2] Notons que ce type d'erreur fait l'objet d'un avertissement lorsque l'on utilise l'option `-Wall` de la commande de compilation `gcc`.

est lue ainsi que le caractère suivant du fichier d'entrée standard et l'on passe à l'adresse suivante.

– Si le caractère courant du format est un caractère « espace » ou « tabulation » alors tous les caractères « espace », « tabulation » et « nouvelle ligne » du fichier d'entrée standard ainsi que tous les caractères « espace » et « tabulation » du format sont lus. Le caractère suivant du fichier d'entrée standard est lu ainsi que le caractère suivant du format.

Exemple 6.4. Le bloc d'instructions suivant lit le nom d'un point (1 caractère), son abscisse (1 flottant) et son ordonnée (1 flottant) et les affecte aux variables `nom`, `x` et `y` de types respectifs `char`, `float` et `float` :

```
(1) {
(2) char nom;
(3) float x, y;
(4) scanf("%c%f%f", &nom, &x, &y);
(5) printf("nom = %c, x = %.2f, y = %.2f\n", nom, x, y);
(6) }
```

Une exécution possible de ce bloc pour la saisie d'un point *P* d'abscisse 12,5 et d'ordonnée 7,25 pourra être la suivante :

```
(i)  P 12.5 7.25↵
(ii) nom = P, x = 12.50, y = 7.25
```

Lorsqu'il a la main l'utilisateur saisit le nom, l'abscisse et l'ordonnée du point puis frappe la touche « Entrée » (i). L'instruction 4 affecte le caractère « P » à `nom`, le nombre 12,5 à `x` et le nombre 7,25 à `y`. L'instruction 5 produit l'affichage (ii) et place le curseur au début de la ligne suivante.

Dans la saisie (i) il est possible de faire précéder chaque nombre saisi d'une suite quelconque de caractères « espace » ou « nouvelle ligne ». Par contre, il ne faudra pas le faire pour le nom du point car c'est le premier caractère lu qui sera affecté à la variable `nom`. □

La saisie alternée de nombres et de caractères peut poser des problèmes, comme le montre l'exemple 6.5.

Exemple 6.5. Le bloc d'instructions suivant lit le nom et les coordonnées de deux points et les affecte respectivement à `nom1`, `x1`, `y1` et `nom2`, `x2`, `y2`.

```
(1) {
(2) char nom1, nom2;
(3) float x1 = 0, y1 = 0, x2 = 0, y2 = 0;
(4) scanf("%c%f%f", &nom1, &x1, &y1);
(5) printf("nom = %c, x = %.2f, y = %.2f\n", nom1, x1, y1);
(6) scanf("%c%f%f", &nom2, &x2, &y2);
(7) printf("nom= %c, x= %.2f, y= %.2f\n", nom2, x2, y2);
(8) }
```

Une exécution possible de ce bloc pour la saisie d'un point *P* d'abscisse 12,5 et d'ordonnée 7,25 et d'un point *Q* d'abscisse 87,5 et d'ordonnée 92,75 est la suivante :

```
(i)  P 12.5 7.25↵
(ii) nom = P, x = 12.50, y = 7.25
(iii) Q 87.5 92.75↵
(iv) nom =
(v)  , x =0.00, y=0.00
```

Que s'est-il passé ? Au lieu de lire le caractère « Q » l'instruction 6 a lu le caractère « nouvelle ligne » ajouté au fichier d'entrée standard par la frappe de la touche « Entrée » (i) et l'a affecté à `nom2`, puis elle s'est interrompue, car le caractère Q n'est pas le début d'un nombre flottant. Les valeurs de `x2` et `y2` n'ont donc pas été modifiées. L'instruction 7 a donc affiché « `nom =` » puis la valeur de `nom2` qui, étant le caractère « nouvelle ligne », a eu pour effet de placer le curseur au début de la ligne suivante. Elle a ensuite affiché « , `x = 0.00`, `y= 0.00` » puisque `x2` et `y2` ont été initialisées à 0 à la ligne 3.

Pour éviter ce comportement, une solution est de placer une espace avant la spécification de conversion `%c` dans le format de la ligne 6 qui entraînera la lecture du caractère « nouvelle ligne » :

```
…
(6) scanf(" %c%f%f", &nom2, &x2, &y2);
…
```

Ce qui donnera l'exécution suivante, celle attendue, pour une saisie identique :

```
P 12.5 7.25↵
nom = P, x = 12.50, y = 7.25
Q 87.5 92.75↵
nom = Q, x = 87.50, y = 92.75
```

□

6.2 Lecture et écriture dans des fichiers texte

Un fichier texte est constitué d'une suite d'octets qui sont interprétés comme des caractères (lettres, chiffres, signes de ponctuation…). Les fichiers de texte permettent de sauvegarder des données et d'échanger des données entre programmes. Ceci parce que les caractères sont codés selon un code normalisé : le code ASCII, compréhensible par tous les langages de programmation. Ils sont manipulables par un éditeur de texte et sont imprimables.

Un fichier texte est affecté à une unité d'entrée-sortie : disque, imprimante, clavier, etc.

L'échange de données entre la mémoire d'un programme et un fichier texte se fait au travers d'un tampon (« buffer » en anglais) qui est une zone de la mémoire qui contient une copie d'un morceau du fichier (voir figure 6.1). Quand un programme lit ou écrit une suite de caractères, il le fait dans le tampon. Parallèlement, un processus est mis en œuvre de façon désynchronisée qui transfère, lorsque cela est nécessaire les caractères à lire ou à écrire de l'unité d'E/S vers le tampon ou inversement.

Une opération de lecture ou d'écriture dans un fichier est réalisée à partir de la position courante dans le fichier, que nous appellerons le curseur (voir figure 6.2). Ce curseur est positionné automatiquement lors de l'ouverture du fichier, soit au début, soit à la fin de celui-ci.

Il est déplacé automatiquement lors des opérations de lecture ou d'écriture ou peut l'être à la demande. Un programme peut de plus tester si le curseur est positionné sur la fin du fichier afin de savoir si celle-ci a été atteinte.

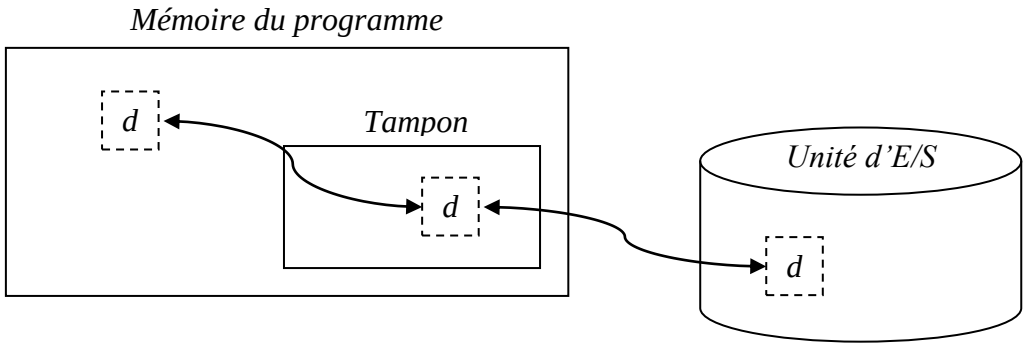


Figure 6.1. Echange de données entre mémoire et fichier

Un fichier possède un descripteur qui est une structure de données contenant un certain nombre d'informations sur ce fichier : sa localisation, les droits d'accès à ce fichier, son mode d'accès, la position du curseur, un indicateur d'erreur, etc. Dans un programme, un fichier est identifié par l'adresse de son descripteur. Cet adresse est de type :

```
FILE *
```

Par exemple :

```
FILE *mon_fichier;
```

déclare une variable `mon_fichier` de type adresse d'un descripteur d'un fichier.

Trois fichiers de texte sont toujours disponibles lors de l'exécution d'un programme :

- le fichier `stdin` qui est le fichier d'entrée standard généralement affecté au clavier du poste de travail ;
- le fichier `stdout` qui est le fichier de sortie standard généralement affecté à l'écran du poste de travail ;
- le fichier `stderr` qui est le fichier d'erreur standard généralement affecté, lui-aussi, à l'écran du poste de travail.

Ces fichiers sont déclarés par :

```
FILE *stdin, *stdout, *stderr;
```

En C, les opérations sur un fichier de texte sont :

- la création : le nom du fichier dont le nom est donné est créé par le système d'exploitation à l'emplacement spécifié par le nom de ce fichier ;
- l'ouverture : le tampon est créé, le curseur est positionné en fonction du mode d'accès et l'adresse du descripteur du fichier est retournée ;
- la fermeture : les informations contenues dans le tampon sont recopiées dans le fichier, puis le tampon est supprimé ;
- les tests de fin de fichier ou d'erreur ;
- la lecture ou l'écriture avec ou sans format : les données échangées peuvent être des caractères ou des lignes.

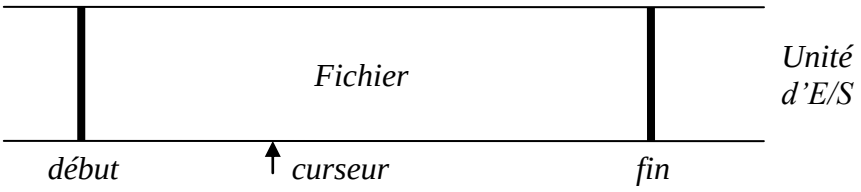


Figure 6.2. *Curseur*

6.2.1 **Création, ouverture et fermeture d'un fichier**

La création d'un fichier et/ou son ouverture est effectuée par l'appel de la fonction `fopen` d'en-tête :

```
FILE *fopen(char *n, char *m)
```

L'appel `fopen(`*n*`,` *m*`)` a pour effet de créer un tampon en mémoire, d'ouvrir après l'avoir éventuellement créé, le fichier *nomfic* selon le mode *m* et de lui associer ce tampon. Cet appel retourne l'adresse du descripteur du fichier si l'ouverture s'est effectuée correctement ou `NULL` sinon. Les modes d'ouverture possibles sont les suivants :

- `"r"` (lecture) : le fichier *n* doit exister, il ne sera accessible qu'en lecture et le curseur est placé en début de fichier ;
- `"w"` (écriture) : si un fichier *n* existe déjà, il est effacé, puis un fichier de nom *n* est créé qui ne sera accessible qu'en écriture et le curseur est placé en début de fichier ;
- `"a"` (ajout) : si un fichier de nom *n* existe déjà, il est effacé, puis un fichier de nom *n* est créé qui ne sera accessible qu'en écriture et le curseur est placé en fin de fichier.

La fermeture d'un fichier est effectuée par l'appel de la fonction `fclose` d'en-tête :

```
int fclose(FILE *f)
```

L'appel `fopen(`*f*`)` a pour effet de transférer le contenu du tampon (on dit : « de vider ») sur l'unité d'entrée/sortie supportant le fichier *f* et de libérer la place occupée par ce tampon. Cet appel retourne 0 si la fermeture s'est effectuée correctement ou sinon un entier ≠ 0.

6.2.2 **Test de fin de fichier ou d'erreur**

Lors de la lecture ou de l'écriture dans un fichier, il peut être nécessaire de savoir si le curseur est positionné en fin de fichier. Ce test peut être réalisé en appelant la fonction `feof` d'en-tête :

```
int feof(FILE *f)
```

L'appel `feof(`*f*`)` retourne un entier ≠ 0 (vrai) si le curseur est positionné sur la fin du fichier *f* ou 0 sinon. Il peut aussi être nécessaire de tester si une lecture ou une écriture s'est effectuée correctement. Ce test peut être réalisé en appelant la fonction `ferror` d'en-tête :

```
int ferror(FILE *f)
```

L'appel `fopen(`*f*`)` retourne la valeur de l'indicateur d'erreur du fichier *f* : un entier ≠ 0 (vrai) si une erreur s'est produite ou 0 sinon.

6.2.3 **Lecture et écriture d'un caractère**

La lecture d'un caractère dans un fichier est effectuée par l'appel de la fonction `fgetc` d'entête :

```
int fgetc(FILE *f)
```

L'appel `fgetc(`*f*`)` a pour effet de lire dans le fichier *f* le caractère dont la position est indiquée par le curseur, puis d'avancer le curseur d'un caractère. Le caractère lu est retourné, sauf si la fin du fichier a été atteinte ou si une erreur s'est produite, auquel cas `EOF` est retourné.

La lecture d'un caractère dans le fichier d'entrée standard peut-être effectuée par l'appel de la fonction `getchar` que nous avons étudiée au paragraphe 6.1.1. L'appel `getchar()` est équivalent à `fgetc(stdin)`.

L'écriture d'un caractère dans un fichier est effectuée par l'appel de la fonction `fputc` d'entête :

```
int fputc(int c, FILE *f)
```

L'appel `fgetc(`*c*`, `*f*`)` a pour effet d'écrire dans le fichier *f* le caractère *c* à la position du curseur puis d'avancer le curseur d'un caractère. Le caractère *c* est retourné, sauf si une erreur s'est produite, auquel cas `EOF` est retourné.

L'écriture d'un caractère dans le fichier standard de sortie peut-être effectuée par l'appel de la fonction `putchar` que nous avons étudiée au paragraphe 6.2.1. L'appel `putchar(c)` est équivalent à `fputc(c, stdin)`.

Exemple 6.6. Le programme suivant crée le fichier `mes_caracteres.txt`, y écrit un par un les caractères saisis par l'utilisateur jusqu'à la frappe de la touche « Entrée » et ferme ce fichier. Il rouvre ensuite ce fichier en lecture, lit un par un les caractères qui y sont enregistrés et les affiche.

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

int main(void)
   {
   FILE *f;
   int c;
   f = fopen("mes-caracteres.txt", "w");
   if (f == NULL)
      {
      printf("Le fichier mes-caracteres.txt ne peut pas être ouvert !");
      exit(1);
      }
   printf("Saisir une suite de caracteres ? ");
   while((c = getchar()) != '\n')
      fputc(c, f);
   fclose(f);
   f = fopen("mes-caracteres.txt", "r");
   if (f == NULL)
      {
      printf("Le fichier mes-caracteres.txt ne peut pas être ouvert !");
      exit(1);
      }
```

```
    printf("Mes caracteres = ");
    while((c = fgetc(f)) != EOF)
       putchar(c);
    fclose(f);
    return 0;
    }
```

□

6.2.4 **Lecture et écriture d'une ligne**

La lecture d'une ligne dans un fichier est effectuée par l'appel de la fonction `fgets` d'en-tête :

```
char *fgets(char *s, int n, FILE *f)
```

L'appel `fgets(`*s*`,` *n*`,` *f*`)` a pour effet de lire dans le fichier *f* à partir de la position du curseur les caractères qui suivent et de les placer dans le tableau de caractères *s* jusqu'à ce que :

- soit $n - 1$ caractères ont été lus ;
- soit le caractère nouvelle ligne a été lu et ajouté à la fin de la chaîne `s ;`
- soit la fin de fichier a été atteinte.

Le caractère de fin de chaîne (caractère nul) est ensuite ajouté à la fin de *s*. L'adresse *s* est retournée, sauf si la fin du fichier a été atteinte sans qu'aucun caractère n'ait été lu ou si une erreur s'est produite, auquel cas `NULL` est retourné.

L'écriture d'une ligne dans un fichier est effectuée par l'appel de la fonction `fputs` d'en-tête :

```
int fputs(char *s, FILE *f)
```

L'appel `fputs(`*s*`,` *f*`)` a pour effet d'écrire dans le fichier *f* à partir de la position du curseur les caractères de la chaîne de caractères *s*. Le caractère de fin de chaîne (caractère de code 0) n'est pas écrit dans le fichier. Un entier ≥ 0 est retourné, sauf si une erreur s'est produite, auquel cas `EOF` est retourné.

Exemple 6.7. Le programme suivant crée le fichier `mon_poeme.txt`, y écrit une suite de vers, puis le ferme. Il rouvre ce fichier en lecture, lit les vers qu'il contient, les range dans le tableau `poeme` et affiche ce tableau. Le tableau `poeme` peut contenir au maximum `NBVERS` ayant au maximum `NBCAR` caractères (un caractère étant réservé pour le caractère de fin de chaîne).

```
#include <stdio.h>
#include <stdlib.h>
#define NBVERS 100
#define NBCAR 100

int main(void)
   {
   FILE *f;
   char poeme[NBVERS][NBCAR + 1]; (il faut compter le caractère de fin de chaîne)
   int i, j;
   f = fopen("mon_poeme.txt", "w");
   if (f == NULL)
      {
      printf("Le fichier mon-poeme.txt ne peut pas être ouvert !");
      exit(1);
      }
```

```
fputs("Souvent pour s'amuser, les hommes d'equipage\n", f);
fputs("Prennent des albatros, vastes oiseaux des mers\n", f);
fputs("Qui suivent, indolents compagnons de voyage,\n", f);
fputs("Le navire glissant sur les gouffres amers\n", f);
fclose(f);
f = fopen("mon_poeme.txt", "r");
if (f == NULL)
   {
   printf("Le fichier mon-poeme.txt ne peut pas être ouvert !");
   exit(1);
   }
i = 0;
while (i < NBVERS && fgets(poeme[i], NBCAR, f))
   i++;
fclose(f);
for (j = 0; j < i; j++)
   printf("%s", poeme[j]);
return 0;
}
```

L'instruction `while` (soulignée) qui lit les vers dans le fichier `mon_poeme` peut-être paraphrasée de la façon suivante : « Tant que le rang du vers à lire (`i`) est inférieur au nombre maximum de vers (`NBVERS`) que l'on peut ranger dans le tableau `poeme` et que ce vers a été correctement lu par la fonction `fgets` et rangé dans la i^e ligne du tableau `poeme`, incrémenter le rang du vers à lire et recommencer.

Cette instruction pourrait être écrite de façon encore plus compacte :

```
while (i < NBVERS && fgets(poeme[i++], NBCAR, f))
   ;
```

mais attention à la lisibilité du programme. □

6.2.5 **Lecture et écriture avec format**

Les opérations de lecture et d'écriture avec format dans un fichier sont réalisées par l'appel des fonctions `fscanf` et `fprintf` dont les fonctions `scanf` et `printf` que nous avons étudiées au paragraphe 6.2 sont des cas particuliers.

La lecture avec format dans un fichier est effectuée par la fonction `fscanf` d'en-tête :

`int fscanf(FILE *f, char *format,` *liste d'expressions de type adresse*`)`

L'appel de cette fonction a le même effet que celui de la fonction `scanf` (voir paragraphe 6.2.2) mais elle lit dans le fichier `f` au lieu de lire dans le fichier d'entrée standard. Cet appel retourne le nombre de variables lues si la lecture s'est bien passée, un entier négatif sinon.

On a :

`scanf(…) ≡ fscanf(stdin, …)`

L'écriture avec format dans un fichier est effectuée par la fonction `fprintf` d'en-tête :

`int fprintf(FILE *f, char *format,` *liste d'expressions*`)`

L'appel de cette fonction a le même effet que celui de la fonction `printf` (voir paragraphe 6.2.1) mais elle écrit dans le fichier `f` au lieu décrire dans le fichier d'entrée standard. Cet appel retourne le nombre de caractères écrits si l'écriture s'est bien passée, un entier négatif sinon.

On a :

```
printf(…) ≡ fprintf(stdout, …)
```

Au lieu d'écrire avec format dans un fichier, on peut le faire dans une chaîne de caractères. L'écriture avec format dans une chaîne de caractères est effectuée par la fonction `sprintf` d'en-tête :

```
int sprintf(char *s, char *format, liste d'expressions)
```

L'appel de cette fonction a le même effet que celui de la fonction `fprintf` mais elle écrit dans la chaîne `s` au lieu d'écrire dans le fichier d'entrée standard. Cet appel retourne le nombre de caractères écrits si l'écriture s'est bien passée, un entier négatif sinon.

L'intérêt de cette fonction est de permettre la conversion de nombres en chaînes de caractères ou l'insertion dans une chaîne de caractères de nombres convertis en chaînes de caractères. Par exemple, si `s` est un tableau d'au moins 3 caractères et `x` est une variable entière de valeur 5, l'appel :

```
sprintf(s, "%d", x * x)
```

rangera la chaîne de caractères : « 25 » dans le tableau `s` et l'appel :

```
sprintf(s, "Le carré de %d est %d.", x, x * x)
```

y rangera la chaîne de caractères : « Le carré de 5 est 25. ».

Signalons que l'opération inverse : la conversion d'une chaîne de caractères en nombre peut être effectuée en utilisant les fonctions `atoi`, `atol` et `atof` de la bibliothèque standard, d'en-têtes respectives :

```
int atoi(char *s)
long atol(char *s)
double atof(char *s)
```

Ces fonctions sont déclarées dans le fichier d'en-têtes `stdlib.h`. Par exemple, l'appel :

```
atoi("25")
```

retournera l'entier 25 de type `int`.

Exercices

Exercice 6.1. On considère une suite de nombres entiers non nuls saisis un par un au clavier. La fin de la suite est marquée par la saisie du nombre 0. Par exemple :

25
137
64
0

Ecrire et tester un programme qui calcule à la volée (c.-à-d. au fur et à mesure de la saisie) :

- la somme des nombres saisis ;
- le plus petit nombre saisi ;
- le plus grand nombre saisi ;

puis affiche ces trois résultats sous la forme :

```
somme = ?
minimum = ?
maximum = ?
```

Si la suite est vide (c.-à-d. si le premier nombre saisi est 0), la somme, le minimum et le maximum devront être égaux à 0.

Exercice 6.2. Ecrire et tester un programme qui demande à l'utilisateur de saisir un nombre entre 1 et 9 sous la forme :

```
Saisir un nombre compris entre 1 et 9 (0 pour terminer) ?
```

puis affiche la table de multiplication par ce nombre des nombres compris entre 1 et 9 et répète cette opération tant que le nombre entré est différent de 0. Si le nombre entré n'est pas compris entre 0 et 9, le programme devra le redemander à l'utilisateur. La table de multiplication devra être affichée sous la forme suivante (où *n* est le nombre saisi) :

```
Table de multiplication par n
1 x n = ..
2 x n = ..
3 x n = ..
4 x n = ..
5 x n = ..
6 x n = ..
7 x n = ..
8 x n = ..
9 x n = ..
```

Les opérandes devront être affichés dans un champ de longueur 1. Les signes `x` et `=` devront chacun être précédé et suivi d'une espace. Le résultat de chaque multiplication devra être affiché dans un champ de longueur 2 et cadré à droite.

Exercice 6.3. Ecrire et tester un programme qui demande à l'utilisateur de saisir un caractère, sous la forme :

```
Saisir un caractere (Entree pour terminer) ?
```

puis l'affiche ainsi que son code ASCII sous la forme :

```
Le code ASCII de c est n
```

et répète cette opération tant que le caractère entré est différent du caractère « nouvelle ligne ». La saisie et l'affichage du caractère devront se faire en utilisant les fonctions `getchar` et `putchar` et l'affichage du code ASCII en utilisant la fonction `printf`.

Exercice 6.4. (Cet exercice nécessite d'avoir étudié le chapitre 9 sur les chaînes de caractères). Il s'agit d'écrire un programme qui crée et interroge une base de données qui décrit les livres d'une bibliothèque et leurs auteurs. Un livre est décrit par sa cote (un numéro qui l'identifie dans la bibliothèque), son titre, son année de publication (un entier positif) et ses auteurs. On décide de stocker ces livres dans deux fichiers :

- le fichier `livres.txt` qui contient une suite de triplets *c t a* (un élément de doublet par ligne) indiquant que le livre de cote *c* a pour titre *t* et a été publié l'année *a*. A chaque livre de la bibliothèque, il correspondra un et un seul triplet dans ce fichier.
- le fichier `auteurs.txt` qui contient une suite de doublets *n c* (un élément de doublet par ligne) indiquant que l'auteur de nom *n* a écrit le livre de cote *c*. A chaque livre de la bibliothèque de cote *c* écrit par *k* ($k \geq 1$) auteurs de noms n_1, …, n_k, il correspondra *k* et seulement *k* doublets dans ce fichier : n_1 *c*, …, n_k *c*.

Supposons par exemple, que la bibliothèque contienne les deux livres :

- *Initiation à l'informatique* de cote L1, écrit par Pierre Legrand et publié en 1995 ;

– *Le langage C mais c'est très simple*, de cote L2 écrit par Anne Dupont et Pierre Legrand et publié en 2004.

Le fichier `livres.txt` aura le contenu suivant :

```
L1
Initiation a l'informatique (les accents sont omis)
1995
L2
Le langage C, mais c'est tres simple
2004
```

et le fichier `auteurs.txt` aura le contenu suivant :

```
Pierre Legrand
L1
Claire Dupont
L2
Pierre Legrand
L2
```

Les identificateurs des fichiers `livres.txt` et `auteurs.txt` seront affectés aux variables globales `flivres` et `fauteurs`.

Les constantes `LCOTE`, `LTITRE`, `LANNEE` et `LNOM` auront pour valeurs respectives le nombre maximum plus un de caractères d'une cote de livre, d'un titre de livre, d'une année et d'un nom d'auteur (il faut compter le nombre caractères de fin de chaîne)..

1. Définir une fonction d'en-tête `void ecrire_livre(char *cote, char *titre, int annee)` qui a pour effet d'écrire à la fin du fichier `flivres` un triplet *cote titre année* (une donnée par ligne). Il faudra vérifier au préalable que les chaînes de caractères `cote` et `titre` ne dépassent les longueurs maximum fixées (`LCOTE` et `LTITRE`) et qu'`annee` est un entier positif. Si une erreur d'écriture est détectée, un message d'erreur sera affiché et l'exécution du programme sera interrompue.
2. Définir une fonction d'en-tête `int ecrire_auteur(char *nom, char *cote)` qui a pour effet d'écrire à la fin du fichier `fauteurs` un doublet *nom cote* (un élément par ligne). Il faudra vérifier au préalable que les chaînes de caractères *nom* et *cote* ne dépassent les longueurs maximum fixées (`LNOM` et `LCOTE`). Si une erreur d'écriture est détectée, un message d'erreur sera affiché et l'exécution du programme sera interrompue.
3. Définir une fonction `int lire_livre(char *cote, char *titre, int *annee)` qui a pour effet de lire dans le fichier `flivres` le prochain triplet *cote titre année*, et qui retourne 0 si la fin du fichier a été atteinte ou si une erreur s'est produite, ou 1 sinon. Cette lecture sera réalisée par la fonction `lire_ligne` (voir chapitre 9, paragraphe 9.4) et l'année convertie en nombre par la fonction `atoi` de la bibliothèque standard.
4. Définir une fonction d'en-tête `int lire_auteur(char *nom, char *cote)` qui a pour effet de lire dans le fichier `fauteurs`, le prochain doublet *nom cote* et qui retourne 0 si la fin du fichier a été atteinte ou si une erreur s'est produite, ou 1 sinon. La lecture des chaînes de caractères *nom* et *cote* pourra être effectuée par la fonction `lire_ligne` (9.2 ci-dessous).
5. Définir une fonction d'en-tête `void livres_publies_apres(int annee_inferieure)` qui a pour effet d'afficher, à raison de un par ligne, les titres des livres publiés après l'année `annee_inferieure`. Cette fonction fera appel à la fonction `lire_livre`. Par exemple, l'appel `livres_publies_apres(2000)` aura pour effet d'afficher :

```
Le langage C, mais c'est tres simple
```

6. Définir une fonction d'en-tête `void auteurs_du_livre(char *titre)` qui a pour effet d'afficher, à raison de un par ligne, les noms des auteurs du livre dont le titre est `titre`. Cette fonction fera appel aux fonctions `lire_livre` et `lire_auteur`. Par exemple, l'appel `auteurs_du_livre("Le langage C, mais c'est très simple")` aura pour effet d'afficher :

```
Claire Dupont
Pierre Legrand
```

7. Définir une fonction `main` qui crée une base de données en utilisant les fonctions `ajouter_livre` et `ajouter_auteur` puis l'interroge en utilisant les fonctions `livre_publies_apres` et `auteur_du_livre`.
8. Tester le programme.

On prendra soin de traiter les erreurs éventuelles de lecture ou d'écriture dans les fichiers.

7
Tableaux, structures et unions

Dans un programme, il est souvent nécessaire de manipuler des données composées d'autres données. Par exemple : un relevé de mesures composé d'une suite de nombres réels ; un point de l'espace composé de son nom : une lettre et de ses coordonnées : deux nombres réels ; une figure géométrique composée d'un ensemble de points. Pour manipuler de telles données, C offre deux structures de données :

- les tableaux composés de variables ou de tableaux de même type ;
- les structures composées de variables ou de tableaux qui peuvent être de types différents.

Un relevé de mesures pourra alors être représenté par un tableau composé de variables de type `float` dont chacune contient une mesure. Un point du plan pourra être représenté par une structure composée d'une variable de type `char` : le nom de ce point et de deux variables de type `float` : l'abscisse et l'ordonnée de ce point. Un ensemble de points pourra être représenté par un tableau de variables dont chacune contient une structure représentant un point.

Il faut aussi pouvoir manipuler des données qui sont de types différents mais appartiennent à un même type générique. Par exemple : un item qui peut être soit un caractère, soit un nombre entier, soit un nombre flottant. Pour cela, C offre une troisième structure de données : les unions.

Dans la suite de ce chapitre, nous étudierons comment définir et manipuler des tableaux (paragraphe 7.1), des structures (paragraphe 7.2) et des unions (paragraphe 7.3) et nous montrerons comment les combiner pour construire des données complexes (paragraphe 7.4). Nous présenterons ensuite trois exemples de programmes qui manipulent de telles données (paragraphe 7.4).

7.1 Tableaux

En C, un tableau est une suite de variables ou de tableaux du même type qui constituent les éléments de ce tableau. Dans le premier cas, ce tableau est monodimensionnel et dans le second cas, il est multidimensionnel.

Attention ! Un tableau n'est pas une valeur : il ne peut pas être affecté à une variable, ni retourné par une fonction. Comme nous le verrons par la suite, c'est l'adresse de son premier élément qui identifie un tableau. □

Par la suite, nous appellerons type de base : un type numérique, un type structure, un type union ou un nom de type introduit par une déclaration `typedef` (voir ci-dessous paragraphe 7.4.1).

7.1.1 **Définition d'un tableau monodimensionnel**

La définition d'un tableau monodimensionnel de variables de type de base *T* a la forme suivante :

T tab[*n*] = {*exp*$_0$, …, *exp*$_m$};

où *tab* est un identificateur, *n* est une expression constante de type entier et *exp*$_0$, …, *exp*$_m$ (*m* ≤ *n*) sont des expressions.

Cette définition crée un tableau monodimensionnel de nom *tab* composé de *n* variables de type *T* : *tab*[0], …, *tab*[*n* – 1]. Ces variables constituent les éléments terminaux du tableau *tab*. Ils sont rangés en mémoire de façon contiguë dans l'ordre suivant :

tab[0], *tab*[1], …, *tab*[*n* – 1]

Les *m* premiers de ces éléments ont pour valeurs initiales respectives les valeurs des expressions *exp*$_0$, …, *exp*$_m$.

La liste des valeurs initiales est facultative. Si elle est omise, la définition d'un tableau se réduit à :

T tab[*n*];

Attention ! Les éléments d'un tableau sont numérotés de 0 à *n* – 1 et non de 1 à *n*. □

Exemple 7.1. Les hauteurs (mesurées en millimètres) de pluie tombées chaque mois d'une année en un lieu donné peuvent être enregistrées dans un tableau `pluie` défini par :

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

Si les hauteurs de pluie sont connues au moment de la définition de ce tableau, elles peuvent être spécifiées comme valeurs initiales. Par exemple :

```
int pluie[12] = {68, 25, 5, 22, 21, 0, 1, 4, 130, 57, 68, 97};
```

Cette définition provoquera la création des variables suivantes dans la mémoire :

pluie[11]	97
pluie[10]	68
pluie[9]	57
pluie[8]	130
pluie[7]	4
pluie[6]	1
pluie[5]	0
pluie[4]	21
pluie[3]	22
pluie[2]	5
pluie[1]	25
pluie[0]	68

□

7.1.2 **Définition d'un tableau multidimensionnel**

Les éléments d'un tableau peuvent être eux-mêmes des tableaux, ce qui permet de construire des tableaux multidimensionnels.

La définition d'un tableau *k*-dimensionnel est une généralisation de celle d'un tableau monodimensionnel. Elle a la forme suivante :

T tab[*n*$_1$][*n*$_2$]…[*n*$_k$] = {*exp*$_0$, …, *exp*$_m$};

où *n*$_1$, … , *n*$_k$ sont des expressions constantes de type entier.

Cette définition crée un tableau de nom *tab* à *k* dimensions tel que :

– le tableau *tab* est composé de n_1 tableaux à $k-1$ dimensions : $tab[0]$, …, $tab[n_1-1]$;

– chaque tableau $tab[i_1]$ $(0 \le i_1 \le n_1 - 1)$ est composé de n_2 tableaux à $k-2$ dimensions : $tab[i_1][0]$, …, $tab[i_1][n_2-1]$;

...

– chaque tableau $tab[i_1]\ldots[i_{k-1}]$ $(0 \le i_1 \le n_1 - 1, \ldots, 0 \le i_{k-1} \le n_{k-1} - 1)$ est composé de n_k variables de type T : $tab[i_1]\ldots[i_{k-1}][0]$, …, $tab[i_1]\ldots[i_{k-1}][n_k-1]$ qui constituent les éléments terminaux du tableau *tab*.

Les éléments terminaux du tableau *tab* sont rangés en mémoire de façon contiguë en faisant varier l'indice le plus interne d'abord :

$tab[0]\ldots[0][0]$, $tab[0]\ldots[0][1]$, …, $tab[0]\ldots[1][0]$, $tab[0]\ldots[1][1]$, …

Les *m* premiers éléments terminaux ont pour valeurs initiales respectives les valeurs des expressions exp_0, …, exp_m. La liste des valeurs initiales est facultative. Si elle est omise, la définition d'un tableau se réduit à :

$T\ tab[n_1][n_2]\ldots[n_k];$

Exemple 7.2. Une matrice *m* à 2 lignes et 3 colonnes peut être représentée par le tableau `m` à 2 dimensions défini par :

```
int m[2][3]
```

Le tableau `m[0]` représente la 1ère ligne de la matrice *m* dont les éléments sont `m[0][0]`, `m[0][1]` et `m[0][2]` et le tableau `m[1]` représente sa 2e ligne dont les éléments sont `m[1][0]`, `m[1][1]` et `m[1][2]`.

Si les éléments de cette matrice sont connus lors de sa définition, ils peuvent être spécifiés comme valeurs initiales. Par exemple, la matrice :

$$m = \begin{bmatrix} 5 & 4 & 3 \\ 6 & 2 & 1 \end{bmatrix}$$

peut être représentée par le tableau `m` défini par :

```
int m[2][3] = {5, 4, 3, 6, 2, 1};
```

ou mieux par :

```
int m[2][3] = {{5, 4, 3}, {6, 2, 1}};
```

en individualisant chaque sous-tableau. L'une ou l'autre de ces deux définitions provoquera la création des variables suivantes dans la mémoire :

`m[1][2]`	1
`m[1][1]`	2
`m[1][0]`	6
`m[0][2]`	3
`m[0][1]`	4
`m[0][0]`	5

□

On peut mêler dans la même définition des variables et des tableaux lorsque ces variables et les éléments terminaux de ces tableaux ont le même type. Par exemple :

```
int mat[2][3], lig[3], elem;
```

définit une matrice `mat` à 2 lignes et 3 colonnes, un tableau `lig` qui pourrait contenir une ligne de la matrice `mat` et une variable `elem` à laquelle pourrait être affectée un élément de la matrice `mat`.

7.1.3 **Expression de type tableau**

Dans une expression, le nom d'un tableau joue le rôle d'une constante de type tableau.

L'expression d'un type tableau en C, est loin d'être naturelle. Pour en faciliter la compréhension, nous la paraphraserons par une description dite « naturelle[1] » ayant la forme suivante : « tableau de *n* éléments de type *T* » où *T* est soit un type de base, soit la description naturelle d'un type tableau, soit la description naturelle d'un type adresse (voir chapitre 8, introduction).

Par exemple, dans une expression le nom `pluie` (voir exemple 7.1) sera interprété comme une constante de type « tableau de 12 éléments de type `int` » et le nom `m` (voir exemple 7.2) comme une constante de type « tableau de 2 éléments de type tableau de 3 éléments de type `int` ».

Attention ! Une expression de type tableau, et donc un nom de tableau, n'est pas une valeur gauche : elle ne peut pas apparaître à gauche du signe d'affectation. Pour le dire plus simplement : un nom de tableau n'est pas un nom de variable ! □

7.1.4 **Sélection de la valeur d'un élément**

La sélection de la valeur d'un élément d'un tableau est réalisée par l'opérateur `[]`.

Si exp_1 est une expression de type « tableau d'éléments de type *T* » et exp_2 est une expression de type entier, alors :

exp_1`[`exp_2`]`

est une expression qui a les propriétés suivantes :

- *type*(exp_1`[`exp_2`]`) = *T*
- *val*(exp_1`[`exp_2`]`) = valeur de l'élément de rang *val*(exp_2) du tableau désigné par exp_1
- elle est une valeur gauche, car un élément terminal d'unn tableau est une variable.

Exemple 7.3. Si `pluie` est le tableau défini dans l'exemple 7.1, la hauteur totale de pluie tombée dans l'année peut être calculée et affichée par l'instruction :

```
{
int i;
h = 0;
for (i = 0 ; i < 12 ; i = i + 1)
   h = h + pluie[i];
printf("Hauteur totale = %d\n", h);
}
```

qui peut aussi s'écrire, en initialisant `h` à 0 dans sa définition et en utilisant les opérateurs `++` et `+=` :

[1] Ce terme est emprunté au livre *C* : *langage, bibliothèque, applications* (InterEditions) d'Henri Garetta. Dans le livre *Le langage C* (Masson) de Brian Kernighan et Denis Ritchie, c'est le terme « description verbale » qui est utilisé.

```
{
int i, h = 0;
for (i = 0 ; i < 12 ; i++)
   h += pluie[i];
printf("Hauteur totale = %d\n", h);
}
```

□

Si exp_1 est une expression de type tableau d'éléments de type T où T est un type tableau et exp_2 est une expression de type entier, l'expression :

exp_1`[`exp_2`]`

a les propriétés suivantes :

- $type(exp_1[exp_2]) = T$
- elle désigne le sous-tableau de rang $val(exp_2)$ du tableau désigné par exp_1 et n'est donc pas une valeur gauche.

Exemple 7.4. Si `m` est la matrice de l'exemple 7.2, la somme des éléments de cette matrice peut-être calculée et affichée par l'instruction :

```
{
int i, j, s = 0;
for (i = 0 ; i < 2 ; i++)
   for (j = 0; j < 3; j++)
      s += m[i][j];
printf("Somme = %d\n", s);
}
```

□

Attention ! Si exp_1 est une expression de type tableau de n éléments, C ne contrôle pas lors de l'évaluation d'une expression exp_1`[`exp_2`]` que la valeur de exp_2 est bien comprise entre 0 et $n - 1$, c.-à-d. qu'il n'y a pas débordement du tableau. Ce contrôle doit donc être effectué par le programme qui manipule ce tableau. □

7.1.5 **Affectation d'une valeur à un élément**

L'affectation d'une valeur à un élément terminal d'un tableau est réalisée par l'opérateur `=` que nous avons déjà étudié (voir chapitre 3, paragraphe 3.2.7) :

Attention ! Une expression de type tableau n'étant pas une valeur gauche, si exp_1 et exp_2 sont de type tableau, on ne pourra pas écrire l'expression t_1 `=` t_2 en pensant affecter le contenu du tableau t_2 au tableau t_1. □

Attention ! Le non contrôle du débordement d'un tableau peut entraîner l'écrasement de la valeur d'une autre variable du programme, si le rang de l'élément auquel est affecté une valeur est supérieur ou égal au nombre d'éléments du tableau. □

Exemple 7.5. Supposons que l'on veuille construire, à partir du tableau `pluie` défini dans l'exemple 7.1, un tableau `pluie_cumulee` tel que `pluie_cumulee[i]` soit égale à la hauteur totale de pluie tombée de janvier jusqu'au `i`^{e} mois inclus. Ce tableau pourra être défini et rempli par l'instruction :

```
{
int pluie_cumulee[12], i;
pluie_cumulee[0] = pluie[0];
for (i = 1; i < 12; i++)
   pluie_cumulee[i] =  pluie_cumulee[i - 1] + pluie[i];
}
```

L'absence d'un contrôle de débordement fait que si par erreur, on avait affecté à `i` une valeur supérieure à 12, aucune erreur n'aurait été détectée, ce qui aurait entraîné une exécution imprévisible du programme.

□

Exemple 7.6. Si `m1` et `m2` sont deux tableaux définis par :

```
int m1[2][3], m2[2][3];
```

l'instruction suivante a pour effet de recopier le contenu du tableau `m1` dans le tableau `m2` :

```
for (i = 0; i < 2; i++)
   for (j = 0; j < 3; j++)
      m2[i][j] = m1[i][j];
```

Il n'est pas possible de remplacer cette instruction par :

```
m2 = m1;
```

en pensant recopier globalement le contenu du tableau `t2` dans le tableau `t1`, ou par :

```
for (i = 0; i < 10; i = i + 1)
   m2[i] = m1[i];
```

en pensant recopier globalement le contenu de chaque ligne de `m1` dans la ligne correspondante de `m2`. Ceci, parce que les expressions `m2` et `m2[i]` ne sont pas des valeurs gauche. □

7.2 Structures

Une structure est une valeur composée d'une suite de variables ou de tableaux qui constituent les champs de cette structure. Contrairement aux éléments d'un tableau, les champs d'une structure peuvent être de types différents. Toute structure a un type formé des déclarations de ses champs.

7.2.1 **Déclaration d'un type structure**

La déclaration :

```
struct s
   {
   decl1;
   …
   decln;
   };
```

où *s* est un identificateur : le nom de la structure (ou son étiquette) et $decl_1$, …, $decl_n$ sont les déclarations des variables ou des tableaux qui constituent les champs des structures de ce type, déclare un nouveau type : le type `struct` *s* qui est un type structure.

Les types structure sont des types de base au même titre que les types numériques.

Exemple 7.7. Un point du plan décrit par son nom (une lettre), son abscisse et son ordonnée (des réels) dans un repère orthonormé, pourra être représenté par une structure de type :

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

Dans les exemples de ce chapitre, on considérera que les points sont des instances du type `struct point`. □

Un type structure ne peut pas avoir deux champs de même nom, par contre, deux types structure différents peuvent avoir deux champs de même nom.

7.2.2 **Définition de variables de type structure**

La définition d'une variable de type structure a la forme suivante :

`struct` *s var* `=` {*exp*$_1$, …, *exp*$_m$}`;`

où *s* est le nom d'une structure, *var* est un identificateur et *exp*$_1$,…, *exp*$_m$ ($m \leq$ nombre de champs des structures de type `struct` *s*) sont des expressions.

Si *s* est le nom d'une structure de champs c_1 de type T_1, ..., c_n de type T_n, cette définition crée une variable *var* de type `struct` *s* dont la valeur est une structure composée de *n* champs de noms respectifs *var*.c_1, ..., *var*.c_n et de types respectifs T_1, ..., T_n.

La valeur initiale de cette variable est une structure dont les *m* premiers champs ont pour valeurs initiales respectives les valeurs des expressions *exp*$_0$, …, *exp*$_m$. La liste des valeurs initiales est facultative. Si elle est omise, la définition d'une variable de type structure se réduira à :

`struct` *s var*`;`

Exemple 7.8. La définition :

```
struct point p = {'A', 1.5, 2.5};
```

provoquera la création des variables suivantes dans la mémoire :

p	p.y	2,5
	p.x	1,5
	p.nom	65

(code ASCII de la lettre A)

Cette représentation (imbrication des cases mémoires) fait clairement apparaître que `p` est une variable composée de 3 variables : `p.nom`, `p.x` et `p.y`. □

Comme pour les variables numériques, on peut définir plusieurs variables dans la même définition. Par exemple :

```
struct point p1, p2;
```

On peut regrouper la déclaration d'un type structure avec la déclaration de variables de ce type. Par exemple :

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

On peut aussi déclarer des variables ayant un type structure anonyme. Par exemple :

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

7.2.3 **Sélection de la valeur d'un champ**

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

Si *exp* est une valeur gauche de type structure ayant un champ *c* qui est une variable de type *T*, alors :

exp.*c*

est une expression qui a les propriétés suivantes :

- *type*(*exp*.*c*) = *T*
- *val*(*exp*.*c*) = valeur du champ *c* de la structure *val*(*exp*)
- elle est une valeur gauche.

Si *exp* est une valeur gauche de type structure ayant un champ *c* dont la déclaration est celle d'un tableau de type *T*, alors :

exp.*c*

est une expression qui a la propriété suivante :

- *type*(*exp*.`c`) = *T* désigne un tableau de type *T* et n'est donc pas une valeur gauche

Exemple 7.9. L'instruction suivante définit deux variables `a` et `b` de type `struct point`, représentant respectivement un point *A* de coordonnées (3 ; 2), un point *B* de coordonnées (7 ; 4). Elle calcule et affiche les coordonnées du milieu du segment *AB*.

```
{
struct point a = {'A', 3, 2}, b = {'B', 7, 4};
printf("Coordonnees du milieu du segment AB = (%.2f ; %.2f)",
       (a.x + b.x) / 2, (a.y + b.y) / 2);
}
```

L'exécution de ce bloc d'instructions produira l'affichage :

```
Coordonnees du milieu du segment AB = (5.00 ; 3.00)
```

□

7.2.4 **Affectation d'une valeur à un champ**

L'affectation d'une valeur à un champ d'une structure et d'une structure à une variable de type structure est réalisée par l'opérateur = (voir chapitre 3, paragraphe 3.2.7). Notons, que contrairement aux tableaux, les structures sont des valeurs et peuvent donc être affectées à des variables ou passées en arguments d'une fonction.

Attention ! Une structure ne peut être affectée qu'à une variable de même type que cette structure. Il n'y a pas de conversion automatique. □

Exemple 7.10. La fonction suivante retourne le point milieu du segment dont les points extrémités `a` et `b` lui sont passés en argument et affecte à ce point le nom `nom` lui-même passé en argument :

```
struct point milieu(char nom, struct point a, struct point b)
   {
   struct point m;
   m.nom = nom;
   m.x = (a.x + b.x) / 2;
   m.y = (a.y + b.y) / 2;
   return m;
   }
```

On notera qu'il n'y a pas conflit de nom entre la variable `nom` et le champ `nom`, car ils n'appartiennent pas à la même catégorie de noms (voir paragraphe 5.3.1). □

7.3 Unions

Une union est une valeur composée d'un ensemble de variables ou de tableaux qui constituent les champs de cette union et qui ont la même adresse. Cette définition peut sembler un peu obscure, nous allons donc l'éclairer au travers d'un exemple (exemple 7.11).

Exemple 7.11. Supposons que nous voulions définir des items dont chacun est soit un caractère, soit un nombre entier, soit un nombre flottant. En C, un tel item pourra être représenté par une union dont le type est déclaré de la façon suivante :

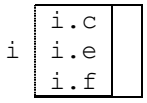
```
union item
   {
   char c ;
   int e;
   float f;
   };
```

Cette déclaration spécifie qu'une variable `i` de type `union item` est composée de trois variables `i.c` de type `char`, `i.e` de type `int` et `i.f` de type `float` qui constituent ses champs. Si la variable `i` a pour valeur un caractère, il sera affecté au champ `c`, si elle a pour valeur un entier, il sera affecté au champ `e` et si elle a pour valeur un flottant, il sera affecté au champ `f`. Puisqu'une variable de type `union item` ne peut avoir que l'une de ces trois valeurs, il est possible d'implanter ses champs à partir de la même adresse et c'est ce qui est fait.

Une variable `i` de type `union item` sera définie par :

```
union item i;
```

Cette définition provoquera la création des variables suivantes dans la mémoire :

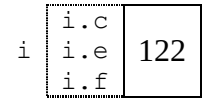


Cette représentation fait clairement apparaître que `i` est une variable composée des 3 variables : `i.c`, `i.e` et `i.f` qui ont la même adresse.

Si la taille d'une valeur de type `char` est de 1 octet et celle d'une valeur de type `int` ou `float` est de 4 octets, la taille de la case mémoire d'une valeur de type `union item` sera de 4 octets.

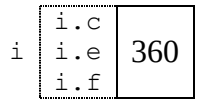
Si l'on souhaite affecter à la variable `i` :

– le caractère « z », on l'affectera au champ `c` : `i.c = 'z'` :

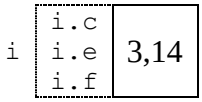


(122 est le code ASCII de « z »)

– la valeur 360, on l'affectera au champ `i` : `i.e = 360` :



– la valeur 3,14, on l'affectera au champ `f` : `i.f = 3.14`



□

Les unions sont définies et manipulées de la même façon que les structures.

Un type union est déclaré de la façon suivante :

```
union u
   {
   decl1;
   …
   decln;
   };
```

où *u* est un identificateur : le nom de l'union (ou son étiquette) et $decl_1$, …, $decl_n$ sont les déclarations des variables ou des tableaux qui constituent les champs des unions de ce type.

La définition d'une variable de type union a la forme suivante :

```
union u var;
```

où *u* est le nom d'une union et *var* est un identificateur. Si *u* est le nom d'une union de champs c_1 de type T_1, .., c_n de type T_n, cette définition crée une variable *var* de type `union u` dont la valeur est une union composée de *n* champs de noms respectifs $var.c_1$, .., $var.c_n$ et de types respectifs T_1, .., T_n.

De façon similaire aux types structure :

– un type union ne peut pas avoir deux champs de même nom, par contre, deux types union différents peuvent avoir deux champs de même nom ;
– on peut définir plusieurs variables dans la même définition, regrouper la déclaration d'un type union avec la définition de variables de ce type ou définir une variable de type union anonyme.

Les opérateurs de sélection et d'affectation opèrent sur les unions de la même façon que sur les structures. Il faut cependant faire attention à l'effet produit par le recouvrement des cases mémoires affectées à chaque champ. Par exemple, si `i` est la variable de type `union item` définie dans l'exemple 7.11 et que la valeur 15,6 a été affectée au champ `i.f`, la valeur de l'expression `i.c` sera égale au caractère dont la représentation binaire est celle du premier octet de la représentation binaire du flottant 15,6.

Pour pouvoir manipuler une valeur de type union, il est nécessaire de connaître son type effectif au moment où elle est accédée. Or en C, il n'existe pas d'opérateur permettant de connaître le type d'une valeur ou de le tester. Il faut donc trouver un moyen de coder ce type. Une solution classique consiste à inclure une valeur de type union dans une structure à deux champs dont le premier contient le type de cette valeur et le second la valeur elle-même, comme le montre l'exemple 7.12.

Exemple 7.12. Reprenons les items (caractère, entier ou flottant) de l'exemple 7.11. Ils pourront être représentés par des structures de type `struct item` est déclaré de la façon suivante :

```
struct item
   {
   char type;
   union
      {
      char c;
      int e;
      float f;
      } u;
   };
```

Si l'item a pour valeur un caractère *c*, le champ `type` aura pour valeur le caractère « C » et le champ `u.c` aura la valeur *c*. Si l'item est un nombre entier *e*, le champ `type` aura pour valeur le caractère « E » et le champ `u.e` aura la valeur *e*. Si l'item est un nombre flottant *f*, le champ `type` aura pour valeur le caractère « F » et le champ `u.f` aura la valeur *f*.

L'affichage d'un item pourra être réalisé en lui appliquant la fonction `afficher_item` dont la définition est la suivante :

```
void afficher_item(struct item i)
   {
   switch (i.type)
      {
      case 'C':
         printf("%c\n", i.u.c);
         break;
      case 'E':
         printf("%d\n", i.u.e);
         break;
      case 'F':
         printf("%f\n", i.u.f);
         break;
      }
   }
```

□

7.4 Données complexes

En imbriquant tableaux, structures et unions, on peut construire des valeurs ou des tableaux aussi complexes que nécessaire pour traiter une application, comme le montreront les exemples du paragraphe 7.5. Auparavant, afin de faciliter la déclaration des types de ces valeurs complexes et de les rendre plus lisibles, il est utile d'apprendre à donner des noms à des expressions de type. C'est l'objet du paragraphe 7.4.1.

7.4.1 Nommage d'une expression de type

Afin de rendre plus lisibles les déclarations de types complexes, il est possible de donner un nom à certaines expressions de type.

Pour donner le nom *N* (un identificateur) à une expression de type *t*, il faut faire la déclaration suivante :

`typedef` *decl*

où *decl* est la déclaration d'une variable de type *t* dans laquelle le nom de la variable est remplacé par *N*.

Par convention, nous écrirons la première lettre d'un nom de type en majuscule et les suivantes en minuscules. Par exemple, les deux déclarations suivantes :

```
typedef struct point Point;
typedef Point Triangle[3];
```

déclarent que `Point` est un synonyme de l'expression de type `struct point` et que `Triangle` est un synonyme de l'expression de type « tableau de 3 variables de type `Point` » et donc de l'expression de type « tableau de 3 variables de type `struct point` ».

Ainsi, une variable `p` dont les instances sont des structures de type `struct point` pourra être déclarée indifféremment `struct point p` ou `Point p`. De même un tableau `t` de 3 variables de type `struct point` pourra être déclaré indifféremment : `struct point t[3]`, `Point t[3]` ou `Triangle t`.

Attention ! Contrairement à ce que son nom pourrait laisser croire, une déclaration `typedef` n'est pas la définition d'un nouveau type. Si c'était le cas, une valeur de type `struct point` et une valeur de type `Point` ne seraient pas de même type et il ne serait donc pas possible, par exemple, d'affecter une valeur de type `POINT` à une variable de type `struct point`. □

7.4.2 **Exemples**

Exemple 7.13. Les vacances scolaires peuvent être représentées par les deux types structures suivants :

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

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

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.

La définition :

```
struct vacances v = {2, 'B', {22, 12, 2007}, {6, 1, 2008}};
```

provoquera la création des variables suivantes dans la mémoire :

v	v.fin	v.fin.annee	2008
		v.fin.mois	1
		v.fin.jour	6
	v.debut	v.debut.annee	2007
		v.debut.mois	12
		v.debut.jour	22
		v.zone	66
		v.periode	2

(code ASCII de B)

Une période de vacances scolaires (Noël, Hiver, Printemps, Eté, Toussaint) pourra être affectée à la variable `v` définie par :

```
struct vacances v;
```

Les vacances de l'année scolaire 2007-2008, toutes zones confondues, pourront être enregistrées dans le tableau `vacances_2007_2008` défini par :

```
struct vacances vacances_2007_2008[15];
```

Ce tableau a 15 éléments, car il y a 5 périodes de vacances pour chacune des 3 zones.

On aurait pu aussi déclarer ces structures et ces variables en utilisant des types nommés :

```
typedef struct
   {
   int jour, mois, annee;
   } Date;

typedef struct
   {
   char periode;
   char zone;
   Date debut;
   Date fin;
   } Vacances;

Vacances v;

Vacances vacances_2007_2008[15];
```

□

Exemple 7.14. Un relevé `r` sur lequel sont notés les hauteurs de pluie tombée chaque mois d'une année en un lieu donné peut être représenté par la variable `r` définie par :

```
struct releve
   {
   int annee;
   int hauteurs[12];
   } r;
```

Cette définition provoquera la création des variables suivantes dans la mémoire :

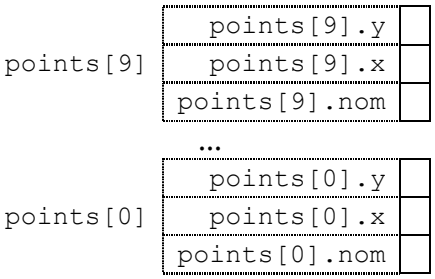
	r.hauteurs[11]	
	...	
r	r.hauteurs[1]	
	r.hauteurs[0]	
	r.annee	

□

Exemple 7.15. Un ensemble de 10 points d'un plan peut être représenté par le tableau `points` défini par :

```
struct point points[10]
```

Cette définition provoquera la création des variables suivantes dans la mémoire :

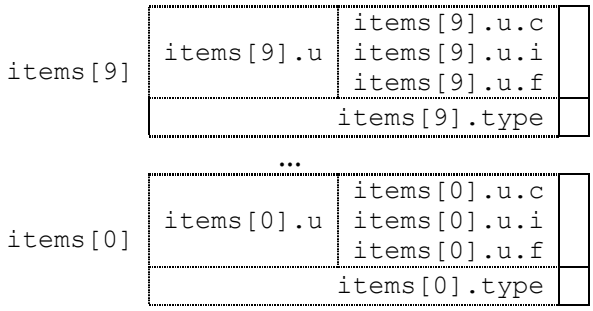


□

Exemple 7.16. Un ensemble de 10 items dont chacun peut être un caractère, un entier ou un flottant (voir exemple 7.12) peut être représenté par le tableau `items` défini par :

```
struct item items[10]
```

Cette définition provoquera la création des variables suivantes dans la mémoire :



7.5 Exemples

Nous présentons dans ce paragraphe quatre programmes qui manipulent des tableaux, des structures et des unions.

7.5.1 Statistiques sur les notes d'un examen

Le programme suivant calcule les notes moyenne, minimum et maximum obtenues lors d'un examen. Voici le texte de ce programme :

```
#include <stdio.h>

int main(void)
   {
   float notes[12] =
      {12, 5, 15.5, 11, 13, 8.75, 14, 6, 16, 10.25, 2, 18};
   float somme, min, max;
   int i;
```

```
/*
 * Calcul des notes moyenne, minimum et maximum
 */
somme = notes[0];
min = notes[0];
max = notes[0];
for (i = 1; i < 12; i++)
   {
   somme += notes[i];
   if (notes[i] > max)
      max = notes[i];
   if (notes[i] < min)
      min = notes[i];
   }
/*
 * Affichage des notes moyenne, minimum et maximum
 */
printf("Note moyenne = %.2f\n", somme / 12);
printf("Note minimum = %.2f\n", min);
printf("Note maximum = %.2f\n", max);
return 0;
}
```

On suppose que cet examen a été passé par 12 étudiants identifiés chacun par un numéro variant de 0 à 11. Les notes obtenues par ces étudiants sont stockées comme valeurs initiales dans un tableau `notes` de 12 éléments tel que la valeur de l'élément `notes[i]` est la note obtenue par l'étudiant `i`.

Les variables `somme`, `min` et `max` contiennent respectivement la somme des notes, la note minimum et la note maximum. Elles sont initialisées à la note de l'étudiant de numéro 0. On parcourt ensuite le tableau `notes` à partir du rang 1. La note courante (`notes[i]`) est cumulée dans la variable `somme`. Si cette note courante est supérieure à la note maximum déjà calculée, elle devient la nouvelle note maximum et si elle est inférieure à la note minimum déjà calculée, elle devient la nouvelle note minimum. La note moyenne est égale à la somme des notes divisée par 12 (le nombre d'étudiants).

On notera l'utilisation de l'instruction `for` pour balayer les éléments du tableau `notes` : le balayage commence par l'élément de rang 1 (`i = 1`), se poursuit tant que ce rang (`i`) est inférieur à la taille du tableau (`i < 12`) puisque l'élément de plus haut rang d'un tableau de n éléments est $n - 1$. A chaque passage dans la boucle, le rang est incrémenté de 1 (`i++`).

L'exécution de ce programme produira l'affichage :

```
Note moyenne = 10.96
Note minimum = 2.00
Note maximum = 18.00
```

7.5.2 **Calcul des notes d'un examen**

Occupons nous maintenant de la saisie, du calcul et de l'affichage des notes de l'examen traité au paragraphe précédent (paragraphe 7.5.1). Nous supposerons que cet examen était constitué d'une épreuve écrite affectée du coefficient 2 et d'une épreuve de travaux pratiques (TP) affectée du coefficient 1. La note finale de l'examen est donc égale à :

$$\frac{2 \times note\ écrit + note\ TP}{3}$$

Nous allons écrire un programme qui demande la saisie des notes obtenues par chaque étudiant aux différentes épreuves de l'examen, calcule la note finale de chaque étudiant en tenant compte des coefficients de chaque épreuve et affiche l'ensemble des notes.

Les notes sont stockées dans un tableau à deux dimensions défini par :

```
float notes[12][3];
```

La note d'écrit de l'étudiant `i` est affectée à l'élément `notes[i][0]`, sa note de travaux pratiques à l'élément `notes[i][1]` et sa note finale à l'élément `notes[i][2]`.

Les résultats de l'examen sont affichés sous la forme suivante :

```
Etudiant  Ecrit    TP Examen
       0  11.00 14.00  12.00
       1   2.50 10.00   5.00
       3  18.00 10.50  15.50
…
```

Voici le texte de ce programme :

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

float notes[12][3];

float saisie_note(void)
   {
   float note;
   do
      {
      printf("note entre 0 et 20 ? ");
      scanf("%f", &note);
      }
   while (note < 0 || note > 20);
   return note;
   }

int main(void)
   {
   int i;
   printf("Saisie\n");
   for (i = 0; i < 12; i++)
      {
      printf("Etudiant %d\n", i);
      printf("Ecrit\n");
      notes[i][0] = saisie_note();
      printf("TP\n");
      notes[i][1] = saisie_note();
      notes[i][2] = (2 * notes[i][0] + notes[i][1]) / 3;
      }
   printf("Resultats\n");
   printf("%8s%6s%6s%7s\n", "Etudiant", "Ecrit", "TP", "Examen");
   for (i = 0; i < 12; i++)
      printf("%8d%6.2f%6.2f%7.2f\n",
             i, notes[i][0], notes[i][1], notes[i][2]);
   return 0;
   }
```

Ce programme est composé de deux fonctions : `saisie_note` et `main`.

La fonction `saisie_note` demande à l'utilisateur de saisir une note entre 0 et 20 et réitère cette demande jusqu'à ce que la note saisie respecte cette contrainte. Cette note est alors retournée.

La fonction `main` comporte deux parties : (i) saisie des notes d'écrit et de TP et calcul de la note finale de chaque étudiant, (ii) affichage du tableau des notes.

L'affichage du tableau des notes nécessite quelques explications. Comme indiqué ci-dessus, cet affichage est fait sur quatre colonnes. La 1ère colonne, intitulée « Etudiant » contient les numéros des étudiants, la 2e intitulée « Ecrit » contient les notes d'écrit, la 3e intitulée « TP » contient les notes de TP et la 4e intitulée « Examen » contient les notes finales de l'examen.

Il faut tout d'abord déterminer la largeur de ces colonnes. Compte tenu du fait qu'un numéro d'étudiant a 2 chiffres au maximum (0 à 11), qu'une note est écrite avec 5 caractères (2 pour la partie entière, 1 pour le point et 2 pour la partie fractionnaire et que les intitulés ou les nombres d'une même ligne doivent être séparés par au moins une espace, les largeurs respectives de ces colonnes en nombre de caractères sont : 8, 6, 6 et 7.

D'où le format de la ligne d'intitulés : `"%8s%6s%6s%7s\n"` et celui d'une ligne de notes `"%8d%6.2f%6.2f%7.2f\n"`. Rappelons (voir chapitre 6, paragraphe 6.1.2) qu'une spécification de conversion qui commence par un nombre non signé *i* indique que la donnée correspondante doit être écrite cadrée à droite dans un champ de *i* caractères.

Exécutons ce programme :

```
Saisie
Etudiant 0
Ecrit
note entre 0 et 20 ? 78
note entre 0 et 20 ? 11
TP
note entre 0 et 20 ? 14
Etudiant 1
Ecrit
note entre 0 et 20 ? 2.5
TP
note entre 0 et 20 ? 10
Etudiant 2
Ecrit
note entre 0 et 20 ? 18
TP
note entre 0 et 20 ? 10.5
…
Resultats
Etudiant Ecrit    TP Examen
       0 11.00 14.00  12.00
       1  2.50 10.00   5.00
       2 18.00 10.50  15.50
…
```

7.5.3 **Barycentre d'un ensemble de points**

Il s'agit d'écrire un programme qui calcule puis affiche les coordonnées et la masse du barycentre d'un ensemble de points massiques (c.-à-d. affectés d'une masse) de l'espace, saisis par l'utilisateur du programme.

Rappelons tout d'abord que le barycentre d'un ensemble de points A_1, …, A_n affectés des masses m_1, …, m_n est le point G tel que :

$$\sum_{i=1}^{n} m_i \overrightarrow{GA_i} = \vec{0}$$

Pour tout point *O* de l'espace, on a :

$$\overrightarrow{OG} = \frac{\sum_{i=1}^{n} \overrightarrow{OA_i}}{\sum_{i=1}^{n} m_i}$$

Les coordonnées spatiales du point *G*, dans un repère cartésien sont donc les suivantes :

$$x_G = \frac{\sum_{i=1}^{n} m_i x_i}{\sum_{i=1}^{n} m_i} \quad y_G = \frac{\sum_{i=1}^{n} m_i y_i}{\sum_{i=1}^{n} m_i} \quad z_G = \frac{\sum_{i=1}^{n} m_i z_i}{\sum_{i=1}^{n} m_i}$$

Le texte du programme est le suivant :

```
#include <stdio.h>
#include <stdlib.h>
#define N 100

struct point
   {
   char nom;
   float x, y, z, m;
   };

struct point points[N], g;

int nb_points, i;

void saisie_points(void)
   {
   float m;
   do
      {
      printf("Nb de points (entre 1 et %d) ? ", N);
      scanf("%d", &nb_points);
      }
   while(nb_points < 1 || nb_points > N);
   for (i = 0; i < nb_points; i++)
      {
      printf("Point %d\n", i);
      printf("x y z ? ");
      scanf("%f %f %f", &points[i].x, &points[i].y, &points[i].z);
      do
         {
         printf("m (m > 0) ? ");
         scanf("%f", &m);
         }
      while (m <= 0);
      points[i].m = m;
      }
   }
```

```
void calcul_barycentre(void)
   {
   g.m = g.x = g.y = g.z = 0;
   for (i = 0; i < nb_points; i++)
      {
      g.m += points[i].m;
      g.x += points[i].m * points[i].x;
      g.y += points[i].m * points[i].y;
      g.z += points[i].m * points[i].z;
      }
   g.x /= g.m;
   g.y /= g.m;
   g.z /= g.m;
   }

void affichage_barycentre(void)
   {
   printf("BARYCENTRE\nx = %.3f\ny = %.3f\nz = %.3f\nm = %.3f",
          g.x, g.y, g.z, g.m);
   }

int main(void)
   {
   saisie_points();
   calcul_barycentre();
   affichage_barycentre();
   return 0;
   }
```

Ce programme est constitué de quatre fonctions :

- la fonction `saisie_points` qui demande à l'utilisateur de saisir le nombre de points puis les coordonnées et la masse de chaque point (lorsqu'une donnée saisie est incorrecte, sa saisie est redemandée) ;
- la fonction `calcul_barycentre` qui calcule les coordonnées et la masse du barycentre ;
- la fonction `affichage_barycentre` qui affiche les coordonnées et la masse du barycentre ;
- la fonction `main` qui appelle successivement les trois fonctions précédentes.

Un point massique est représenté par une structure de type :

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

dont le champ `nom` contient le nom du point et dont les champs `x`, `y`, `z` et `m` représentent respectivement les coordonnées et la masse de ce point.

L'ensemble des points est stocké dans le tableau `points` dont le nombre d'éléments doit être au moins égal à la cardinalité maximum des ensembles de points qui seront traités par le programme. Dans ce programme, nous avons paramétré cette cardinalité maximum par la constante symbolique `N` définie par une macro. Le tableau `points` est donc défini par :

```
struct point points[N];
```

Le nombre de points (`nb_points`) dont le barycentre est à calculer est demandé à l'utilisateur et affecté à la variable `n`. Le programme vérifie que le nombre de points saisi est compris

(i)

```
Nb de points (entre 1 et 100) ? 2↵
Point 0
x y z ? 0 0 0↵
m (m > 0) ? 1↵
Point 1
x y z ? 3 0 0↵
m (m > 0) ? 0↵
m (m > 0) ? 2↵
BARYCENTRE
x = 2.000
y = 0.000
z = 0.000
m = 3.000
```

(ii)

```
Nb de points (entre 1 et 100) ? 8↵
Point 0
x y z ? 0 0 0↵
m (m > 0) ? 0.5↵
Point 1
x y z ? 1 0 0↵
m (m > 0) ? 0.5↵
...
Point 7
x y z ? 0 1 1↵
m (m > 0) ? 0.5↵
BARYCENTRE
x = 0.500
y = 0.500
z = 0.500
m = 4.000
```

Figure 7.1. *Deux exécutions du programme Barycentre*

entre 1 et `N`. Si ce n'est pas le cas, un message est affiché et l'exécution du programme est interrompue.

Le barycentre est affecté à la variable `g` de type `struct point`.

La figure 7.1 rassemble deux exécutions de ce programme : (i) barycentre de deux points dont l'un a une masse double de l'autre, (ii) barycentre de huit points de même masse placés aux sommets d'un cube.

7.5.4 **Equilibrage d'un mobile décoratif**

On considère un mobile décoratif simple[2] composé d'un fléau horizontal ayant une figure circulaire, rectangulaire ou triangulaire accrochée à chacune de ses extrémités, comme le montre la figure 7.2. On suppose que les fils auxquels sont accrochées les figures ont un poids négligeable devant celui d'une figure. On suppose de plus que les figures sont découpées dans la même matière (du carton, par exemple) de densité uniforme 1. On considérera donc que le rapport des poids de deux figures est égal au rapport de leurs aires.

Il s'agit de calculer la position du point d'attache du fléau au fil qui le suspend, de façon à ce qu'il reste horizontal, connaissant les dimensions des figures suspendues à ses extrémités.

La position du point d'attache sera définie par le rapport $d_1 / (d_1 + d_2)$ qui mesure sa distance à partir de l'extrémité gauche du fléau en pourcentage de la longueur du fléau, (voir figure ci-dessus). Compte tenu de nos hypothèses sur la réalisation des figures, on a :

$$d_1 / (d_1 + d_2) = poids(f_1) / (poids(f_1) + poids(f_2)) = aire(f_1) / (aire(f_1) + aire(f_2))$$

[2] Extrait d'un exercice proposé dans le livre *Approches impératives et fonctionnelle de l'algorithmique* (Springer) de Sébastien Veigneau.

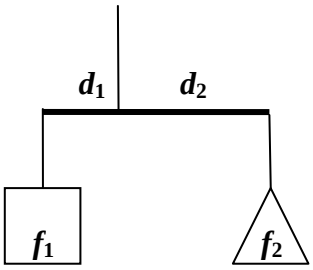


Figure 7.2. *Un mobile décoratif*

Le texte du programme est le suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14

struct figure
   {
   char type;
   union
      {
      struct {float rayon;} c;
      struct {float longueur, largeur;} r;
      struct {float base, hauteur;} t;
      } u;
   };

float saisie_dimension(void)
   {
   float v;
   do
      {
      printf("valeur ? ");
      scanf(" %f", &v);
      }
   while (v <= 0);
   return v;
   }

struct figure saisie_figure(void)
   {
   struct figure f;
   do
      {
      printf("Forme (c, r, t) ? ");
      scanf(" %c", &f.type);
      }
   while (f.type != 'c' && f.type != 'r' && f.type != 't');
   switch (f.type)
      {
      case 'c':
         printf("Rayon\n");
         f.u.c.rayon = saisie_dimension();
         break;
```

```
        case 'r':
           printf("Longueur\n");
           f.u.r.longueur = saisie_dimension();
           printf("Largeur\n");
           f.u.r.largeur = saisie_dimension();
           break;
        case 't':
           printf("Base\n");
           f.u.t.base = saisie_dimension();
           printf("Hauteur\n");
           f.u.t.hauteur = saisie_dimension();
           break;
        }
     return f;
     }

  float aire_figure(struct figure f)
     {
     float aire;
     switch (f.type)
        {
        case 'c':
           aire = PI * pow(f.u.c.rayon, 2);
           break;
        case 'r':
           aire = f.u.r.longueur * f.u.r.largeur;
           break;
        case 't':
           aire = f.u.t.base * f.u.t.hauteur / 2;
           break;
        }
     return aire;
     }

  int main(void)
     {
     struct figure f1, f2;
     float a1, a2;
     printf("Figure 1\n");
     f1 = saisie_figure();
     a1 = aire_figure(f1);
     printf("Figure 2\n");
     f2 = saisie_figure();
     a2 = aire_figure(f2);
     printf("Position du point d'attache = %.2f\n", a2 / (a1 + a2));
     return 0;
     }
```

Les figures sont représentées par des structures de type `struct figure`. Le champ `type` a pour valeur `'c'`, si la figure est circulaire, `'r'` si elle est rectangulaire et `'t'` si elle triangulaire. Le champ `u` a pour valeur les dimensions de la figure. C'est une union car ces dimensions diffèrent selon la forme de la figure. Cette union a trois champs :

- une structure `c` ayant un champ `rayon`, si la figure est un cercle ;
- une structure `r` ayant un champ `longueur` et un champ `largeur`, si la figure est un rectangle ;
- une structure `t` ayant un champ `base` et un champ `hauteur`, si la figure est un rectangle.

La saisie d'une figure est réalisée par la fonction `saisie_figure` qui demande à l'utilisateur de saisir la forme de la figure et en fonction de celle-ci, soit son rayon, soit sa longueur et sa largeur, soit sa base et sa hauteur. La saisie de la valeur d'une dimension est réalisée par la fonction `saisie_dimension`. Lorsqu'une donnée saisie est incorrecte, sa saisie est redemandée.

L'aire d'une figure `f` est calculée par la fonction `aire_figure`.

Exécutons ce programme dans le cas du composant de la figure 7.2 :

```
Figure 1
Forme (c, r, t) ? r↵
Longueur
valeur ? 0↵
valeur ? 1↵
Largeur
valeur ? 1↵
Figure 2
Forme (c, r, t) ? t↵
Base
valeur ? 1↵
Hauteur
valeur ? 1↵
Position du point d'attache = 0.33
```

Le poids du triangle est la moitié de celui du carré. Le fléau doit donc être suspendu en un point situé au tiers de sa longueur en partant de sa gauche. C'est bien le résultat affiché par le programme.

Exercices

Exercice 7.1. La moyenne $\bar{x}$ et l'écart-type σ_X d'une série de valeurs $X = x_1, .., x_n$ sont calculées par les formules suivantes :

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\sigma_X = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} = \sqrt{\left(\frac{1}{n}\sum_{i=1}^{n} {x_i}^2\right) - \bar{x}^2}$$

Modifier le programme de l'exemple du paragraphe 7.6.1 afin qu'il affiche la moyenne des notes et leur écart-type. On utilisera les fonctions `pow` et `sqrt` de la bibliothèque standard (voir exercice 5.1).

Exercice 7.2. Soit les deux matrices suivantes :

$$m_1 = \begin{bmatrix} 0 & -1 & 1 \\ 2 & 0 & -1 \\ 1 & -1 & 0 \end{bmatrix} \text{ et } m_2 = \begin{bmatrix} 1 & 2 & 0 \\ -1 & 1 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

Ecrire un programme qui :

- définit deux tableaux `m1` et `m2` contenant les matrices m_1 et m_2 et un tableau `m` qui contiendra leur somme puis leur produit ;
- calcule dans `m` la somme de m_1 et m_2 puis l'affiche ligne par ligne ;

– calcule dans `m` le produit de m_1 et m_2 puis l'affiche ligne par ligne.

Exercice 7.3. L'aire d'un triangle peut être calculée à partir des longueurs de ses 3 côtés en utilisant la formule de Héron[3]. Soit *T* un triangle dont les côtés ont pour longueur *a*, *b* et *c*. On a :

$$aire(T) = \sqrt{p(p-a)(p-b)(p-c)} \text{ où } p = \frac{a+b+c}{2}$$

On suppose qu'un point d'un plan est représenté par une structure de type :

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

où `x` et `y` sont les coordonnées de ce point dans un repère orthonormé. Ce type structure sera déclaré globalement.

1. Définir une fonction d'en-tête `float distance(struct point p1, struct point p2)` qui retourne la distance entre les points `p1` et `p2`.
2. Définir une fonction d'en-tête `float aire_triangle(struct point s1, struct point s2, struct point s3)` qui retourne l'aire d'un triangle de sommets `s1`, `s2` et `s3`, calculée par la formule de Héron.
3. Tester cette fonction en l'appelant dans la fonction `main` pour calculer la surface d'un triangle dont les coordonnées des sommets sont saisies au clavier.

On utilisera les fonctions `pow` et `sqrt` de la bibliothèque standard (voir chapitre 5, exercice 5.1).

Exercice 7.4. On considère le tableau suivant qui donne un certain nombre d'informations sur les 8 planètes du soleil (Pluton n'apparaît pas dans ce tableau car elle été rétrogradée au rang de planète naine par l'Union Astronomique Internationale en août 2006 !).

Nom	Distance moyenne au soleil (km)	Diamètre équatorial (km)	Masse
Mercure	57 910 000	4 870	$3{,}30\times10^{23}$
Vénus	108 210 000	12 100	$4{,}87\times10^{24}$
Terre	149 600 000	12 756	$5{,}97\times10^{24}$
Mars	227 940 000	6 790	$6{,}42\times10^{23}$
Jupiter	778 340 000	142 800	$1{,}90\times10^{27}$
Saturne	1 427 010 000	119 300	$5{,}68\times10^{25}$
Uranus	2 869 600 000	47 100	$8{,}68\times10^{25}$
Neptune	4 496 700 000	48 400	$1{,}02\times10^{26}$

Il s'agit d'écrire un programme qui recherche la planète qui passe le plus prés du soleil, celle qui passe le plus loin, la plus grande et la plus massive. Dans ce programme :

– une planète sera représentée par une structure de type `struct planete` à 4 champs : `nom` de type tableau de 8 caractères, `dsoleil` de type `float`, `diametre` de type `int` et masse de type `float` ;

– le tableau des planètes sera représenté par un tableau de 8 structures `planete`.

[3] Héron d'Alexandrie était un mathématicien grec du Ier siècle après J.-C.

Ecrire un programme C qui :

- déclare le type `struct planete` ;
- définit le tableau `planetes` en initialisant les champs de chaque structure avec les valeurs contenues dans chaque ligne (par exemple, l'élément 0 du tableau `planetes` sera initialisé à : `{"Mercure", 5.791e7, 4870, 3.3e23}`) ;
- définit une variable `plus` de type structure à 4 champs de type `int` : `pres`, `loin`, `grande` et `lourde` ;
- affecte au champ `pres` le rang dans le tableau `planetes` de la planète qui passe le plus prés du soleil, au champ `loin` le rang de celle qui passe le plus loin du soleil, au champ `grande` le rang de la plus grande (celle qui a le plus grand diamètre) et au champ `massive` le rang de la plus massive (celle qui a la plus grande masse) ;
- affiche les réponses sous la forme :

  ```
  ? passe le plus près du soleil
  ? passe le plus loin du soleil
  ? est la plus grande
  ? est la plus massive
  ```

  (où ? est un nom de planète qui sera affiché avec la spécification de conversion `%s`).

Exercice 7.5. Soit un tableau `tab` de *n* entiers. On veut trier les nombres de ce tableau par ordre croissant. Pour cela on utilise la méthode suivante dite de « tri par sélection ». On cherche dans `tab`, à partir du rang 0, le nombre le plus petit. Soit *r* son rang. On échange le contenu des cases 0 et *r*. Le nombre le plus petit est maintenant à sa place. On réitère le processus, en cherchant le plus petit nombre à partir du rang 1, puis du rang 2 et ainsi de suite jusqu'au rang $n - 2$ (l'élément de rang $n - 1$ sera alors à sa place).

Considérons, par exemple, le tableau de 3 nombres : [8, 2, 5]. Le plus petit nombre à partir du rang 0 est 2 qui a le rang 1. On échange donc 8 et 2 et le tableau devient : [2, 8, 5]. Le plus petit nombre à partir du rang 1 est 5 qui a le rang 2. On échange donc 8 et 5 et le tableau devient [2, 5, 8] : il est trié.

Ecrire un programme :

- qui définit un tableau `tab` de `N` entiers de type `int` initialisé par `N` valeurs (au moins 10 valeurs dont des ex-æquo) où `N` est une constante symbolique définie par une macro ;
- qui trie ce tableau selon la méthode expliquée ci-dessus et qui l'affiche.

Aide. Ecrire la boucle qui recherche à partir du rang *i* dans le tableau `tab`, le rang *j* du plus petit nombre (ou du premier d'entre eux, s'il y en a plusieurs) puis échange le nombre contenu dans la case *i* et celui contenu dans la case *j*. Insérer cette boucle dans une boucle externe qui parcoure les nombres du tableau `tab` est les range à leur place.

8
Adresses et pointeurs

Dans les programmes que nous avons étudiés jusqu'à présent, l'accès aux variables, aux tableaux et aux fonctions s'est fait par leur nom. Mais il y a des cas où il est nécessaire de le faire par leur adresse pour, entre autres, accéder aux éléments d'un tableau, définir des fonctions qui accèdent directement à la valeur d'une variable définie dans le corps de la fonction appelante, passer un tableau ou une fonction en argument d'une fonction, accéder à des variables anonymes définies dynamiquement pendant l'exécution du programme.

Rappelons :

- qu'une variable est une case de la mémoire dans laquelle est rangée une valeur. L'adresse d'une variable est l'adresse de cette case.
- qu'un tableau est constitué d'éléments qui sont des tableaux ou des variables et qui sont rangés de façon contigüe en mémoire. L'adresse d'un tableau est celle de l'élément de rang 0 de ce tableau.
- que le code exécutable d'un programme est lui-même rangé dans la mémoire. A chaque fonction de ce programme, il correspond un point d'entrée dans ce code qui est l'adresse de la première instruction machine à exécuter quand cette fonction est appelée. Cette adresse est l'adresse de la fonction.

En C, les adresses des variables, des tableaux et des fonctions sont des valeurs au même titre que les nombres, les structures ou les unions. Elles peuvent être la valeur d'un argument ou la valeur de retour d'un opérateur ou d'une fonction. Il est possible d'obtenir l'adresse d'une variable, d'un tableau ou d'une fonction à partir de son nom, d'accéder à la valeur d'une variable ou à un élément de tableau dont l'adresse est donnée, d'appeler une fonction en indiquant son adresse.

Attention ! Si les adresses des fonctions sont des valeurs, les fonctions elles ne le sont pas. Il n'est pas possible d'écrire un programme qui crée une nouvelle fonction ou modifie le code d'une fonction. □

Il est d'usage d'appeler « pointeur » une expression ou une variable dont la valeur est l'adresse d'une variable, d'un tableau ou d'une fonction. On dit d'une telle expression qu'elle « pointe » sur cette variable, ce tableau ou cette fonction qui, eux, sont « pointés » par cette expression.

Les adresses manipulées par un programme C ont un type. De même que l'expression d'un type tableau (voir chapitre 7, paragraphe 7.1.3), l'expression d'un type adresse est loin d'être naturelle en C. Pour en faciliter la compréhension, nous la paraphraserons par une description naturelle ayant l'une des trois formes suivantes :

- « adresse d'une variable de type T » où T est un type de base ou la description naturelle d'un type adresse ;
- « adresse d'un T » où T est la description naturelle d'un type tableau ;
- « adresse d'une fonction de type $T_1 \times ... \times T_n \rightarrow T$ » où T_1, .., T_n et T sont soit des types de base, soit des descriptions naturelles de type.

Rappelons :

- qu'un type de base est soit un type numérique, soit un type structure, soit un type union, soit un nom de type introduit par une déclaration `typedef` ;
- que la description naturelle d'un type tableau a la forme suivante : « tableau de *n* éléments de type *T* » où *T* est soit un type de base, soit la description naturelle d'un type tableau, soit la description naturelle d'un type adresse.

Par exemple :

- « adresse d'une variable de type `int` » ;
- « adresse d'un tableau de 3 éléments de type adresse d'une variable de type `int` » ;
- « adresse d'une fonction de type `float` → `int` ».

sont des descriptions naturelles d'un type adresse.

Dans la suite de ce chapitre, nous étudierons : la déclaration de variables de type adresse et les opérateurs de manipulation d'adresses (paragraphe 8.1) ; le passage d'arguments par adresse (paragraphe 8.2) ; les rôles spécifiques joués par l'adresse nulle et les adresses de type générique (paragraphe 8.3) ; l'allocation dynamique de variables ou de tableaux (paragraphe 8.4). Nous présenterons ensuite deux exemples de programme mettant en œuvre ces notions (paragraphe 8.5).

8.1 Adresses de variables, de tableaux et de fonctions

8.1.1 Adresses de variables

La déclaration d'une variable *var* de type « adresse d'une variable de type *T* », où *T* est un type de base, a la forme suivante :

*T *var;*

Par exemple :

```
int *pi;
char *pc;
struct personne *pp1;
Personne *pp2;
```

déclarent successivement un pointeur `pi` sur une variable de type `int`, un pointeur `pc` sur une variable de type `char`, un pointeur `pp1` sur une variable de type `struct personne` et un pointeur `pp2` sur une variable de type `Personne`.

On peut déclarer simultanément des variables de type *T* et des variables de type « adresse d'une variable de type *T* ». Par exemple :

```
int i, *pi;
```

est la déclaration d'une variable `i` de type `int` et d'un pointeur `pi` sur une variable de type `int`.

Un élément d'un tableau ou un champ d'une structure peut être de type adresse. Par exemple :

```
struct personne *personnes[50];
```

déclare un tableau `personnes` dont les éléments pointent sur une variable de type `struct personne` » et :

```
struct livre
   {
   char titre[30];
   struct personne *auteur;
   }
```

déclare un type `struct livre` dont le champ `auteur` pointe sur une variable de type `struct personne`.

Un type structure *T* est récursif s'il possède un champ qui pointe directement ou indirectement sur une variable de type *T*. Par exemple :

```
struct personne
   {
   char nom[30];
   struct personne *mere;
   struct personne *pere;
   }
```

est la déclaration d'un type structure récursif. Une telle déclaration est possible car un nom de structure peut être utilisé avant ou dans sa propre déclaration. Cette déclaration peut être simplifiée en nommant le type `struct personne` :

```
typedef struct personne Personne;

struct personne
   {
   char nom[30];
   Personne *mere;
   Personne *pere;
   }
```

Si le père ou la mère d'une personne est inconnu le champ `pere` ou le champ `mere` auront pour valeur l'adresse nulle qui est une instance de tout type adresse (voir paragraphe 8.3).

L'adresse d'une variable est obtenue par l'opérateur `&`. Si *exp* est une valeur gauche de type *T* alors :

`&`*exp*

est une expression qui a les propriétés suivantes :

- *type*(`&`*exp*) = adresse d'une variable de type *T*
- *val*(`&`*exp*) = adresse de la variable désignée par *exp*.

Attention ! `&`*exp* n'est pas une valeur gauche. Par exemple, si `i` est une variable de type `int`, on ne pourra pas écrire `&i = 12` en pensant affecter la valeur 12 à `i`. Il suffit, évidemment, d'écrire `i = 12`. □

L'opérateur `*` permet d'obtenir la valeur d'une variable à partir de son adresse. On dit que l'on « déréférence » cette adresse. Si *exp* est une expression de type « adresse d'une variable de type *T* » alors :

`*`*exp*

est une expression qui a les propriétés suivantes :

- *type*(`*`*exp*) = *T*
- *val*(`*`*exp*) = valeur de la variable d'adresse *val*(*exp*)
- c'est une valeur gauche.

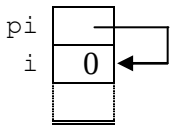
Exemple 8.1. Soit l'instruction :

```
(1) {
(2) int i, *pi;
(3) i = 0;
(4) pi = &i;
(5) *pi = 15;
(6) }
```

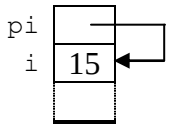
Remarquons que l'expression `pi = &i` est correctement typée : `i` est de type `int` et `pi` est l'adresse d'une variable de type `int` tout comme l'expression `&i`.

L'évolution de la pile au cours de l'exécution de cette instruction est la suivante :

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



– après l'exécution de l'instruction (5) :



□

L'opérateur `->` permet d'obtenir la valeur d'un champ d'une structure ou d'une union dont l'adresse est donnée. L'expression :

exp`->`*c*

est équivalente à :

`(*`*exp*`).`*c*

Exemple 8.2. Soit `p` et `pp` deux variables définies par :

```
struct personne
   {
   char nom[30];
   int age;
   } p = {"Dupont", 36}, *pp;
```

Après de l'exécution de l'instruction :

```
pp = &p;
```

on aura :

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

□

8.1.2 **Adresses de tableaux**

Un tableau est identifié par l'adresse de son élément de rang 0. Une expression de type tableau a donc pour valeur l'adresse de l'élément de rang 0 du tableau qu'elle désigne. C'est le cas, notamment, du nom d'un tableau qui de plus est une constante (voir chapitre 7, paragraphe 7.1.3).

Exemple 8.3. Soit les deux tableaux définis par :

```
int pluie[12];
int m[2][3];
```

Dans une expression :

- `pluie` est une constante de type « adresse d'une variable de type `int` » qui a pour valeur l'adresse de la variable `pluie[0]` ;
- `m` est une constante de type « adresse d'un tableau de 3 éléments de type `int` » qui a pour valeur l'adresse du sous-tableau `m[0]`. □

La déclaration d'une variable *p* de type « adresse d'un tableau de n_1 éléments de type tableau de ... de type tableau de n_k éléments de type *T* » où *T* est un type de base, a la forme suivante :

T `(*`*p*`)[`n_1`]...[`n_k`];`

Par exemple :

```
float (*ptf)[3];
```

déclare une variable `ptf` de type « adresse d'un tableau de 3 éléments de type `float` ».

L'opérateur de déréférencement `*` s'applique aussi aux adresses de tableaux. Si *exp* est une expression de type « adresse d'un tableau d'éléments de type *T* », alors :

`*`*exp*

est une expression qui a les propriétés suivantes :

- *type*(`*`*exp*) = « adresse d'un élément (tableau ou variable) de type *T* »
- *val*(`*`*exp*) = adresse de l'élément de rang 0 du tableau pointé par *exp*
- ce n'est pas une valeur gauche.

Arithmétique des adresses

C a la particularité d'offrir trois opérations arithmétiques pour manipuler les adresses d'éléments de tableaux : l'addition et la soustraction d'un nombre à une adresse et la soustraction de deux adresses.

Soit *tab* un tableau. Si exp_1 est une expression dont la valeur *a* est l'adresse d'un élément du tableau *tab* et si exp_2 est une expression dont la valeur *i* est un nombre entier, alors :

- *val*(exp_1 `+` exp_2) = adresse du i^e élément suivant l'élément d'adresse *a*
- *val*(exp_1 `-` exp_2) = adresse du i^e élément précédant l'élément d'adresse *a*

Si exp_1 et exp_2 sont des expressions dont les valeurs a_1 et a_2 sont les adresses d'un élément du tableau *tab*, alors :

- *val*(exp_1 `-` exp_2) = *i*, si $a_1 + i = a_2$

L'opérateur `[]` d'accès à un élément de tableau (chapitre 7, paragraphe 7.1.4) est défini en fonction de l'addition d'une adresse et d'un nombre. On a l'équivalence :

exp_1`[`exp_2`]` ≡ `*(`exp_1 `+` exp_2`)`

Notons de plus que si *tab* est le nom d'un tableau, on a :

tab + *i* ≡ `&`*tab*`[`*i*`]`

car la valeur du nom d'un tableau est égale à l'adresse de son élément de rang 0.

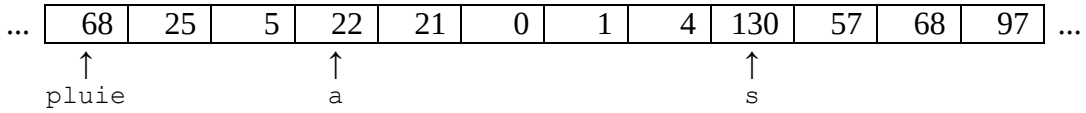
Exemple 8.4. L'instruction :

```
{
int pluie[12] = {68, 25, 5, 22, 21, 0, 1, 4, 130, 57, 68, 97};
int *a, *s;
a = pluie + 3;
s = pluie + 8;
printf("val(*a) = %d, val(*s) = %d, val(s - a) = %d",
       *a, *s, s – a);
}
```

affichera :

```
val(*a) = 22, val(*s)= 130, val(s - a)= 5
```

comme on le comprendra facilement en observant l'implantation du tableau `pluie` en mémoire :



On constatera notamment que l'on a bien *val*(`*(pluie + 3)`) = `pluie[3]` = 22.

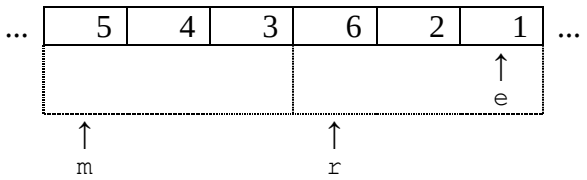
L'instruction :

```
{
int m[2][3] = {{5, 4, 3}, {6, 2, 1}};
int (*r)[3];
int *e;
r = m + 1;
e = (*r) + 2;
printf("val(*e) = %d, val((*r)[2]) = %d, val(m - r) = %d",
       *e, (*r)[2] , m - r);
}
```

où la variable `r` est de type adresse d'un tableau de 3 éléments de type `int` à laquelle est affecté l'adresse du sous-tableau de rang 1 du tableau `m`, affichera :

```
val(*e) = 1, val((*r)[2]) = 1, val(m - r) = -1
```

comme on le comprendra facilement en observant l'implantation du tableau `m` en mémoire :



□

8.1.3 **Adresses de fonctions**

Dans une expression, le nom d'une fonction se comporte comme une constante qui a pour valeur l'adresse du début du code de cette fonction.

La déclaration d'une variable *var* de type « adresse d'une fonction de type $T_1 \times ... \times T_n \rightarrow T$ », a la forme suivante :

T `(*`*var*`)(`T_1`, ..., `T_n`)`

Par exemple :

```
float (*op)(float, float);
```

déclare une variable `op` de type « adresse d'une fonction de type `float` × `float` ⟶ `float` ».

En C, l'appel de fonction est l'opérateur `()`.

Si *exp* est une expression de type adresse d'une fonction de type $T_1 \times ... \times T_n \rightarrow T$ et exp_1, ..., exp_n sont des expressions de types respectifs T_1, ..., T_n alors :

$exp(exp_1, ..., exp_n)$

est une expression telle que :

- $type(exp(exp_1, ..., exp_n)) = T$
- $val(exp(exp_1, ..., exp_n))$ = valeur retournée par l'appel de la fonction d'adresse $val(exp)$ avec les arguments effectifs $val(exp_1)$, ..., $val(exp_n)$ (voir chapitre 5, paragraphe 5.1.3)

Notons que l'écriture $(*exp)(exp_1, ..., exp_n)$ est aussi admise pour marquer le fait qu'une fois l'adresse de la fonction obtenue, il faut la déréférencer pour obtenir son code.

8.1.4 **Déclarations complexes**

La déclaration en C de variables de type adresse ou de tableaux dont les éléments terminaux sont de type adresse est complexe comme l'ont montré certains exemples ci-dessus. Comment, par exemple, déclarer :

- une variable `ppi` de type adresse d'une variable de type adresse d'une variable de type `int` ;
- une variable `pti` de type adresse d'un tableau de 3 éléments de type `int` ;
- un tableau `tfi` de 4 éléments de type adresse d'une fonction de type `int` × `int` ⟶ `int` ?

Voici une méthode pour le faire. Soit *T* le type de la variable ou du tableau à déclarer :

1. Ecrire la description naturelle du type de la variable ou du tableau à déclarer. Cette description devra se terminer par un type de base T_{base} (dans les exemples ci-dessus T_{base} = `int`).
2. Ecrire, en s'appuyant sur cette description, l'expression *E* qui permet d'obtenir une valeur du type de base à partir d'une variable de type *T*.
3. Dans cette expression, remplacer les indices de tableau par le nombre d'éléments du tableau correspondant et les arguments des fonctions par l'expression de leur type.
4. Assembler la déclaration sous la forme :

   T_{base} *E;*

Appliquons cette méthode pour obtenir les déclarations des variables `ppi`, `pti` et celle du tableau `tfi` décrits ci-dessus :

- **Déclaration de la variable `ppi`.** Pour obtenir l'entier valeur de la variable pointée par la variable pointée par la variable `ppi`, il faut tout d'abord déréférencer la valeur de `ppi` : `*ppi`. On obtient l'adresse de la variable pointée par `ppi`. Il faut ensuite déréférencer cette adresse : `**ppi`. On obtient l'entier recherché. La déclaration de la variable `ppi` est donc :

  ```
  int **ppi;
  ```

  Notons que les parenthèses sont inutiles car l'opérateur `*` est associatif de droite à gauche.
- **Déclaration de la variable `pti`.** Pour obtenir le i^e entier du tableau pointé par `pti`, il faut tout d'abord déréférencer la valeur de `pti` : `*pti`. On obtient le tableau pointé par `pti`. Il

faut ensuite obtenir la valeur du i^e élément de ce tableau : `(*pti)[`i`]`. On obtient l'entier recherché. Après avoir remplacé l'indice *i* par le nombre d'éléments du tableau pointé par `pti` (soit 3), on obtient l'expression : `(*pti)[3]`. La déclaration de la variable `pti` est donc :

```
int (*pti)[3];
```

Notons que les parenthèses autour de `*pti` sont indispensables car l'opérateur `[]` a une priorité supérieure à celle de l'opérateur `*`.

– **Déclaration du tableau `tfi`.** Pour obtenir l'entier retourné par la i^e fonction du tableau `tfi` appliquée aux entiers a_1 et a_2, il faut tout d'abord obtenir le i^e élément de rang *i* du tableau `tfi` : `tfi[`i`]`. On obtient ainsi l'adresse de la i^e fonction. Il faut ensuite déréférencer cette adresse pour obtenir son code : `*tfi[`i`]`. Il faut enfin appliquer ce code aux entiers a_1 et a_2 : `(*tfi[`i`])(`a_1`, `a_2`)`. On obtient l'entier recherché. Après avoir remplacé l'indice *i* par le nombre d'éléments du tableau `tfi` (soit 4) et les arguments de la fonction par leur type (soit `int`) on obtient l'expression : `(*tfi[4])(int, int)`. La déclaration du tableau `tfi` est donc :

```
int (*tfi[4])(int, int)
```

Notons que les parenthèses autour de `*tfi[4]` sont indispensables car l'opérateur `()` d'application d'une fonction `()` a une priorité supérieure à celle de l'opérateur `*`.

Rappelons de plus que l'on peut regrouper dans la même déclaration des variables ou des tableaux ayant le même type de base. Par exemple, on pourrait regrouper les trois déclarations précédentes dans la déclaration unique :

```
int **ppi, (*pti)[3], (*tfi[4])(int, int);
```

Pour améliorer la lisibilité de ces déclarations, il ne faut pas hésiter à les décomposer en nommant certaines expressions de types. Par exemple, la déclaration du tableau `tfi` aurait pu être décomposée en déclarant un type `Fonction` synonyme du type adresse d'une fonction de type `int` × `int` → `int`, puis en déclarant le tableau `tfi` comme un tableau de 4 variables de type `Fonction` :

```
typedef int (*Fonction)(int, int);
Fonction tfi[4];
```

8.2 Passage d'arguments par adresse

En C, les arguments d'une fonction sont « passés par valeur » : les arguments effectifs sont affectés aux l'argument formels correspondants (voir chapitre 5, paragraphe 5.1.3). Dans le cas où un argument effectif est la valeur d'une variable de la fonction appelante, la valeur de cette variable est recopiée dans l'espace mémoire de la fonction appelée.

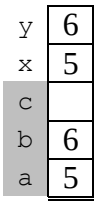
Exemple 8.5.

Soit le programme suivant :

```
int somme(int x, int y)
    {
    return (x + y);
    }
```

```
int main(void)
   {
   int a, b, c;
   a = 5;
   b = 6;
   c = somme(a, b);
   return 0;
   }
```

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



On constate effectivement 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 a deux inconvénients :

- une fonction ne peut pas modifier la valeur d'une variable de la fonction appelante car elle n'a accès qu'à une copie de la valeur de cette variable ;
- si l'argument effectif est une structure volumineuse, sa copie peut être coûteuse en temps et en mémoire.

La solution, dans ces deux cas, est de passer en argument l'adresse de cette variable au lieu de sa valeur. On dit alors que cet argument est « passé par adresse ».

8.2.1 **Passage en argument de l'adresse d'une variable**

L'exemple 8.6 illustre le passage en argument de l'adresse d'une variable et son intérêt lorsqu'une fonction doit retourner plusieurs valeurs.

Exemple 8.6. Le programme suivant appelle une fonction `somme_et_produit` pour calculer la somme et le produit de deux nombres.

```
(1)  void somme_et_produit(int x, int y, int *s, int *p)
(2)     {
(3)     *s = x + y;
(4)     *p = x * y;
(5)     }

(6)  int main(void)
(7)     {
(8)     int somme, produit;
(9)     somme_et_produit(2, 3, &somme, &produit);
(10)    return 0;
(11)    }
```

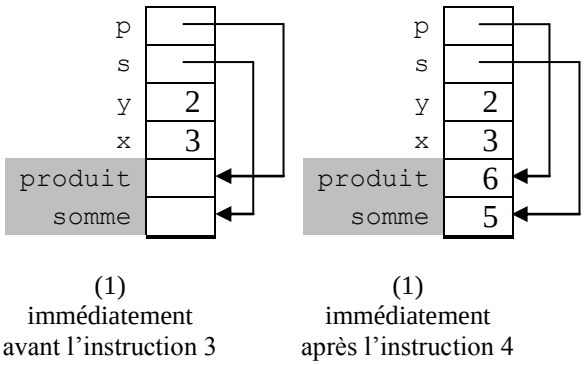


Figure 8.1. *Exécution du corps de la fonction* **`somme_et_produit`**

La figure 8.1 montre l'évolution de la pile durant l'exécution du corps de la fonction `somme_et_produit`. □

L'exemple 8.7 montre comment définir une fonction qui modifie les champs d'une structure dont l'adresse est passée en argument.

Exemple 8.7. On suppose que les employés d'une entreprise sont décrits par une structure de type :

```
struct employe
   {
   char nom[30];
   float salaire;
   }
```

La fonction :

```
void augmenter_salaire(struct employe *pe, float taux)
   {
   pe->salaire *= (1 + taux);
   }
```

augmente de `taux`% le salaire d'un employé `pe` où `pe` est un pointeur sur la structure qui décrit cet employé. □

8.2.2 **Passage en argument de l'adresse d'un tableau**

En C, passer un tableau en argument c'est passer l'adresse de ce tableau en argument. Le tableau lui-même ne peut pas l'être puisqu'un tableau n'étant pas une valeur, il ne peut pas être affecté à un argument formel qui est une variable. De toute façon ce serait une solution coûteuse, puisqu'elle entraînerait une copie du tableau à l'appel de la fonction et une au retour dans la fonction appelante, dans le cas où la fonction appelée a modifié le tableau.

Il y a deux façons de déclarer un argument de type adresse d'un tableau d'éléments de type *T* : soit comme un nom de tableau d'éléments de type *T*, soit comme une variable de type adresse d'un élément de type *T*. Le nombre d'éléments du tableau passé en argument peut être omis mais pas les nombres d'éléments de ses sous-tableaux qui sont nécessaires pour calculer leurs adresses.

Par exemple :

– si le tableau dont l'adresse doit être passée en argument est de type tableau de 12 éléments de type `int` et que le nom de cet argument est `t`, il pourra être déclaré : `int t[12]`, `int t[]` ou `int *t` ;
– si le tableau dont l'adresse doit être passée en argument est de type tableau de 2 tableaux de 3 éléments de type `int` et que le nom de cet argument est `t`, il pourra être déclaré : `int t[2][3]`, `int t[][3]` ou `int (*t)[3]`.

Attention ! Ce n'est pas parce que l'on indique le nombre d'éléments du tableau passé en argument et celui de chacun de ses sous-tableaux, qu'un contrôle de débordement aura lieu. Il faudra donc, si nécessaire, passer également ces nombres d'éléments en arguments et effectuer cette vérification dans le corps de la fonction appelée. □

Exemple 8.8. La fonction suivante calcule et retourne la somme des éléments d'un tableau dont l'adresse `t` et le nombre d'éléments `n` lui sont passés en arguments :

```
int somme_tableau(int t[], int n)
   {
   int i, s = 0;
   for (i = 0; i < n; i++)
      s += t[i];
   return s;
   }
```

L'argument `t` aurait pu être déclaré `int t[12]` ou `int *t`. La déclaration `int t[]` a l'avantage de mettre en évidence que cette fonction peut être utilisée pour un tableau de nombre d'éléments quelconque à condition bien sûr que ce nombre d'éléments soit passé en argument.

La fonction suivante calcule et retourne le déterminant d'une matrice de 2×2 entiers dont l'adresse `m` est passée en argument :

```
int determinant_matrice_2_2(int m[2][2])
   {
   return m[0][0] * m[1][1] - m[0][1] * m[1][0];
   }
```

L'argument `m` aurait pu être déclaré `int m[][2]` ou `int (*m)[2]` mais ni `m[][]` ni `(*m)[]`. □

8.2.3 **Passage en argument de l'adresse d'une fonction**

En C, passer une fonction en argument c'est passer l'adresse de cette fonction en argument. Le code de cette fonction ne peut pas l'être puisque ce n'est pas une valeur.

Exemple 8.9. La fonction `operation` définie ci-dessous retourne la valeur de l'appel de la fonction d'adresse `op` avec les arguments effectifs `g` et `d` :

```
float operation(float (*op)(float, float), float g, float d)
   {
   return op(g, d);
   }
```

Notons que les types des arguments sont cohérents. La fonction passée en argument s'applique à deux flottants et les variables `g` et `d` sont elles-mêmes de type flottant.

Si l'on suppose que `max` est une fonction définie par :

```
float max(float x, float y)
   {
   if (x >= y)
```

```
        return x;
    else
        return y;
    }
```

alors l'appel `operation(max, 2.6, 7.3)` aura la valeur 7,3. □

8.2.4 **Passage en argument de l'adresse d'une variable non modifiable**

Nous avons vu que le passage de l'adresse d'une variable à une fonction permettait à celle-ci de modifier la valeur de cette variable. Ce n'est pas toujours souhaitable. Par exemple, on peut vouloir définir une fonction réalisant des calculs sur la valeur d'une variable dont l'adresse lui est passée en argument tout en interdisant de changer la valeur de cette variable. Cette interdiction peut être mise en place en déclarant constante la variable pointée par l'argument. De même, pour interdire la modification des éléments d'un tableau passé en argument, on déclarera constants les éléments du tableau pointé par cet argument.

Par exemple, si *T* est un type de base, la déclaration :

```
const T *var;
```

déclare un pointeur *var* sur une variable de type *T* à valeur non modifiable et la déclaration :

```
const T tab[];
```

déclare un tableau *tab* d'éléments de type *T* à valeur non modifiable.

Supposons qu'un point soit décrit par une structure de type :

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

Dans le corps de la fonction d'en-tête :

```
float calcul(const struct point *p1, const struct point *p2)
```

les valeurs des coordonnées des points `p1` et `p2` pourront être utilisées pour réaliser le calcul mais elles ne pourront pas être modifiées. L'expression `p1->x`, par exemple, ne pourra pas apparaitre à gauche de l'opérateur `=`.

Dans le corps de la fonction d'en-tête :

```
int calcul(const int t[]);
```

les valeurs des éléments du tableau `t` pourront être utilisées pour réaliser le calcul, mais elles ne pourront pas être modifiées. L'expression `t[i]`, par exemple, ne pourra pas apparaitre à gauche de l'opérateur d'affectation.

8.3 Adresse nulle et adresses de type générique

Il est parfois nécessaire de disposer d'un pointeur qui pointe sur « rien ». Comme, dans l'implantation d'un programme C, aucune variable n'est rangée à l'adresse 0, cette adresse, que nous appellerons l'adresse nulle, a été choisie comme « l'adresse de rien ». Elle est la valeur de la constante `NULL` définie dans le fichier d'en-têtes `stddef.h`.

L'adresse nulle est une instance de tout type d'adresse.

C offre aussi un type d'adresse générique : le type `void *` que l'on peut qualifier de type « adresse quelconque ». Toute adresse peut être affectée à une variable de type `void *` et inversement. Par contre, pour accéder à une entité de type *T* à partir de son adresse de type générique, il faut tout d'abord convertir cette adresse en une adresse d'une entité de type *T*.

L'intérêt des adresses génériques est de permettre la définition de fonctions polymorphes : des fonctions qui peuvent s'appliquer à des adresses de tout type. De bons exemples, sont les fonctions `qsort` et `bsearch` de la bibliothèque standard qui permettent de trier un tableau ou de rechercher un élément dans un tableau quelque soit le type des éléments de ce tableau. Ces deux fonctions sont déclarées dans le fichier d'en-têtes `stdlib.h`.

8.4 Allocation dynamique de variables ou de tableaux

Dans les programmes que nous avons étudiés jusqu'à présent, les variables manipulées sont définies dans le texte du programme. Ceci est suffisant lorsque les variables sont connues lors de l'écriture du programme, mais ce n'est pas toujours le cas. Considérons, par exemple, un programme qui gère des objets dont le nombre varie au cours de l'exécution du programme. Une première solution est de fixer dans le texte du programme un nombre maximum *N* d'objets et de réserver en mémoire une place pour ces *N* objets. Mais, si *N* a été a été sous-évalué, il faudra le modifier et recompiler le programme pour pouvoir gérer un plus grand nombre d'objets et si *N* a été surévalué, de la place mémoire aura été réservée pour rien. Une meilleure solution serait, lors de la création d'un nouvel objet, de pouvoir allouer la place mémoire nécessaire à sa description et lors de la suppression d'un objet, de pouvoir libérer la place mémoire occupée par sa description. Ceci est possible en C, grâce à la fonction `malloc` qui permet d'allouer dynamiquement une variable en mémoire et à la fonction `free` qui permet de libérer la place occupée par cette variable lorsqu'elle n'est plus utilisée.

C'est l'utilisation de ces deux fonctions que nous étudierons au paragraphe 8.4.3, mais auparavant, il nous faut apprendre à exprimer des types indépendamment d'une déclaration de variable ou de tableau et à calculer la place occupée par une valeur ou un tableau d'un certain type. C'est l'objet des paragraphes 8.4.1 et 8.4.2.

8.4.1 Types anonymes

Il est parfois nécessaire d'exprimer un type sans l'associer à un nom de variable ou de tableau. C'est le cas, dans les opérations de changement de type (voir chapitre 3, paragraphe 3.2.6), de calcul de la taille mémoire d'une instance d'un certain type (voir ci-dessous paragraphe 8.4.2) et dans la déclaration d'un pointeur sur une fonction (voir ci-dessous paragraphe 8.4.3).

Pour obtenir l'expression d'un type, il suffit d'écrire la déclaration d'une variable de ce type, puis de supprimer de cette déclaration le nom de cette variable. Par exemple :

- `int` est l'expression du type de base `int` (`int` ~~`i`~~ → `int`) ;
- `int *` est l'expression du type dont les instances sont des adresses de variable de type `int` (`int *`~~`p`~~ → `int *`) ;
- `Personne [10]` est l'expression du type dont les instances sont des adresses de tableaux de type « tableau de 10 éléments de type `Personne` » (`Personne` ~~`t`~~`[10]` → `Personne [10]`) ;
- `struct personne *[10]` est l'expression du type dont les instances sont des adresses de tableaux de 10 éléments de type « adresse d'une variable de type `struct personne` » (`struct personne *`~~`t`~~`[10]` → `struct personne *[10]`).

8.4.2 **Taille des instances d'un type**

La taille mémoire des instances d'un type peut être obtenue en utilisant l'opérateur `sizeof` qui peut être utilisé de deux façons différentes :

- l'appel `sizeof(`*T*`)` retourne la taille en nombre d'octets d'une valeur de type *T* ;
- l'appel `sizeof` *exp* a pour valeur la taille en nombre d'octets d'une valeur de type *type*(*exp*).

Ces expressions ont pour particularité d'être évaluées lors de la compilation du programme et non lors de son exécution car les informations sur les types des données ne sont pas conservées dans le code exécutable d'un programme.

Par exemple :

- l'appel `sizeof(float)` retourne la taille d'une valeur de type `float` ;
- l'appel `sizeof p` retourne la taille d'une valeur de type `Personne` si `p` est une variable de type `Personne`.

Attention ! si *tab* est un nom de tableau, l'expression :

`sizeof` *tab*

a pour valeur la taille de ce tableau (c.-à-d. le nombre d'octets occupés par la suite de ses éléments) et l'expression :

`sizeof` *tab* `/ sizeof` *tab*`[0]`

a pour valeur le nombre d'éléments de ce tableau. Par exemple, si `tab` est un tableau défini par :

```
int tab[10];
```

et si la taille d'une valeur de type `int` est 4 octets, on a :

val(`sizeof tab`) = 40
val(`sizeof tab / sizeof tab[0]`) = 10

□

Le type d'une expression `sizeof` est le type `size_t` qui est déclaré dans le fichier d'en-têtes `stddef.h`. Toute variable ayant pour valeur une taille devra donc être déclarée de type `size_t`. De même, toute fonction retournant une taille devra avoir `size_t` comme type de retour.

8.4.3 **Allocation dynamique d'une variable**

L'allocation dynamique d'une variable est réalisée en utilisant la fonction `malloc` de la bibliothèque standard, déclarée dans le fichier `stdlib.h`. Elle a pour en-tête :

```
void *malloc (size_t t)
```

Elle alloue dans le tas un emplacement de `t` octets et retourne l'adresse de cet emplacement. Rappelons (voir chapitre 2, paragraphe 2.6) que le tas est l'une des trois parties de la mémoire d'un programme : celle qui contient des variables allouées dynamiquement.

L'allocation dynamique d'une variable de type *T* sera alors réalisée en évaluant l'expression :

`(`*T* `*) malloc(`*taille d'une valeur ou d'un tableau de type T*`)`

qui aura pour valeur l'adresse de cette variable. Notons que `malloc` retournant une adresse de type générique (`void *`), il est nécessaire de la convertir en une adresse d'une variable de type *T* en lui appliquant l'opérateur de changement de type `(T *)`.

Si l'allocation a échoué par manque de mémoire, l'adresse nulle (valeur de la constante `NULL`) est retournée. Il faut tester cette éventualité à la suite de chaque appel de `malloc`.

Exemple 8.10. On suppose qu'un point du plan est représenté par une instance du type `Point` dont la déclaration est la suivante :

```
typedef struct
   {
   char nom;
   float x, y;
   } Point;
```

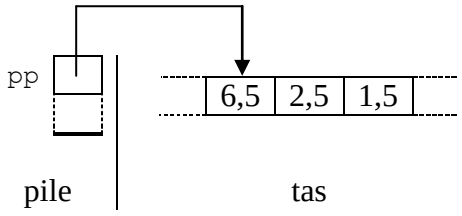
La fonction suivante :

```
Point *creer_point(char nom, float x, float y)
   {
   Point *pp;
   pp = (Point *) malloc(sizeof(Point));
   if (pp == NULL)
      {
      printf("Allocation impossible !");
      exit(1);
      }
   pp->nom = nom;
   pp->x = x;
   pp->y = y;
   return pp;
   }
```

crée un point de nom `nom` et de coordonnées `x` et `y`. Plus précisément, elle alloue dynamiquement une variable de type `Point` et affecte son adresse à la variable `pp` de type « adresse d'une variable de type `Point` ». Elle affecte ensuite aux champs `nom`, `x` et `y` de la structure pointée par `pp` les valeurs des variables `nom`, `x` et `y`. Enfin, elle retourne l'adresse `pp`.

La figure suivante montre l'état de la mémoire après l'exécution de l'instruction :

```
pp = creer_point('A', 2.5, 6.5);
```



□

8.4.4 **Allocation dynamique d'un tableau**

Rappelons qu'un tableau, en C, est une suite contiguë d'éléments qui peuvent être des variables ou des tableaux. Pour allouer dynamiquement un tableau de *n* éléments de type *T*, il faut allouer dans le tas un emplacement de (*n* × taille de *T*) octets dans lequel seront stockés les *n* éléments de ce tableau et récupérer l'adresse du début de cette zone convertie en une

adresse d'une variable de type *T* : celle de l'élément de rang 0 de ce tableau. Ceci peut être fait en utilisant la fonction `malloc` et en évaluant l'expression :

`(`*T* `*) malloc(`*n* `*` *taille de T*`)`

Exemple 8.11. La fonction suivante alloue dynamiquement un tableau de `n` entiers, remplit ce tableau de telle façon que chaque élément a pour valeur son rang et retourne l'adresse de l'élément de rang 0 de ce tableau

```
int *allouer_tableau(int n)
   {
   int *tab, i;
   tab = (int *) malloc(n * sizeof(int));
   if (tab == NULL)
      {
      printf("Allocation impossible !");
      exit(1);
      }
   for (i = 0; i < n; i++)
      tab[i] = i;
   return tab;
   }
```

□

On peut aussi utiliser la fonction `calloc` de la bibliothèque standard Cette fonction est déclarée dans le fichier `stdlib.h`. Son en-tête est :

```
void *calloc (size_t n, size_t t)
```

Elle alloue dans le tas `n` variables consécutives de taille `t`, les initialise à 0 (ce que ne fait pas la fonction `malloc`) et retourne l'adresse de la première de ces variables. L'allocation dynamique d'un tableau de *n* éléments de type *T* sera alors réalisée en évaluant l'expression :

`(`*T* `*) calloc(`*n*`,` *taille de T*`)`

qui aura pour valeur l'adresse de l'élément de rang 0 de ce tableau.

En cas d'échec de l'allocation, du à une place mémoire insuffisante, la fonction `calloc` retourne l'adresse nulle.

8.4.5 **Libération d'un emplacement mémoire alloué dynamiquement**

Lorsqu'une variable ou un tableau alloué dynamiquement par l'appel de la fonction `malloc` ou `calloc`, n'est plus utilisé par le programme en cours, on pourra demander sa libération en faisant appel à la fonction `free` de la bibliothèque standard. Cette fonction est déclarée dans le fichier `stdlib.h`. Son en-tête est :

```
void free(void *a)
```

Elle signale que l'emplacement d'adresse `a`, qui devra avoir été alloué par un appel à `malloc` ou `calloc`, peut être récupéré.

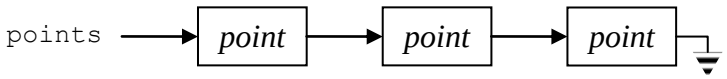
Attention ! Avant de faire appel à `free`, il faut être sûr que la variable ou le tableau stocké dans l'emplacement à libérer ne sera plus utilisé dans la suite de l'exécution du programme. □

8.5 Exemples

8.5.1 Gestion d'un ensemble de points

L'objectif de ce programme est de gérer un ensemble de points du plan qui évolue par création ou suppression de points. Ce programme pourrait, par exemple, servir de base à la gestion de la mémoire d'un programme de dessin de figures géométriques planes. On suppose qu'un point est décrit par son nom, son abscisse et son ordonnée et qu'il est identifié par son nom c.-à-d. qu'il n'y a pas deux points de même nom dans l'ensemble des points.

Nous proposons de représenter cet ensemble de points sous la forme d'une liste chaînée, comme le montre la figure suivante :



Cette liste chaînée peut être représentée en C à l'aide de pointeurs de la façon suivante :

– on décrit un point par une structure instance du type `Point` déclaré par :

```
typedef struct point Point;

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

et dont le champ `suivant` pointe sur le point suivant dans la liste chaînée ou sur rien si ce point est le dernier de la liste chaînée ;

– on définit une variable `points` de type `Point *` qui pointe sur le premier point de la liste chaînée ou sur rien si l'ensemble des points est vide.

La création d'un point et son insertion dans l'ensemble de points est réalisée par la fonction `creer_point` dont la définition est la suivante :

```
(1) void creer_point(char nom, float x, float y)
(2)    {
(3)    Point *prec, *pp;
(4)    prec = NULL;
(5)    pp = points;
(6)    while (pp != NULL && pp->nom != nom)
(7)       {
(8)       prec = pp;
(9)       pp = pp->suivant;
(10)      }
(11)   if (pp != NULL)
(12)      {
(13)      printf("Le point %c existe deja !\n", nom);
(14)      return;
(15)      }
(16)   pp = (Point *) malloc(sizeof(Point));
(17)   if (pp == NULL)
(18)      {
(19)      printf("Allocation impossible !");
(20)      exit(1);
(21)      }
```

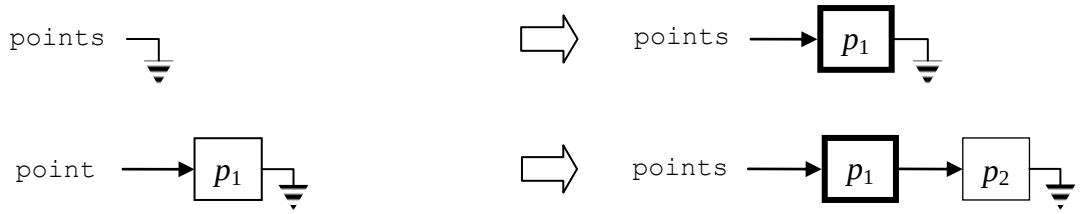
```
(22)    pp->nom = nom;
(23)    pp->x = x;
(24)    pp->y = y;
(25)    if (prec == NULL)
(26)       points = pp;
(27)    else
(28)       prec->suivant = pp;
(29)    pp->suivant = NULL;
(30)    printf("Le point %c a ete cree !\n", nom);
(31)    }
```

Cette fonction a pour arguments le nom, l'abscisse et l'ordonnée du point à créer. S'il n'existe pas déjà un point de même nom, elle alloue un emplacement en mémoire au point à créer et l'accroche à la fin de la liste chaînée. La procédure est la suivante (voir Figure 8.2.a) :

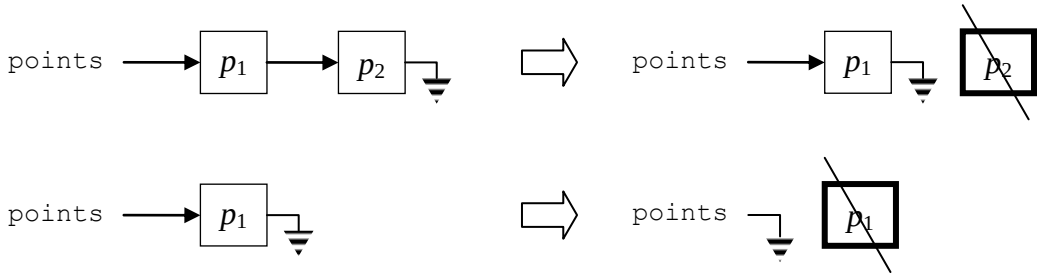
1. On parcoure la liste chaînée depuis son début (`points`) en mémorisant l'adresse du point courant (`pp` initialisée à `points`) et l'adresse du point qui le précède (`prec` initialisée à `NULL`) tant que la fin n'a pas été atteinte et que le nom du point courant est différent de celui du point à créer (lignes 4 à 10).
2. S'il existe un point ayant le nom du point à créer (`pp` pointe sur lui et n'est donc pas égal à `NULL`) : le point à créer ne peut pas être créé (lignes 11 à 15).
3. Sinon, on alloue dynamiquement une variable de type `Point` et d'adresse `pp` et on lui affecte le nom, l'abscisse et l'ordonnée du point à créer (lignes 16 à 24).
4. Si la liste chaînée est vide (`prec` est égal à `NULL`, sa valeur initiale) le point créé est aussi le premier de la liste chaîné, on fait pointer `points` sur le point créé (ligne 26).
5. Sinon, on fait pointer le dernier point de la liste chaînée (pointé par `prec`) sur le point créé (ligne 28).
6. Le point créé étant le dernier de la liste, il n'a pas de suivant : on le fait pointer sur rien (ligne 29). Le point créé est maintenant accroché à la fin de la liste chaînée.

La suppression d'un point est un peu plus compliquée parce que le chaînage est monodirectionnel. Elle est réalisée par la fonction `supprimer_point` dont la définition est la suivante :

```
(1)  void supprimer_point(char nom)
(2)     {
(3)     Personne *prec, *pp;
(4)     prec = NULL;
(5)     p = points;
(6)     while (p != NULL && pp->nom != nom)
(7)        {
(8)        prec = pp;
(9)        pp = pp->suivant;
(10)       }
(11)    if (pp == NULL)
(12)       {
(13)       printf("Le point %c n'existe pas !\n", nom);
(14)       return;
(15)       }
(16)    if (prec == NULL)
(17)       points = pp->suivant;
(18)    else
(19)       prec->suivant = pp->suivant;
```



(a) *insertion*



(b) *suppression*

Figure 8.2. *Insertion et suppression d'un point*

```
(20)    free(pp);
(21)    printf("Le point %c a ete supprime !\n", nom);
(22)    }
```

Cette fonction a pour arguments le nom du point à supprimer. Si ce point existe, elle le décroche de la liste chaînée et libère la place qu'il occupait en mémoire. Elle opère selon la procédure suivante (voir Figure 8.2.b) :

1. On parcoure la liste chaînée de son début (`points`) en mémorisant l'adresse du point courant (`pp` initialisée à `points`) et l'adresse du point qui le précède (`prec` initialisée à `NULL`) tant que la fin n'a pas été atteinte et que le nom du point courant est différent de celui du point à créer (lignes 4 à 10).
2. S'il n'existe pas de point ayant le nom du point à supprimer (`pp` est égal à `NULL`, puisque la fin de la liste a été atteinte) : le point à supprimer ne peut pas être supprimé (lignes 11 à 15).
3. Si le point à supprimer (pointé par `pp`) est le premier de la liste chaînée (`prec` est égal à `NULL`, sa valeur initiale), on fait pointer `points` sur le point qui suit le point à supprimer ou sur rien si le point à supprimer est aussi le dernier de la liste chaînée : le point à supprimer a maintenant été décroché de la liste chaînée (ligne 17).
4. Sinon, on fait pointer le point (pointé par `prec`) qui précède le point à supprimer sur le point qui suit le point à supprimer ou sur rien si le point à supprimer est le dernier de la liste chaînée : le point à supprimer a maintenant été décroché de la liste chaînée (ligne 19).
5. On libère l'emplacement mémoire occupé par le point à supprimer (ligne 20).

Pour afficher les points de l'ensemble de points, il suffit de parcourir la liste chaînée de son début à sa fin. Cet affichage est réalisé par la fonction `afficher_points` dont la définition est la suivante :

```
void afficher_points(void)
   {
   Point *pp;
   printf("Ensemble des points\n");
   pp = points;
   while (pp != NULL)
      {
      printf("%c, %.2f %.2f\n", pp->nom, pp->x, pp->y);
      pp = pp->suivant;
      }
   }
```

La fonction `main` crée un ensemble vide de points. Elle demande la création de quatre points dont deux fois un point de même nom puis affiche l'ensemble des points créés. Elle demande la suppression de deux points dont deux fois le même puis elle affiche l'ensemble des points restants. Sa définition est la suivante :

```
int main(void)
   {
   points = NULL;
   creer_point('A', 2.5, 3);
   creer_point('B', 5.25, 8);
   creer_point('C', 12.5, 5.5);
   creer_point('A', 0, 0);
   afficher_points();
   supprimer_point('A');
   supprimer_point('C');
   supprimer_point('A');
   afficher_points();
   return 0;
   }
```

Assemblons ce programme :

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```
Définition du type `Point` *et de la variable points*
Définition de la fonction `creer_point`
Définition de la fonction `supprimer_point`
Définition de la fonction `afficher_points`
Définition de la fonction `main`

et exécutons-le :

```
Le point A a ete cree !
Le point B a ete cree !
Le point C a ete cree !
Le point A existe deja !
Ensemble des points
A (2.50 ; 3.00)
B (5.25 ; 8.00)
C (12.50 ; 5.50)
Le point A a ete supprime !
Le point C a ete supprime !
Le point A n'existe pas !
Ensemble des points
B (5.25 ; 8.00)
```

		Sexe		
		F	M	Total
Catégorie	A	29	81	110
	B	65	111	176
	C	74	176	250
	Total	168	368	536

Figure 8.3. *Le tableau croisé Catégorie × Sexe*

8.5.2 **Tableau croisé**

Soit *E* et *F* deux ensembles de classes. Il s'agit de construire un tableau croisé *E* × *F* permettant de faire des statistiques sur la répartition d'un ensemble d'entités par classe de *E* et par classe de *F*.

Par exemple, la répartition des employés d'un établissement de la fonction publique par catégorie de personnel (A, B, C) et par sexe (F : féminin, M : masculin) peut être décrite par le tableau croisé de la figure 8.3.

Si l'ensemble *E* est composé de *m* classes E_0, ..., E_{m-1} et l'ensemble *F* de *n* classes F_0, ..., F_{n-1}, le tableau croisé sera implanté sous la forme d'un tableau, alloué dynamiquement, de *m* + 1 éléments dont chacun pointe vers un tableau, alloué dynamiquement, de *n* + 1 éléments de type `int` représentant une ligne du tableau croisé. Les noms (une lettre) des lignes du tableau croisé seront stockés dans un tableau de *m* caractères, alloué dynamiquement. Les noms (une lettre) des colonnes du tableau croisé seront stockés dans un tableau de *n* caractères, alloué dynamiquement.

Cette implantation est illustrée par la figure 8.4 où :

- `tc` pointe sur le tableau des pointeurs sur les lignes ;
- `tnl` pointe sur le tableau des noms de ligne ;
- `tnc` pointe sur le tableau des noms de colonne ;
- *a* = nom de la classe E_i
- *b* = nom de la classe F_j
- *c* = nombre d'entités appartenant aux classes E_i et F_j
- *d* = nombre d'entités appartenant à la classe E_i
- *e* = nombre d'entités appartenant à la classe F_j
- *f* = nombre d'entités total

Notons que `tc[`*i*`][`*j*`]` aura pour valeur le nombre contenu dans la case (*i*, *j*) du tableau croisé. En effet, l'expression `tc[`*i*`][`*j*`]` est équivalente à l'expression `*(*(tc + i) + j)` qui a pour valeur le nombre contenu dans le `j`^{e} élément du tableau pointé par le `i`^{e} élément du tableau pointé par `tc`. La variable `tc` doit donc être de type `int **`.

Le programme sera constitué des fonctions suivantes :

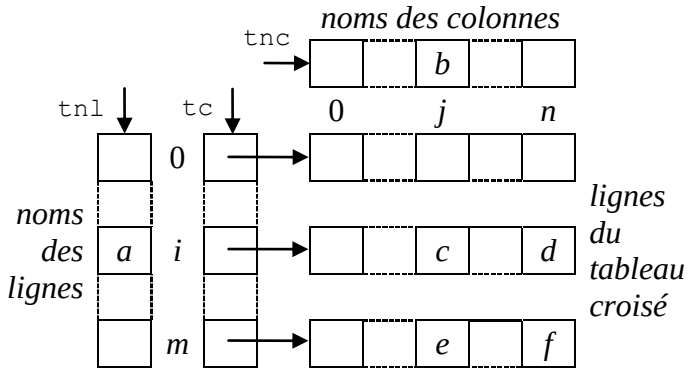


Figure 8.4. *Représentation du tableau croisé Catégorie × Sexe*

- `erreur_allocation`
- `allouer_tableau_croise`
- `allouer_noms`
- `saisir_tc`
- `accumuler_tc`
- `afficher_tc`
- `main`

La fonction `erreur_allocation` est appelée en cas d'impossibilité d'allocation d'un des tableaux. Cet appel provoque la sortie du programme.

```
void erreur_allocation(void)
   {
   printf("Allocation impossible !");
   exit(1);
   }
```

La fonction `allouer_tableau_croise` reçoit en arguments le nombre `m` de classes de *E* et le nombre `n` de classes de *F*. Elle alloue le tableau des pointeurs sur les lignes, puis chacune des lignes. Elle retourne l'adresse de l'élément de rang 0 du tableau des pointeurs sur les lignes, qui sera le point d'entrée dans le tableau croisé.

```
int **allouer_tableau_croise(int m, int n)
   {
   int i, **tc;
   tc = (int **) malloc(m * sizeof(int *));
   if (tc == NULL)
      erreur_allocation();
   for (i = 0; i < m; i++)
      {
      tc[i] = (int *) malloc(n * sizeof(int));
      if (tc[i] == NULL)
         erreur_allocation();
      }
   return tc;
   }
```

La fonction `allouer_noms` alloue un tableau de `k` éléments de type `char`. Elle sera appelée pour allouer le tableau des noms de ligne ou celui des noms de colonne.

```
char *allouer_noms(int k)
   {
   char *noms;
   noms = (char *) malloc(k * sizeof(char *));
   if (noms == NULL)
      erreur_allocation();
   return noms;
   }
```

La fonction `saisir_tc` demande à l'utilisateur de saisir le nombre d'entités pour chaque couple de classe (E_i, F_j) et le range dans la case (i, j) du tableau croisé `tc` (`tc[`i`][`j`]`). De plus, elle initialise à 0 les dernières cases de la dernière ligne et de la dernière colonne du tableau croisé `tc`.

```
void saisir_tc(int m, int n, int **tc, char *tnl, char *tnc)
   {
   int i, j;
   for (i = 0; i < m; i++)
      {
      for (j = 0; j < n; j++)
         {
         printf("%c x %c ? ", tnl[i], tnc[j]);
         scanf("%d", &tc[i][j]);
         }
      tc[i][n] = 0;
      }
   for (j = 0; j <= n; j++)
      tc[m][j] = 0;
   }
```

La fonction `accumuler_tc` calcule le nombre d'entités par ligne, le nombre d'entités par colonne, le nombre d'entités total et affecte ces nombres aux cases correspondantes de la dernière colonne et de la dernière ligne du tableau croisé `tc` :

```
void accumuler_tc(int m, int n, int **tc)
   {
   int i, j;
   for (i = 0; i < m; i++)
      {
      for (j = 0; j < n; j++)
         {
         tc[i][n] += tc[i][j];
         tc[m][j] += tc[i][j];
         }
      tc[m][n] += tc[i][n];
      }
   }
```

La fonction `afficher_tc` affiche le tableau croisé `tc`.

```
void afficher_tc(int m, int n, int **tc, char *tnl, char *tnc, int lc)
   {
   int i, j;
   printf("     ");
   for (j = 0; j < n; j++)
      printf("%*c", lc, tnc[j]);
```

```
   printf("%*s\n", lc, "Total");
   for (i = 0; i < m; i++)
      {
      printf("    %c", tnl[i]);
      for (j = 0; j <= n; j++)
         printf("%*d", lc, tc[i][j]);
      printf("\n");
      }
   printf("Total");
   for (j = 0; j <= n; j++)
      printf("%*d", lc, tc[m][j]);
   printf("\n");
   }
```

Dans une spécification de conversion, la longueur d'un champ peut ne pas être spécifiée dans le format, mais calculée à l'exécution. En ce cas, dans le format, cette longueur devra être remplacée par `*` et l'expression calculant cette longueur devra précéder immédiatement l'expression dont la valeur est à afficher dans ce champ. Ainsi l'instruction `printf("%*d", lc, tc[m][j]);` affichera le nom de la colonne `j` dans un champ ayant pour largeur `lc`.

La fonction `main` construit et affiche le tableau croisé *Catégorie* × *Sexe* présenté ci-dessus :

```
int main(void)
   {
   int **tc;
   char *tnl, *tnc;
   tc = allouer_tableau_croise(4, 3);
   tnl = allouer_noms(3);
   tnc = allouer_noms(2);
   tnl[0] = 'A';
   tnl[1] = 'B';
   tnl[2] = 'C';
   tnc[0] = 'F';
   tnc[1] = 'M';
   printf("Nombre d'employes par categorie et par sexe\n");
   saisir_tc(3, 2, tc, tnl, tnc);
   accumuler_tc(3, 2, tc);
   printf("Tableau croise\n");
   afficher_tc(3, 2, tc, tnl, tnc, 6);
   return 0;
   }
```

Assemblons ce programme :

```
#include <stdio.h>
#include <stdlib.h>
```
Définition de la fonction `erreur_allocation`
Définition de la fonction `allouer_tableau_croise`
Définition de la fonction `allouer_noms`
Définition de la fonction `saisir_tc`
Définition de la fonction `accumuler_tc`
Définition de la fonction `afficher_tc`
Définition de la fonction `main`

et exécutons le :

```
Nombre d'employes par catégorie et par sexe
A x F ? 29↵
A x M ? 81↵
B x F ? 65↵
B x M ? 111↵
C x F ? 74↵
C x M ? 176↵
Tableau croisé
          F     M Total
    A    29    81   110
    B    65   111   176
    C    74   176   250
Total   168   368   536
```

Exercices

Exercice 8.1.

1. Définir une fonction `echanger` qui échange les valeurs de deux variables de type `int` dont les adresses sont fournies en arguments.
2. Tester cette fonction en l'appelant dans la fonction `main` dans le corps de laquelle on devra définir deux variables `x` et `y`, demander à l'utilisateur de saisir une valeur pour `x` et une valeur pour `y`, échanger les valeurs de `x` et `y`, et enfin afficher les valeurs de `x` et `y`.

Exercice 8.2. On représente un vecteur de coordonnées (x_1 ; … ; x_n) dans un espace à n dimensions par un tableau de n éléments de type `float` contenant respectivement les coordonnées x_1, …, x_n.

1. Définir une fonction `produit_scalaire` qui reçoit en arguments un nombre `n` de type `int` et les adresses de deux tableaux `v1` et `v2` de `n` éléments de type `float` représentant chacun un vecteur dans un espace à `n` dimensions, et qui retourne le produit scalaire de ces deux vecteurs.
2. Tester la fonction `produit_scalaire` en l'appelant dans la fonction `main` pour calculer et afficher le produit scalaire de deux vecteurs définis dans la fonction `main` par :

   ```
   float vect1[2] = {2.5, 1.5}, vect2[2] = {3, -2};
   ```

   dont le produit scalaire est égal à 4,5.
3. Réaliser d'autres tests en faisant varier la dimension (`n`) et les coordonnées des vecteurs.

Exercice 8.3.

1. Définir une fonction `statistiques` qui reçoit en arguments : l'adresse `t` d'un tableau de flottants, le nombre `n` d'éléments de ce tableau et deux pointeurs `pm` et `pe` sur une variable de type flottant et qui affecte aux variables pointées par `pm` et `pe` respectivement la moyenne et l'écart-type des `n` nombres contenus dans le tableau `t`. Les formules permettant de calculer la moyenne et l'écart-type d'une série de valeurs ont été données dans l'énoncé de l'exercice 7.1.
2. Tester la fonction `statistiques` en l'appelant dans la fonction `main` pour calculer et afficher la moyenne et l'écart-type des notes des 12 étudiants de l'exemple du paragraphe 7.5.1.

Exercice 8.4. Il s'agit de définir une fonction permettant de trier un tableau de nombres entiers en spécifiant la relation d'ordre.

Soit *T* un tableau dont les éléments prennent leurs valeurs dans un ensemble *E* et *R* une relation d'ordre définie sur les éléments de *E*. Le tableau *T* est dit trié selon *R* si pour toute paire (e_1, e_2) d'éléments consécutifs de *T*, $val(e_1)$ *R* $val(e_2)$ est vraie. Un tableau de 0 ou 1 élément est toujours trié.

1. Définir une fonction d'en-tête `int infegal(int e1, int e2)` qui retourne 1 si `e1` est inférieur ou égal à `e2` et 0 sinon.
2. Définir une fonction d'en-tête `int supegal(int e1, int e2)` qui retourne 1 si `e1` est supérieur ou égal à `e2` et 0 sinon.
3. Définir une fonction `trier` qui reçoit en arguments le tableau à trier, le nombre d'éléments à trier et la relation d'ordre utilisée pour le tri : une fonction de type `int (*)(int, int)` et qui trie ce tableau conformément à cette relation d'ordre. La méthode de tri utilisée sera celle étudiée dans l'exercice 7.5.
4. Tester la fonction `trier` en l'appelant dans la fonction `main` pour trier le tableau défini dans la fonction main par :

   ```
   tab[12] = {68, 25, 5, 22, 21, 1, 4, 1, 130, 57, 68, 97}
   ```

   par ordre croissant (relation d'ordre `infegal`) puis par ordre décroissant (relation d'ordre `supegal`).

Exercice 8.5. Cet exercice est consacré à la représentation des polynômes à coefficients dans `int` et à une indéterminée, et au calcul de leur somme.

Un polynôme à coefficients dans *K* et à une indéterminée *x* est une expression de la forme suivante :

$$a_0x^0 + a_1x^1 + \ldots + a_nx^n$$

Les expressions a_ix^i où $a_i \in K$ et *i* est un entier naturel sont des monômes de coefficient a_i et de degré *i*. Le degré d'un polynôme est le degré de son monôme de plus haut degré et de coefficient non nul. Par exemple : $7 - 5x^2 + 2x^3$ est un polynôme de degré 3. Le polynôme 0 est le polynôme nul, qui par convention a le degré $-\infty$.

La somme de deux polynômes $a_0x^0 + a_1x^1 + \ldots + a_mx^m$ et $b_0x^0 + b_1x^1 + \ldots + b_nx^n$ est le polynôme $(a_0 + b_0)x^0 + (a_1 + b_1)x^1 + \ldots + (a_{max(m, n)} + b_{max(m, n)})x^{max(m, n)}$ en considérant qu'un polynôme de degré *d* a des monômes de degré > *d* dont les coefficients sont nuls.

Attention ! Le degré de la somme de deux polynômes de même degré *d* peut être inférieur à *d* car la somme des coefficients de deux monômes de même degré peut être nulle. Par exemple, la somme des polynômes $5 + x + 3x^2$ et $5 - 3x^2$ est le polynôme $10 + x$ de degré 1. □

Un polynôme de degré *n* sera représenté comme une instance du type `POLYNOME` déclaré par :

```
typedef struct
   {
   int degre;
   int *coefs;
   } POLYNOME;
```

et sera identifié par l'adresse de cette instance. Le champ `degre` a la valeur *n* et le champ `coefs` est un pointeur sur un tableau de $n + 1$ éléments dont l'élément de rang *i* ($0 \leq i \leq n$) a pour valeur le coefficient du monôme de degré *i*. Pour le polynôme nul, le champ `degre` et l'élément de rang 0 du tableau des coefficients auront la valeur 0.

1. Définir une fonction d'en-tête `POLYNOME *creer_polynome_vide(int d)` qui crée un polynôme de degré `d` à coefficients entiers non encore instanciés. On allouera un emplacement pour stocker le tableau des coefficients du polynôme, puis un emplacement pour stocker la structure représentant ce polynôme, et l'on affectera au champ `degre` de cette structure le degré `d`.
2. Définir une fonction `POLYNOME *saisir_polynome(int d)` qui crée un polynôme de degré `d` dont les coefficients sont demandés à l'utilisateur. On créera un polynôme vide de degré `d` puis on lui affectera les coefficients saisis par l'utilisateur. Si `d` n'est pas nul, il faudra vérifier qu'un coefficient nul n'est pas affecté au monôme de degré `d`.
3. Définir une fonction d'en-tête `POLYNOME *somme(POLYNOME *p1, POLYNOME *p2)` qui retourne l'adresse du polynôme somme des polynômes `p1` et `p2`. On créera un polynôme vide ayant le degré requis puis on lui affectera ses coefficients conformément à la définition de la somme de deux polynômes donnée ci-dessus.
4. Définir une fonction `main` qui crée deux polynômes dont les coefficients sont saisis par l'utilisateur, affiche ces polynômes, calcule leur somme et l'affiche. Tester cette fonction sur des paires de polynômes variés quand à leurs degrés et leurs coefficients. Par exemple sur les trois paires de polynômes suivants :
   - $1 - 2x + 3x^4$ et $1 + 2x$ dont la somme est le polynôme $2 + 3x^4$;
   - $5 + x + 3x^2$ et $5 - 3x^2$ dont la somme est égale à $10 + x$;
   - $5x$ et $-5x$ dont la somme est le polynôme nul.

Pour afficher un polynôme, on utilisera la fonction `afficher_polynome` qui fait appel à la fonction `afficher_monome`. Les définitions de ces deux fonctions sont les suivantes :

```
void afficher_monome(int c, int d)
   {
   if (d == 0)
      printf("%d", c);
   if (c == 1 && d == 1)
      printf("x");
   if (c > 1 && d == 1)
      printf("%dx", c);
   if (c == 1 && d > 1)
      printf("x%d", d);
   if (c > 1 && d > 1)
      printf("%dx%d", c, d);
   }

void afficher_polynome(POLYNOME *p)
   {
   int d, i, c;
   d = p->degre;
   i = 0;
   if (d > 0)
      while (p->coefs[i] == 0)
         i++;
   c = p->coefs[i];
   if (c < 0)
      printf("-");
   afficher_monome(abs(c), i);
```

```
for (i = i + 1; i <= d; i++)
   {
   c = p->coefs[i];
   if (c != 0)
      {
      if (c < 0)
         printf(" - ");
      else
         printf(" + ");
      afficher_monome(abs(c), i);
      }
   }
printf("\n");
}
```

Par exemple, le polynôme $7 + x - 5x^3$ sera affiché sous la forme `7 + x - 5x2 + 2x4`, le polynôme $-3 + x^2$ sous la forme `-3 + x2` et le polynôme nul sous la forme `0`.

9
Chaînes de caractères

La plupart des applications informatiques nécessitent la prise en compte de données textuelles. Les langages de programmation traitent ces données sous la forme de chaînes de caractères. Une chaîne de caractères est une suite éventuellement vide de caractères. Par exemple, un mot, une phrase, le titre d'un livre sont des chaînes de caractères.

En C, contrairement à d'autres langages de programmation, les chaînes de caractères ne constituent pas une structure de données à part. Elles sont représentées et manipulées comme des tableaux de caractères.

Un problème important est celui de la norme de codage utilisée pour coder les caractères d'une chaîne. Les normes les plus couramment utilisées sont :

- l'ASCII de base dont le jeu de caractères (voir Figure 8.1) comporte 128 caractères, numérotés de 0 à 127 et codés sur 1 octet. Le caractère 0 est le caractère nul, les caractères 1 à 31 et 127 sont des caractères de contrôle (tabulation, saut de ligne, retour chariot, etc.) non imprimables et les caractères 32 à 127 sont les chiffres, les lettres majuscules et minuscules non accentuées et des caractères spéciaux (signes de ponctuation, opérateurs arithmétiques, etc.).
- les normes ASCII étendues (ISO-8859-*n*) qui utilisent le 8^{e} octet laissé libre par l'ASCII de base pour coder les lettres accentuées et les caractères spéciaux des langues européennes. La norme ISO-8859-15, par exemple, couvre tous les caractères des langues européennes occidentales (allemand, anglais, espagnol, français, italien, etc.).
- la norme ISO 10646 qui, en accord avec l'Unicode, définit un jeu de caractères universel : l'UCS (Universal Character Set) qui compte actuellement une centaine de milliers de caractères couvrant pratiquement toutes les langues du monde ainsi que les caractères mathématiques ou graphiques. Les caractères sont numérotés à partir de 0. Les caractères de 0 à 127 sont ceux de l'ASCII de base. Le codage le plus classique de l'UCS est l'UTF-8 qui est un codage en longueur variable. Les caractères de 0 à 127, et donc ceux de l'ASCII, sont codés sur 1 octet et les autres sur 2 à 4 octets.

Classiquement, en C, et c'est ce que nous supposerons dans ce chapitre, les chaînes de caractères sont représentées comme des tableaux d'éléments de type `char` et sont manipulées en utilisant les fonctions de la bibliothèque standard de C qui sont déclarées dans le fichier d'en-têtes `string.h`. Ces fonctions, ainsi que celles de lecture ou d'écriture de chaînes de caractères déclarées dans le fichier d'en-têtes `stdio.h`, ne sont compatibles qu'avec des caractères codés sur 1 octet selon la norme ASCII de base ou une norme ASCII étendue. Il faudra donc veiller à ne pas les utiliser pour manipuler des chaînes dont les caractères ne sont pas codés selon ces normes. Pour manipuler des chaînes de caractères codés selon les différentes formes de codage de l'UCS, l'UTF-8 notamment, il faut utiliser des fonctions spécifiques[1].

[1] Par exemple celles déclarées dans le fichier d'en-têtes `wchar.h`.

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

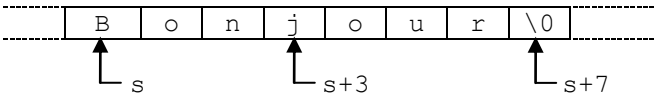
Figure 9.1. *Jeu de caractères de l'ASCII de base*

Dans ce chapitre, nous étudierons : la représentation d'une chaîne de caractères au sein d'un tableau (paragraphe 9.1) ; la lecture et l'écriture d'une chaîne de caractères (paragraphe 9.2) ; la manipulation de chaînes de caractères en utilisant les fonctions de la bibliothèque standard (paragraphe 9.3). Nous mettrons ensuite en œuvre ces notions en programmant un éditeur de texte très simplifié (paragraphe 9.4).

9.1 Représentation d'une chaîne de caractères

En C, une chaîne de caractères est rangée dans un tableau de caractères. Sa fin est marquée par le caractère de code 0, que nous appellerons le caractère nul et que nous noterons `\0`. Une chaîne de caractères est identifiée par l'adresse de son premier caractère dans ce tableau. Donc par la suite, si *e* est une expression de type « adresse d'un caractère », l'expression « la chaîne de caractères *e* » désignera la chaîne de caractères qui débute à l'adresse *e*.

Soit, par exemple, un tableau de caractères `s` dans lequel la chaîne de caractères « Bonjour » a été rangée à partir de l'élément de rang 0 et donc à partir de l'adresse `s`, comme le montre la figure suivante :



La chaîne de caractères `s` est « Bonjour », la chaîne de caractères `s + 3` est « jour » et la chaîne de caractères `s + 7` est vide.

Une constante littérale de type caractère est notée en plaçant ce caractère entre apostrophes (`'`). Par exemple : `'A'`, `'9'`, `';'`.

Une constante littérale de type chaîne de caractères est notée en plaçant cette chaîne de caractères entre guillemets (`"`). Par exemple :

```
"Bonjour"
```

Dans une constante littérale de type caractère ou chaîne de caractères, tout caractère imprimable ou non, peut être représenté par un caractère ou un nombre octal précédé du caractère d'échappement « \ » (anti-slash). Entre autres :

- `\n` nouvelle ligne (LF)
- `\r` retour chariot (CR)
- `\t` tabulation
- `\'` apostrophe (pour la distinguer du délimiteur de constante littérale caractère),
- `\"` guillemet (pour le distinguer du délimiteur de constante littérale chaîne de caractères)
- `\\` anti-slash (pour le distinguer du caractère d'échappement)
- `\0` caractère nul (caractère ayant le code ASCII 0)

Par exemple, la constante littérale `"Il a dit : \"Bonjour\""` représente la chaîne de caractères : « Il a dit : "Bonjour" ».

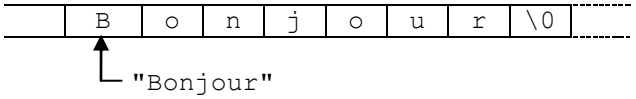
Une chaîne de caractères peut être initialisée lors de la définition du tableau de caractères dans lequel elle est rangée. Par exemple, la chaîne de caractères « Bonjour » aurait pu être placée dans le tableau `s` lors de sa définition :

```
char s[8] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

Cette écriture étant très lourde, l'écriture abrégée suivante est autorisée :

```
char s[8] = "Bonjour";
```

La valeur d'une constante littérale de type chaîne de caractères est l'adresse à partir de laquelle le compilateur a rangé cette chaîne de caractères. Par exemple :



Quelques pièges à éviter

Il ne faut pas confondre un caractère et une chaîne de caractères. Par exemple :

- `'A'` est le caractère « A » ;
- `"A"` est la chaîne de caractères composée du seul caractère « A ».

On ne peut pas affecter une chaîne de caractères à une expression de type tableau de caractères, ni à une variable de type adresse d'un caractère. Par exemple, si `s` est un tableau de caractères défini par :

```
char s[8];
```

on ne peut pas écrire :

```
s = "Bonjour";
```

car `s` étant le nom d'un tableau, n'est pas une valeur gauche. Par contre, si la variable `s` est définie par :

```
char *s;
```

on peut écrire :

```
s = "Bonjour";
```

car `s` est une variable et donc une valeur gauche. Mais attention ! C'est l'adresse de la chaîne « Bonjour » (c.-à-d. celle de son caractère « B ») que l'on a affecté à `s` et non la chaîne « Bonjour » elle-même. Pour réaliser une telle affectation, il faut recopier la chaîne « Bonjour » caractère par caractère à partir de l'adresse `s`.

Enfin, une erreur classique est de comparer les adresses de début de deux chaînes de caractères en croyant comparer les chaînes de caractères elles-mêmes. Par exemple, si deux chaînes de caractères `s1` et `s2` sont définies par :

```
char s1[10] = "Bonjour", s2[10] = "Bonjour";
```

l'expression `s1 == s2` n'aura pas forcément la valeur 1, car `s1` et `s2` sont les adresses de début des chaînes qu'elles désignent et ne seront pas égales si ces deux chaînes ont été rangées à des adresses différentes. Pour obtenir le résultat désiré, il faut comparer ces deux chaînes caractère par caractère.

Pour copier une chaîne de caractères à partir d'une adresse ou pour comparer deux chaînes de caractères, le mieux est d'utiliser les fonctions de la bibliothèque standard que nous étudierons au paragraphe 9.3.

9.2 Lecture et écriture d'une chaîne de caractères

La lecture ou l'écriture d'une chaîne de caractères est réalisée en utilisant les fonctions d'entrées-sorties étudiées au chapitre 6 :

- `getchar`, `putchar`, `fgetc` et `fputc` pour lire ou écrire une chaîne caractère par caractère ;
- `gets`, `fgets`, `puts` et `fputs` pour lire une chaîne de caractères contenue dans une ligne d'un fichier ou pour écrire une chaîne de caractères dans un fichier ;
- `scanf`, `fscanf`, `printf` et `fprintf` pour lire ou écrire une chaîne de caractères avec la spécification de conversion `%s`.

La lecture d'une chaîne de caractères contenue dans une ligne d'un fichier nécessite de contrôler que la chaîne lue ne soit pas plus longue que le nombre d'éléments du tableau dans lequel elle doit être stockée. Les fonctions `scanf`, `fscanf` ou `gets` sont déconseillées pour cette raison car elles n'effectuent pas ce contrôle. La spécification de format `%s` a de plus l'inconvénient d'arrêter la lecture à la rencontre d'un caractère « espace », « tabulation » ou « nouvelle ligne ». Par exemple, si la chaîne enregistrée dans le fichier est « Bonne journée ! », sa lecture retournera la chaîne « Bonne ».

Une solution consiste à utiliser la fonction `fgets`. Rappelons que l'appel `fgets(`*s*`, `*n*`, `*f*`)` a pour effet de lire les caractères de la ligne suivante du fichier *f* et de les placer dans le tableau de caractères *s* jusqu'à ce que soit $n - 1$ caractères aient été lus, soit le caractère « nouvelle ligne » ait été lu, soit la fin du fichier ait été atteinte, puis d'ajouter le caractère nul à la fin de la chaîne *s*.

L'utilisation de la fonction `fgets` peut malgré tout poser deux problèmes. Le premier est qu'il n'est pas assuré que la marque de fin de ligne dans le fichier lu soit le caractère « nouvelle ligne » (LF). Ce peut-être aussi la séquence de caractères « retour chariot », « nouvelle ligne » (CRLF) comme dans les fichiers Windows. Normalement, les fonctions de la bibliothèque standard doivent se charger de réaliser les conversions nécessaires pour que la marque de fin de ligne apparaisse toujours comme étant le caractère « nouvelle ligne ». Mais il s'avère que ce n'est pas le cas dans certaines implantations de C. Le second problème est le fait que la marque de fin de ligne soit ajoutée à la chaîne de caractères lue. On peut bien entendu enlever

cette marque en la remplaçant par le caractère nul, mais là encore il est nécessaire de savoir si cette marque est CR ou CRLF.

Nous allons, à titre d'exercice, définir une fonction `lire_ligne` qui améliore la fonction `fgets` en prenant en compte les deux types de marque de fin de fichier, en tronquant les lignes qui déborderaient du tableau dans lequel elles doivent être stockées et en ne rajoutant pas les marques de fin de ligne à la chaîne de caractères lue. La définition de cette fonction est la suivante :

```
char *lire_ligne(char *ligne, int tligne, FILE *f)
   {
   char c;
   int i = 0;
   while ((c = fgetc(f)) != EOF && c != '\n' && c != '\r')
      if (i < tligne - 1)
         ligne[i++] = c;
   ligne[i] = '\0';
   if (c == '\r')
      if ((c = fgetc(f)) == EOF || c != '\n')
         return NULL;
   if (c == EOF && i == 0)
      return NULL;
   return ligne;
   }
```

Cette fonction lit la ligne suivante du fichier `f` caractère par caractère. Tant que le caractère lu n'est ni la fin de fichier, ni le caractère LF, ni le caractère CR, il est ajouté à la fin de la chaîne `ligne` si le nombre de caractères lus est inférieur à `tligne - 1` (pas de débordement). A la sortie de cette itération, la fin de la ligne a été atteinte, le caractère nul est ajouté à la chaîne `ligne`. Si le dernier caractère lu est CR, le caractère suivant doit être LF, sinon `NULL` est retournée (erreur). Si `EOF` a été lu et qu'aucun caractère n'a été ajouté à la chaîne `s`, `NULL` est retournée (fin de fichier). Dans les autres cas (ligne terminée par LF ou CRLF ou bien dernière ligne non vide terminée par `EOF`), `ligne` est retournée.

Exemple 9.1. Soit le fichier `haiku.txt` dont le contenu est le suivant[2] :

```
Un vieil étang,
Une grenouille saute,
Le bruit de l'eau.
```

Que les deux premières lignes soient terminées par CR ou CRLF et que la dernière ligne le soit terminée par CR, CRLF ou la fin de fichier, l'instruction :

```
{
char cc[20];
FILE *f;
f = fopen("haiku.txt", "r");
if (f == NULL)
   {
   printf("Le fichier haiku.txt ne peut pas etre ouvert !");
   exit(1) ;
   }
while (lire_ligne(cc, N, f) != NULL)
   printf("%s\n", cc);
fclose(f);
}
```

[2] Haiku du poète japonais Basho Matsuo (1644-1694).

.

affichera, si N ≥ 20 :

```
Un vieil étang,
Une grenouille saute,
Le bruit de l'eau.
```

et, si N = 15 :

```
Un vieil étang
Une grenouille
Le bruit de l'
```

□

9.3 Manipulation d'une chaîne de caractères

Les chaînes de caractères étant des tableaux, elles peuvent être manipulées en tant que tels (voir chapitre 7, paragraphe 7.1). Par exemple, la longueur d'une chaîne de caractères `s` peut être calculée par la fonction `longueur` dont la définition est la suivante :

```
int longueur(char *s)
   {
   int i;
   i = 0;
   while (s[i] != '\0')
      i++;
   return i;
   }
```

Cette définition exploite le fait que si les caractères d'une chaîne sont numérotés à partir de 0, la longueur de cette chaîne est égale à la position du caractère nul.

Cependant, il est plus pratique et particulièrement recommandé d'utiliser les fonctions de manipulation de chaînes de caractères de la bibliothèque standard, déclarées dans le fichier d'en-têtes `string.h`. Ces fonctions permettent notamment de calculer la longueur d'une chaîne de caractères, de comparer deux chaînes de caractères, de copier une chaîne de caractères à partir d'un adresse, de déplacer des suites de caractères.

Attention ! Une chaîne de caractères étant stockée dans un tableau, il est important de contrôler le non débordement de ce tableau avant d'exécuter une opération de copie, de concaténation ou de déplacement. □

9.3.1 **Longueur d'une chaîne de caractères**

La fonction `strlen` d'en-tête :

```
int strlen(const char *s)
```

retourne la longueur de la chaîne de caractères `s`.

Par exemple, l'appel `strlen("Bonjour")` a la valeur `7`. On notera que la déclaration de l'argument `s` est préfixée par `const`, empêchant ainsi la modification de la chaîne dont on calcule la longueur (voir chapitre 8, paragraphe 8.3).

9.3.2 **Comparaison**

La fonction `strcmp` d'en-tête :

```
int strcmp(const char *s1, const char *s2)
```

compare, selon l'ordre lexicographique, la chaîne de caractères `s1` avec la chaîne de caractères `s2`. Elle retourne un entier négatif, nul ou positif selon que la chaîne `s1` est respectivement inférieure, égale ou supérieure à la chaîne `s2`.

Par exemple :

- la valeur de l'expression `strcmp("cat", "chat")` est négative ;
- la valeur de l'expression `strcmp("chat", "chat")` est nulle ;
- la valeur de l'expression `strcmp("chat", "cat")` est positive.

9.3.3 **Copie.**

La fonction `strcpy` d'en-tête :

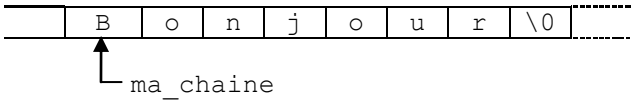
```
char *strcpy(char *s1, const char *s2)
```

copie la chaîne de caractères `s2` à partir de l'adresse `s1` et retourne `s1`.

Par exemple, après l'évaluation de l'appel :

```
strcpy(ma_chaine, "Bonjour")
```

la chaîne de caractères « Bonjour » aura été recopiée à partir de l'adresse `ma_chaine` :



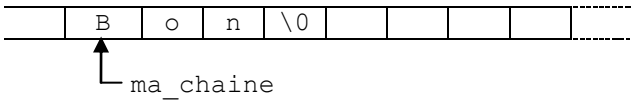
9.3.4 **Concaténation**

La fonction `strcat` d'en-tête :

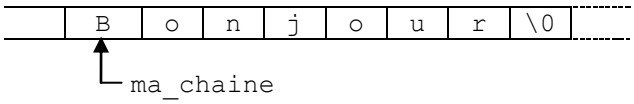
```
char *strcat(char *s1, const char *s2)
```

copie la chaîne de caractères `s2` à partir de l'adresse du caractère de fin de la chaîne `s1` et retourne l'adresse `s1`.

Supposons, par exemple, que la variable `ma_chaine` ait pour valeur l'adresse de début de la chaîne de caractères « Bon » :



Après l'appel `strcat(ma_chaine, "jour")` la chaîne `ma_chaine` est « Bonjour » :



Attention ! `s1` ne doit pas être l'adresse désignée par une constante littérale chaîne de caractères. Par exemple, l'expression `strcat("Bon", "jour")` est interdite. D'une manière générale, on de doit pas apporter de modifications à la zone mémoire contenant les constantes. Cet avertissement s'applique donc aussi à l'appel des fonctions `strcpy` et `memmove`.

9.3.5 **Copie et concaténation avec contrôle de débordement**

Les fonctions `strncpy` et `strncat` permettent de contrôler que les caractères copiés ou concaténés ne débordent de la place prévue pour les stocker. Elles ont pour en-têtes :

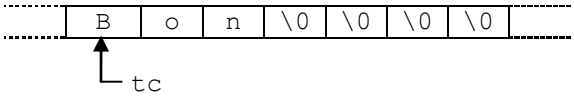
```
char *strncpy(char *s1, const char *s2, size_t n)
char *strncat(char *s1, const char *s2, size_t n)
```

Soit *k* le minimum de `n` et de `strlen(s2)`. La fonction `strncpy` copie, à partir de l'adresse `s1`, les *k* premiers caractères de la chaîne `s2` suivis de `n` – *k* caractères nuls et la fonction `strncat` copie les *k* premiers caractères de la chaîne `s2` suivis du caractère nul à partir de l'adresse du caractère de fin de la chaîne `s1`. Les deux fonctions retournent `s1`.

Il faut cependant faire attention en utilisant la fonction `strncpy`, car elle ne place un caractère nul à la fin de la chaîne recopiée que si `n` est supérieur à la longueur de cette chaîne. Par exemple, si `tc` est un tableau de 7 caractères, l'appel :

```
strncpy(tc, "Bonjour", 3)
```

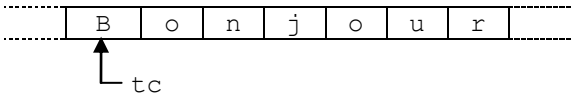
produira :



mais l'appel :

```
strncpy(tc, "Bonjour", 7)
```

produira :



9.3.6 **Déplacement**

La fonction `memmove` d'en-tête :

```
void *memmove(void *s1, const void *s2, size_t n)
```

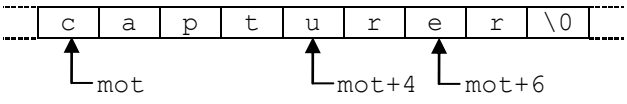
déplace à partir de l'adresse `s1`, la suite des `n` caractères qui débute à l'adresse `s2`. Elle retourne `s1`.

Cette fonction est très utile pour supprimer ou insérer une suite de caractères dans une chaîne de caractères, comme nous allons le montrer.

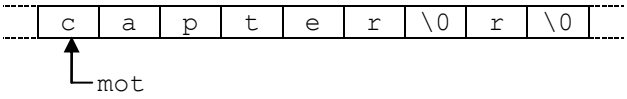
Attention ! Redisons-le. Avant d'effectuer une opération d'insertion, il faudra vérifier si la place est disponible pour la faire dans la chaîne de caractères dans laquelle cette insertion est réalisée. □

Suppression d'une sous-chaîne de caractères

Montrons, par exemple, comment transformer la chaîne de caractères « capturer » en la chaîne de caractères « capter ». Supposons, que cette chaîne de caractères soit rangée dans le tableau de caractères `mot` de la façon suivante :



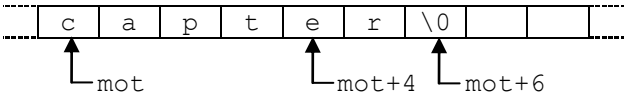
L'appel `memmove(mot + 4, mot + 6, 3)` décale la suite de caractères `er\0` de 2 cases vers la gauche, ce qui produit :



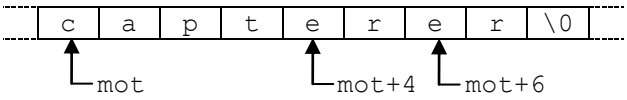
La chaîne de caractères `mot` est maintenant « capter ».

Insertion d'une sous-chaîne de caractères

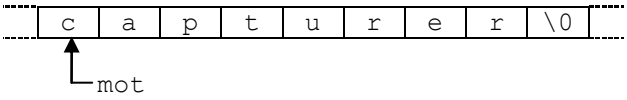
Réalisons l'opération inverse. Supposons que la chaîne de caractères débutant à l'adresse `mot` soit la chaîne de caractères « capter » :



L'appel `memmove(mot + 6, mot + 4, 3)` décale la suite de 3 caractères qui débute à l'adresse `mot + 4` de 2 cases vers la droite, ce qui produit :



et l'appel `memmove(mot + 4, "ur", 2)` copie la suite de 2 caractères « ur » à partir de l'adresse `mot + 4`, ce qui produit :



La chaîne de caractères `mot` est maintenant « capturer ».

9.4 Un petit éditeur de texte

Un éditeur de texte est un logiciel qui permet de créer un texte, d'élaborer sa mise en page, de le modifier, d'y faire des recherches, etc. Les éditeurs de texte actuellement disponibles sur les ordinateurs sont des outils extrêmement sophistiqués. L'éditeur que nous nous proposons de programmer est, lui, très élémentaire mais constitue un bon exercice de manipulation des chaînes de caractères. Il permet de créer et de modifier un texte au moyen de deux opérations :

- *ajouter* une chaîne de caractères à la fin du texte ;
- *remplacer* dans le texte toutes les occurrences d'une chaîne de caractères par une autre.

Pour programmer ces deux opérations deux fonctions de base, `couper_coller` et `chercher`, sont définies :

- l'appel `couper_coller(`*r*`, `*n*`, `*s*`)` où *r* et *n* sont des entiers ($0 \leq r \leq$ *longueur du texte* et $0 \leq n \leq$ *longueur du texte* $- r$) et *s* est une chaîne de caractères, supprime *n* caractères dans le texte à partir du rang *r* puis insère la chaîne de caractères *s* à partir de ce même rang. Le nom de cette fonction provient du fait qu'elle combine les deux opérations *couper* et *coller* présentes dans tous les éditeurs de texte. Par exemple, si le texte est : « Mes BD préférées », il sera transformé en : « Mes bandes dessinées préférées » par l'appel `couper_coller(4, 2, "bandes dessinées")`. On a coupé la chaîne « BD » puis collé la chaîne : « bandes dessinées ».

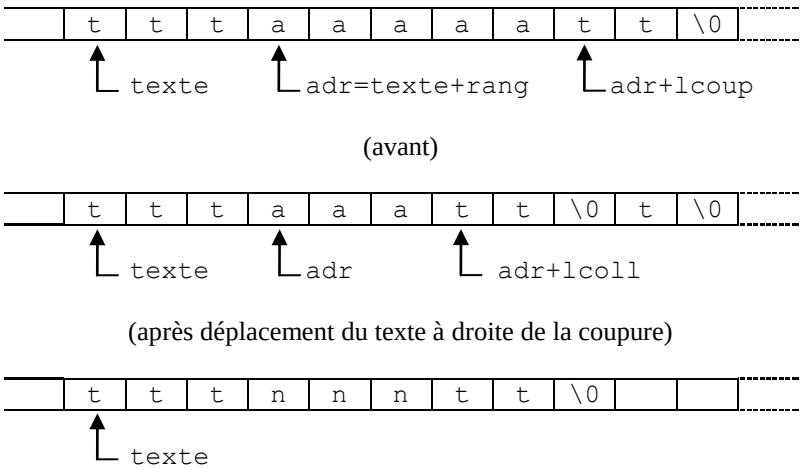


Figure 9.2. *Couper-coller*

– l'appel `chercher(`*r*`,` *m*`)` où *r* est un entier ($0 \leq r <$ *longueur du texte*) et *m* est une chaîne de caractères, retourne le rang dans le texte de la première occurrence de *m* ou bien –1 s'il n'en existe pas. Par exemple, si le texte est : « Mes BD préférées », l'appel `chercher(0, "BD")` retournera la valeur 4 et l'appel `chercher(0, "bandes dessinées")` retournera la valeur –1.

Notre éditeur est un programme C constitué de la façon suivante :

– un tableau `texte` de `TTEXTE` caractères contenant le texte à éditer ;
– une variable globale `ltexte` contenant la longueur courante du texte (caractère nul de fin de chaîne non compté) ;
– les fonctions de base `couper_coller` et `chercher` ;
– les fonctions `ajouter` et `remplacer` qui réalisent les opérations de même nom ;
– la fonction `main` qui constitue l'interface de l'éditeur et qui permet de répéter les opérations d'ajout et de remplacement autant de fois que nécessaire pour obtenir le texte définitif.

La fonction `couper_coller` est définie de la façon suivante :

```
void couper_coller(int rang, int lcoup, char *coll)
   {
   char *adr;
   int lcoll, delta;
   adr = texte + rang;
   lcoll = strlen(coll);
   delta = lcoll - lcoup;
   if (ltexte + delta >= TTEXTE)
      {
      printf("Texte trop long !\n");
      return;
      }
```

```
    memmove(adr + lcoll, adr + lcoup, ltexte - rang - lcoup + 1);
    memmove(adr, coll, lcoll);
    ltexte = ltexte + delta;
    }
```

L'appel de cette fonction a pour effet de supprimer (couper) dans le texte, `lcoup` caractères à partir de la position `rang` puis d'insérer (coller) à partir de cette position la chaîne `coll`. La variable `lcoll` a pour valeur la longueur de la chaîne à coller et la variable `delta` a pour valeur le nombre de caractères ajouté au texte par cette opération (`delta = lcoll - lcoup`). La procédure est illustrée par la figure 9.2 où `aaaaa` représente la sous-chaîne à couper et `nnn` la chaîne à coller. Soit `adr` l'adresse de début de la sous-chaîne à couper (`adr = texte + rang`). Une fois l'opération effectuée, la sous-chaîne qui suit la sous-chaîne à couper (marque de fin incluse) devra suivre immédiatement la chaîne qui a été collée. Il faut donc la déplacer de l'adresse `adr + lcoup` vers l'adresse `adr + lcoll`. C'est ce que l'on fait en appelant la fonction `memmove`. On insère ensuite, toujours en appelant la fonction `memmove`, la chaîne à coller (marque de fin exclue) à l'adresse `adr`.

Il faut de plus contrôler que le collage n'entraîne pas un débordement du tableau `texte`. Cela se produit si `ltexte + delta >= TTEXTE`. En ce cas, l'opération n'est pas réalisée et le message : « Texte trop long ! » est affiché.

La fonction `chercher` peut être définie de la façon suivante :

```
int chercher(char *modele, int rang)
   {
   int j;
   j = 0;
   while ((texte[rang] != '\0') && (modele[j] != '\0'))
      if (texte[rang + j] == modele[j])
         j++;
      else
         {
         rang++;
         j = 0;
         }
   if (modele[j] == '\0')
      return rang;
   else
      return -1;
   }
```

Cette fonction recherche dans le texte la première occurrence de la chaîne `modele` à partir de la position `rang` que l'on suppose supérieure ou égale à 0 et inférieure ou égale à la longueur de la chaîne `texte`. L'algorithme utilisé consiste à faire glisser la chaîne `modele` le long de la chaîne `texte` jusqu'à ce que chaque caractère de la chaîne `modele`, excepté le caractère nul, soit égal au caractère correspondant de la chaîne `texte` ou bien jusqu'à atteindre la fin de la chaîne `texte`. On initialise la variable `j` à 0 puis, tant que l'on pas atteint la fin du texte ou la fin du modèle, on teste si le caractère courant du texte (`texte[rang + j]`) est égal au caractère courant du modèle (`modele[j]`). Si oui, on passe au caractère suivant (`j++`) du texte et du modèle. Sinon, on passe au caractère suivant du texte (`rang++`) et l'on revient au premier caractère du modèle (`j = 0`) qui glisse donc d'un caractère. Si la fin du modèle a été atteinte, cela signifie que la chaîne `modele` est contenue dans la chaîne `texte` à partir de la position `rang` : on retourne cette position. Si la fin du texte a été atteinte cela signifie que la chaîne `modele` n'est pas contenue dans la chaîne `texte` : on retourne –1.

Notons qu'il existe des algorithmes plus performants mais que celui-ci est le plus simple à mettre en œuvre.

La définition de la fonction ajouter est très simple :

```
void ajouter(char *nouv)
   {
   couper_coller(ltexte, 0, nouv);
   }
```

Cette fonction ajoute la chaîne `nouv` à la fin du texte en appelant la fonction `couper_coller`.

La fonction `remplacer` peut être définie de la façon suivante :

```
void remplacer(char *anc, char *nouv)
   {
   int lanc, lnouv, rang;
   lanc = strlen(anc);
   lnouv = strlen(nouv);
   rang = chercher(anc, 0);
   while (rang != -1)
      {
      couper_coller(rang, lanc, nouv);
      rang = rang + lnouv;
      rang = chercher(anc, rang);
      }
   }
```

Cette fonction remplace dans le texte toutes les occurrences de la chaîne `anc` par la chaîne `nouv`. Les variables `lanc` et `lnouv` ont pour valeur les longueurs des chaînes `anc` et `nouv`. La variable `rang` a pour valeur le début dans le texte de la dernière occurrence de la chaîne `anc` ou –1 s'il n'y en a plus. La recherche de chaque occurrence de la chaîne `anc` est faite par la fonction `chercher` et le remplacement de cette occurrence par la chaîne `nouv` est fait par la fonction `couper_coller`. La recherche reprend toujours au caractère qui suit immédiatement la chaîne de remplacement.

C'est la fonction `main` qui constitue l'interface de l'éditeur. Sa définition est la suivante :

```
int main(void)
   {
   char commande, texte1[TTEXTE], texte2[TTEXTE];
   do
      {
      printf("\nTaper");
      printf("\nA pour Ajouter\nR pour Remplacer\nF pour Finir\n? ");
      lire_ligne(texte1, TTEXTE, stdin);
      commande = texte1[0];
      switch (commande)
         {
         case 'A':
            printf("Ajouter ? ");
            ajouter(lire_ligne(texte1, TTEXTE, stdin));
            break;
```

```
            case 'R':
               printf("Rechercher ? ");
               lire_ligne(texte1, TTEXTE, stdin);
               printf("Remplacer par ? ");
               lire_ligne(texte2, TTEXTE, stdin);
               remplacer(texte1, texte2);
               break;
            }
         printf("TEXTE: %s\n", texte);
         }
      while (commande != 'F');
      return 0;
      }
```

L'utilisateur dispose de trois commandes : A (Ajouter), R (Remplacer) et F (Finir). Il peut répéter autant de fois que nécessaire les deux premières. L'édition du texte se termine par la commande F. La commande A demande la chaîne à ajouter. La commande R demande la chaîne à remplacer et la chaîne de remplacement. Après l'exécution de chaque commande, le texte est affiché. Si la commande n'est ni A, ni R, ni F, le texte est affiché sans modification. Pour lire les chaînes de caractères à ajouter, à remplacer ou de remplacement il est fait appel à la fonction `lire_ligne` que nous avons définie au paragraphe 9.2. Il ne reste plus qu'à assembler le programme.

```
#include <stdio.h>
#include <string.h>
#define TTEXTE 1024
char texte[TTEXTE] = "";
int ltexte = 0;
```
définition de la fonction `couper_coller`
définition de la fonction `ajouter`
définition de la fonction `remplacer`
définition de la fonction `lire_ligne` ***et*** `main`
définition de la fonction `main`

Nous allons tester notre éditeur en éditant la première phrase de la préface du volume 1 du célèbre ouvrage de Donald E. Knuth : « The Art of Computer Programming » :

```
Taper
A pour Ajouter
R pour Remplacer
F pour Finir
? A↲
Ajouter ? The process of preparing programs for a digital computer is
especially attractive , not only because it can be economically and
scientifically rewarding , but also because it can be an esthetic
experience much like composing poetry or music.↲
TEXTE: The process of preparing programs for a digital computer is
especially attractive , not only because it can be economically and
scientifically rewarding , but also because it can be an esthetic
experience much like composing poetry or music.
```

Nous avons saisi, ce qui est incorrect, une espace devant chaque virgule. Nous corrigeons cette erreur en remplaçant la chaîne : « , » par la chaîne : « , » :

```
Taper
A pour Ajouter
R pour Remplacer
F pour Finir
? R↲
Rechercher ?  ,↲
Remplacer par ? ,↲
TEXTE: The process of preparing programs for a digital computer is
especially attractive, not only because it can be economically and
scientifically rewarding, but also because it can be an esthetic
experience much like composing poetry or music.
```

Il reste encore une erreur. Nous avons saisi : « esthetic » au lieu de : « aesthetic ». Nous corrigeons cette erreur :

```
Taper
A pour Ajouter
R pour Remplacer
F pour Finir
? R↲
Rechercher ? esthetic↲
Remplacer par ? aesthetic↲
TEXTE: The process of preparing programs for a digital computer is
especially attractive, not only because it can be economically and
scientifically rewarding, but also because it can be an aesthetic
experience much like composing poetry or music.
```

Enfin, il ne reste plus qu'à attribuer cette citation à son auteur :

```
Taper
A pour Ajouter
R pour Remplacer
F pour Finir
? A↲
Ajouter ?  (Donald E. Knuth, The Art of Computer Programming)↲
TEXTE: The process of preparing programs for a digital computer is
especially attractive, not only because it can be economically and
scientifically rewarding, but also because it can be an aesthetic
experience much like composing poetry or music. (Donald E. Knuth, The Art
of Computer Programming)
```

On tape « F » pour finir :

```
Taper
A pour Ajouter
R pour Remplacer
F pour Finir
? F↲
TEXTE: The process of preparing programs for a digital computer is
especially attractive, not only because it can be economically and
scientifically rewarding, but also because it can be an aesthetic
experience much like composing poetry or music. (Donald E. Knuth, The Art
of Computer Programming)
```

Exercices

Exercice 9.1. Ecrire un programme qui affiche, les mois de l'année dont les noms sont les plus courts et les mois de l'année dont les noms sont les plus longs. Les noms de mois seront stockés dans un tableau de 12 chaînes de caractères.

Exercice 9.2. Une chaîne de caractères est un palindrome si et seulement si elle est la même quand elle est lue de gauche à droite ou de droite à gauche. Par exemple, les chaînes de caractères :

- « ESOPERESTEICIETSEREPOSE » ;
- « ELUPARCETTECRAPULE » ;
- « TULASTROPECRASECESARCEPORTSALUT » ;

sont des palindromes.

1. Définir une fonction booléenne d'en-tête `int est_palindrome(char *cc)` qui retourne 1 (vrai) si la chaîne de caractères `cc` est un palindrome et 0 (faux) sinon.
2. Ecrire un programme qui lit une chaîne de caractères en utilisant la fonction `lire_ligne` définie au paragraphe 9.2 ci-dessus, puis teste, en appelant la fonction `est_palindrome`, si elle est un palindrome.
3. Modifier ce programme de telle façon qu'au lieu de lire la chaîne de caractères à tester, il la reçoive comme argument de la fonction `main`. Par exemple, si le fichier exécutable de ce programme a pour nom `palindrome`, son exécution devra pouvoir être lancée par la commande :

   ```
   palindrome ESOPERESTEICIETSEREPOSE
   ```

   si la chaîne de caractères à tester est : « ESOPERESTEICIETSEREPOSE ».

Exercice 9.3.

Pour enrichir notre petit éditeur de texte, on souhaite lui rajouter une troisième commande C (Compter) qui compte le nombre d'occurrences d'un modèle dans le texte. Définir une fonction d'en-tête `int compter(char *m)` qui sera appelée par cette commande et qui compte le nombre d'occurrences du modèle `m` dans le texte. Cette fonction utilisera la fonction `chercher`. Par exemple, l'appel `compter("it")` lorsque le texte est la dernière version de celui qui a servi à tester l'éditeur retournera 3.

Exercice 9.4.

Toujours dans le but d'enrichir notre éditeur de texte, on souhaite lui rajouter une quatrième commande N (normaliser) qui transforme le texte de façon à respecter les règles d'écriture suivantes des signes de ponctuation de l'anglais :

- pas d'espace avant et une espace après une virgule, un point-virgule, un deux-points, un point, un point d'interrogation, un point d'exclamation, une parenthèse fermante ;
- une espace avant et pas d'espace après une parenthèse ouvrante ;
- pas d'espace avant et après un tiret.

Définir une fonction d'en-tête `int normaliser(void)` qui est appelée par cette commande et qui normalise le texte en appliquant ces règles. Par exemple, l'appel `normaliser()` lorsque le texte est la première version de celui qui a servi à tester l'éditeur devra retourner la deuxième version de ce texte. On utilisera la fonction `couper_coller` et définir quatre fonctions auxiliaires suivantes : (i) ajouter une espace avant un signe de ponctuation, (ii) enlever une espace avant un signe de ponctuation, (iii) ajouter une espace après un signe de ponctuation et (iv) enlever tous les espaces après un signe de ponctuation ou après une espace.

10
Récursivité

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 factorielle est appelée dans sa propre définition.

Remarquons que le calcul de la factorielle selon cette définition se termine car n décroît 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 ou plusieurs 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 la plupart des langages de programmation, le langage C permet de définir des fonctions récursives, c.-à-d. des fonctions qui s'appellent elles-mêmes.

Dans ce chapitre, nous proposerons une démarche pour définir une fonction récursive (paragraphe 10.1) et nous montrerons comment se déroule, en mémoire, l'appel d'une telle fonction (paragraphe 10.2). Nous présenterons ensuite deux exemples très classiques de programmation récursive : le jeu des Tours de Hanoi et la recherche dichotomique d'une valeur dans un tableau trié (paragraphe 10.3).

10.1 Définition d'une fonction récursive

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

1. établir la relation de récurrence ;
2. établir les cas de base ;
3. vérifier que l'application de la relation de récurrence convergera vers l'un des cas de base ;
4. écrire en C la définition de la fonction qui devra tester si l'un des cas de base a été atteint, et relancer l'application de la relation de récurrence, si ce n'est pas le cas.

Appliquons cette démarche à la définition de la fonction *fac* qui appliquée à un nombre entier n ($n \geq 0$) calcule la factorielle de n.

1. Etablir la relation de récurrence.

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

2. Etablir les cas de base.

   $fac(n) = 1$, si $n = 0$

3. Vérifier que l'application de la relation de récurrence convergera vers l'un des cas de base.
   A chaque appel de la fonction *fac* on soustrait 1 à *n* qui deviendra donc égal à 0 au $(n + 1)^e$ appel, ce qui arrêtera la récursion puisque le cas de base sera atteint, pourvu que la condition $n \leq 0$ ait été respectée lors de l'appel initial.

4. Ecrire en C la définition de la fonction qui devra tester si l'un des cas de base a été atteint, et relancer l'application de la relation de récurrence, si ce n'est pas le cas.

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

Le cas de base a été étendu aux valeurs négatives de `n` (`if (n <= 0) return 1`) afin que le programme ne boucle pas indéfiniment si, suite à une erreur de programmation, la fonction `fac` est appelée avec une valeur négative de `n`.

La factorielle d'un nombre pouvant être très grande, le type `unsigned long` a été choisi comme type de retour de la fonction `fac` car il est le type entier ayant la plus grande extension en C conforme à la norme ANSI C89[1]. Lorsque l'extension du type `unsigned long` est l'extension minimum prévue par cette norme, soit 0 à 4 294 967 295 (voir chapitre 2, paragraphe 2.3), cette fonction ne pourra calculer que la factorielle d'un nombre inférieur ou égal à 12. En effet, 12! = 479 001 600 et 13! = 6 227 020 800. Si la fonction `fac` est appelée avec `n` > 12, une valeur sera tout de même retournée, car les opérations arithmétiques sur les entiers non signés représentés sur *n* bits sont effectuées modulo *n*, mais cette valeur sera bien évidemment incorrecte. Il faudra donc que le programme appelant vérifie que la fonction `fac` est appelée avec un argument valide.

Exemple 10.1. Le programme suivant calcule la factorielle d'un nombre saisi au clavier par l'utilisateur :

```
#include <stdio.h>

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

[1] La dernière norme ISO de C, dite C99, a rajouté un nouveau type entier, le type `long long int` dont les instances sont des nombres codés sur 64 bits et dont l'extension minimale est de −9 223 372 036 854 775 807 à +9 223 372 036 854 775 807 pour les entiers signés et de 0 à 18 446 744 073 709 551 615 pour les entiers signés. Son utilisation permettrait de calculer la factorielle d'un nombre de 0 à 20.

```
int main(void)
   {
   int n;
   do
      {
      printf("Entrer un nombre entier positif ? ");
      scanf("%d", &n);
      }
   while (n < 0);
   printf("%lu", fac(n));
   return 0;
   }
```

La spécification de conversion `%lu` est utilisée pour afficher un nombre de type `unsigned long` sous forme décimale non signée.

Testons ce programme pour plusieurs valeurs de `n` :

```
Entrer un nombre entier positif ? -1
Entrer un nombre entier positif ? 0
0 ! = 1

Entrer un nombre entier positif ? 20
12 ! = 479001600

Entrer un nombre entier positif ? 21
13 ! = 1932053504
```

Manifestement, la valeur retournée pour `n` = 13 est incorrecte (elle inférieure à 12 !). On pourra vérifier qu'elle est égale à 12 ! × 13 (mod 2^{32}) car dans l'implantation de C sur laquelle ce programme a été exécuté, les nombres entiers de type `unsigned long` sont représentés sur 32 bits. □

Appliquons maintenant notre démarche à la définition de 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. Etablir la 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. Etablir les cas de base.

   La somme des entiers compris entre *i* et *i* est égale à *i* :

   $somme(i, j) = i$, si $i = j$

3. Vérifier que l'application de la relation de récurrence convergera vers l'un des cas de base.

   A 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 sera atteint, pourvu que la condition $i \leq j$ ait été respectée lors de l'appel initial..

4. Ecrire en C la définition de la fonction qui devra tester si l'un des cas de base a été atteint, et relancer l'application de la relation de récurrence, si ce n'est pas le cas.

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

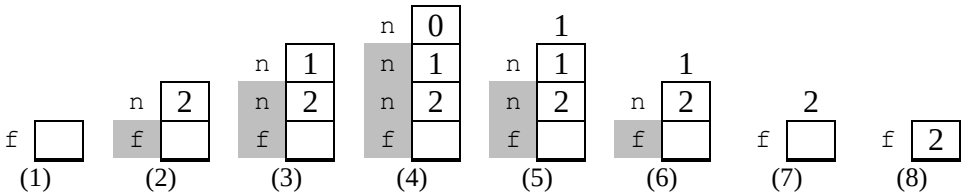


Figure 10.1. *Evaluation de* **`fac(2)`**

Le cas de base a été étendu aux valeurs de `i` supérieures à celles de `j` (`if (i > j) return 1`) afin que le programme ne boucle pas indéfiniment si, suite à une erreur de programmation, la fonction `somme` est appelée avec une valeur de `i` supérieure à celle de `j`.

Récursivité et déclarativité

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

```
unsigned long fac(int n)
   {
   int i;
   unsigned long f;
   f = 1;
   for (i = 1; i <= n; i++)
      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, qui par contre, produit souvent des programmes plus efficaces.

Malgré tout, pour être objectif, il faut dire que cette définition itérative de la factorielle correspond assez bien à la définition de $n!$ ($n > 0$) comme étant le produit des nombres entiers strictement positifs inférieurs ou égaux à n :

$$n! = \prod_{i=1}^{n} i \ (n > 0)$$

10.2 Appel d'une fonction récursive

L'appel d'une fonction récursive déclenche une succession d'appels de cette fonction jusqu'à atteindre l'un des cas de base à partir duquel il est possible de terminer l'évaluation de ces appels dans l'ordre inverse de leur déclenchement. C'est la pile qui conserve la mémoire des appels successifs de la fonction.

Considérons l'évolution de la pile au cours de l'exécution du programme de l'exemple 10.1. Les étapes sont les suivantes (voir figure 10.1) :

1. La fonction `main` a été appelée ; la variable `f` est empilée.

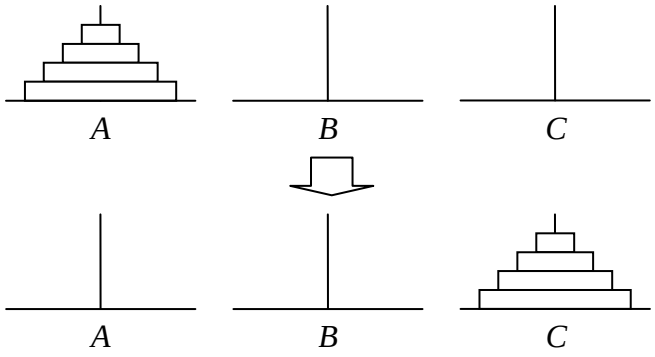


Figure 10.2

2. La fonction `fac` a été appelée avec l'argument effectif 2. La variable `n` est empilée avec la valeur 2.
3. La fonction `fac` a été appelée avec l'argument effectif 1. La variable `n` est empilée avec la valeur 1.
4. La fonction `fac` a été appelée avec l'argument effectif 0. La variable `n` est empilée avec la valeur 0.
5. L'exécution du corps de la fonction `fac` retourne la valeur 1 : la variable locale `n` est dépilée, la valeur 1 est empilée puis dépilée après avoir été substituée à l'expression `fac(0)`.
6. L'exécution du corps de la fonction `fac` retourne la valeur 1 : la variable locale `n` est dépilée, la valeur 1 est empilée puis dépilée après avoir été substituée à l'expression `fac(1)`.
7. L'exécution du corps de la fonction `fac` retourne la valeur 2 : la variable locale `n` est dépilée, la valeur 2 est empilée puis dépilée après avoir été substituée à l'expression `fac(2)`.
8. La valeur 2 est affectée à `f`.

10.3 Exemples

10.3.1 **Les tours de Hanoi**

Le jeu des tours de Hanoi est l'un des exemples les plus classiques de la puissance d'expression de la récursivité. On dispose de 3 tiges verticales *A, B* et *C,* sur lesquelles peuvent être enfilés des disques percés en leur milieu. Au départ 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 *C* en utilisant la tige *B* comme intermédiaire, par une suite de mouvements dont chacun consiste à déplacer un disque d'une tige à une autre sans jamais poser un disque sur un disque de taille plus petite (voir figure 10.2).

La solution n'est a priori pas évidente mais elle est étonnamment simple si l'on raisonne par récurrence :

– Si l'on sait déplacer, par une suite de mouvements valides, $n - 1$ disques d'une tige à une autre sur laquelle peuvent être enfilés des disques tous plus grand que ces $n - 1$ disques, en

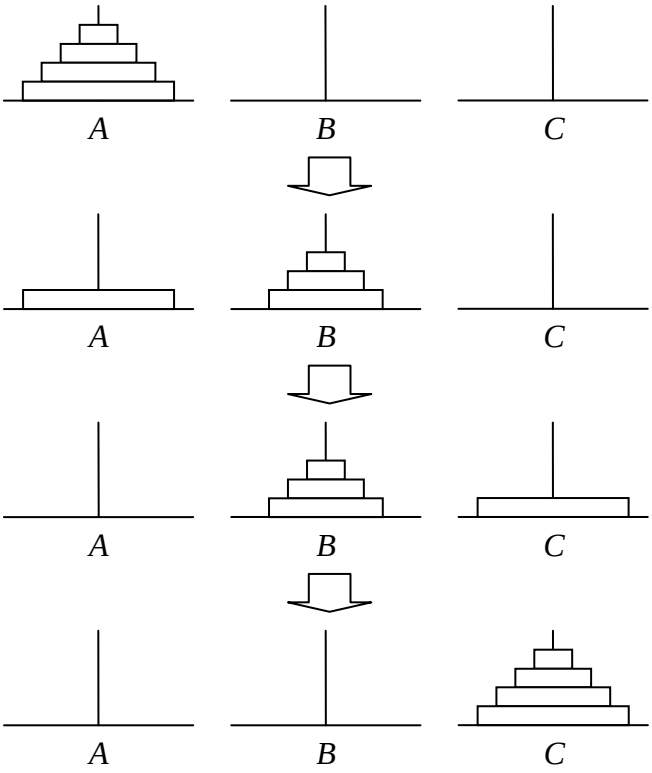


Figure 10.3

utilisant la tige restante comme intermédiaire, alors, on sait le faire pour *n* disques. Il suffit :

i. de déplacer les *n* – 1 disques supérieurs de la tige de départ à la tige intermédiaire (c'est possible car la tige intermédiaire est initialement vide) ;
ii. de déplacer le disque supérieur de la tige de départ à la tige d'arrivée (c'est possible car le disque déplacé a une taille inférieure aux tailles des disques déjà enfilés sur la tige d'arrivée) ;
iii. de déplacer les *n* – 1 disques de la tige intermédiaire à la tige d'arrivée (c'est possible car le disque supérieur de la tige d'arrivée a une taille plus grande que celles des disques à déplacer).

(voir figure 10.3).

– On sait déplacer 0 disques : aucun mouvement n'est nécessaire.

On en déduit une procédure récursive pour jouer aux Tours de Hanoi :

– Pour déplacer les *n* disques supérieurs d'une tige *x* à une tige *z* en utilisant une tige *y* comme intermédiaire, il faut déplacer les *n* – 1 disques supérieurs de la tige *x* à la tige *y* en utilisant la tige *z* comme intermédiaire, puis déplacer le disque supérieur de la tige *x* à la tige *z* et enfin déplacer les *n* – 1 disques supérieurs de la tige *y* à la tige *z*en utilisant la tige *x* comme intermédiaire.
– Pour déplacer 0 disques, aucun mouvement n'est nécessaire.

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)
```

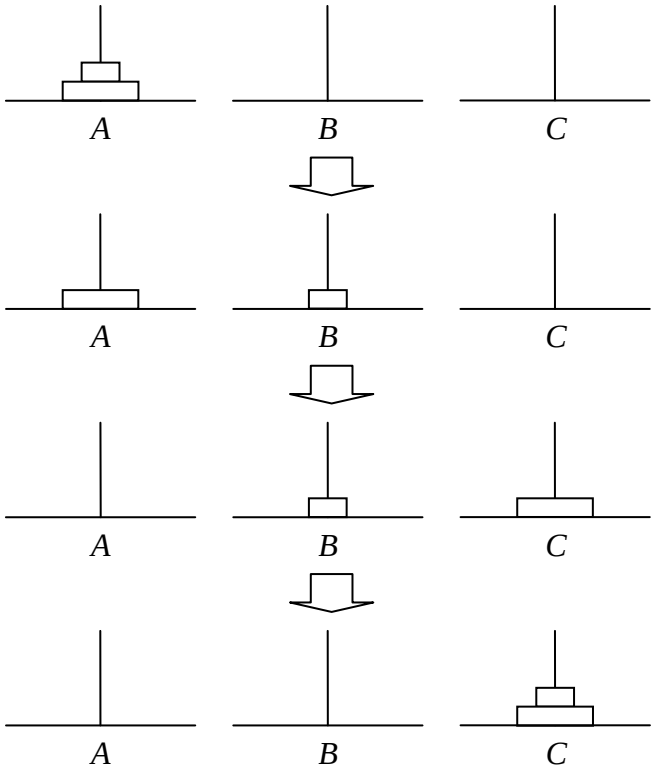


Figure 10.4

qui déplace une pile de `n` disques de la tige `x` à la tige `z` en utilisant la tige `y` comme intermédiaire. Les arguments `x`, `y` et `z` ont pour valeur 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);
       }
    }

int main(void)
    {
    int n;
    do
       {
       printf("Nombre de disques ? ");
       scanf("%d", &n);
       }
    while (n < 0);
    hanoi('A', 'B', 'C', n);
    return 0;
    }
```

Le cas de base a été étendu aux valeurs négatives de `n` (`if (n <= 0) return`) afin que le programme ne boucle pas indéfiniment si, suite à une erreur de programmation, la fonction `hanoi` est appelée avec un nombre de disques négatif.

Exécutons ce programme en jouant avec deux disques :

```
Nombre de disques ? 2
Déplace un disque de la tige A a la tige B
Déplace un disque de la tige A a la tige C
Déplace un disque de la tige B a la tige C
```

(voir figure 10.4).

Il est intéressant de calculer le nombre de mouvements nécessaires pour déplacer une pile de n disques. Pour cela, on peut s'appuyer sur l'énoncé de la procédure récursive. Soit $nbm(n)$ ce nombre de mouvements. On a :

$$nbm(n) = nbm(n-1) + 1 + nbm(n-1) = 2 \times nbm(n-1) + 1$$

En développant et étant donné que $nbm(0) = 0$ (il faut 0 mouvements pour déplacer 0 disque), on obtient :

$$nbm(n) = 2^{n-1} + 2^{n-2} + ... + 2^1 + 2^0 = 2^n - 1$$

Par exemple, il faudra 3 mouvements pour déplacer une pile de 2 disques, 255 mouvements pour déplacer une pile de 8 disques, 4 294 967 295 mouvements pour déplacer une pile de 32 disques (plus de 4 milliards).

Ce jeu a été proposé par le mathématicien français Edouard Lucas en 1863, sous la forme d'une légende racontant que la fin du monde arriverait lorsque les moines d'un temple de Hanoi ayant commencé ce jeu avec une pile de 64 disques l'auraient terminé. Le nombre de mouvements nécessaires étant égal à $2^{64} - 1$ et en supposant que les moines déplacent un disque par seconde, on peut calculer qu'il faudrait à ces moines près de 600 milliards d'années pour terminer ce jeu. La fin du monde n'est donc pas encore arrivée !

10.3.2 **Recherche dichotomique**

La recherche d'un élément dans un ensemble est une opération fréquente dans les applications informatiques. Par exemple, la recherche dans un fichier de la fiche d'une personne dont le nom est donné. De très nombreuses méthodes ont été proposées pour réaliser cette opération de façon performante. C'est l'une d'entre elles que nous étudions ci-dessous, appelée recherche dichotomique. Cette méthode s'applique à des ensembles munis d'une relation d'ordre et dont les éléments sont rangés par ordre croissant dans un tableau dont le nombre d'éléments est supérieur ou égal à la cardinalité de l'ensemble.

La recherche dichotomique est une méthode de type « diviser pour régner ». Schématiquement, elle consiste à diviser de façon répétitive le tableau en deux jusqu'à obtenir un tableau de taille unitaire qui contient ou non l'élément recherché. Plus précisément, soit *tab* le tableau contenant les éléments à rechercher et *r* l'élément recherché. L'algorithme utilisé pour réaliser une recherche dichotomique est le suivant. On coupe le tableau *tab* en son milieu générant ainsi deux sous-tableaux de tailles égales à un élément prêt. Par construction, tout élément du tableau de gauche est inférieur à tout élément du tableau de droite. Soit *e* l'élément contenu dans la dernière case du tableau de gauche. Si $r \leq e$, on recherche *r* dans le tableau de gauche, sinon ($r > e$), on le recherche dans le tableau de droite. On répète cette opération jusqu'à obtenir un tableau de taille unitaire. Si l'unique élément de ce tableau est égal à *r*, alors *r* appartient au tableau *tab*, sinon il n'appartient pas à ce tableau.

Considérons, par exemple, le tableau *tab* suivant contenant 9 nombres entiers :

tableau *tab*

1	4	6	11	17	25	42	78	93
0	1	2	3	4	5	6	7	8

et supposons que le nombre à rechercher est 25. Cette recherche se déroule de la façon suivante :

1. On coupe le tableau *tab* en deux :

1	4	6	11	17	25	42	78	93
0	1	2	3	4	5	6	7	8

   Le nombre recherché 25 est supérieur à 17, il ne peut donc appartenir qu'au sous tableau de droite.

2. On coupe le sous-tableau de droite en deux :

1	4	6	11	17	25	42	78	93
0	1	2	3	4	5	6	7	8

   Le nombre recherché 25 est inférieur ou égal à 42, il ne peut donc appartenir qu'au sous tableau de gauche.

3. On coupe le sous-tableau de gauche en deux :

1	4	6	11	17	25	42	78	93
0	1	2	3	4	5	6	7	8

4. Le nombre recherché 25 est inférieur ou égal à 25, il ne peut donc appartenir qu'au sous-tableau de gauche qui n'a qu'une seule case :

1	4	6	11	17	25	42	78	93
0	1	2	3	4	5	6	7	8

5. Cette case contient le nombre 25 qui est le nombre recherché : le nombre 25 appartient donc au tableau *tab*.

On notera que la coupure d'un sous-tableau qui commence à la case de rang *i* et se termine à la case de rang *j*, se fait après l'élément de rang (*i* + *j*) / 2 (division entière) qui est l'élément milieu du segment (*i* ; *j*). Si *i* + *j* est impair les deux sous-tableaux sont de taille égale, sinon le tableau de gauche a une case de plus que le tableau de droite.

Programmons maintenant cet algorithme en C dans le cas de tableaux d'entiers. La recherche sera réalisée par l'appel d'une fonction récursive `chercher` ayant l'en-tête :

```
chercher(int r, int tab[], int i, int j)
```

et telle que l'appel `chercher(`*r*`, `*t*`, `*i*`, `*j*`)` retourne 1 si le nombre *r* appartient au sous-tableau du tableau `tab` qui commence au rang *i* et se termine au rang *j*.

La définition de la fonction `chercher` est alors la suivante :

```
int chercher(int r, int tab[], int i, int j)
   {
   int m;
   if (i < 0 || i > j)
      {
      printf("Appel de chercher incorrect !");
      exit(1);
      }
   if (i == j)
      return tab[i] == r;
   else
      {
      m = (i + j) / 2;
      if (r <= tab[m])
         return chercher(r, tab, i, m);
      else
         return chercher(r, tab, m + 1, j);
      }
   }
```

Comme cela a été dit plus haut, la coupure du sous-tableau est faite après l'élément de rang `m = (i + j) / 2`.

Pour rechercher si un élément `r` appartient à un tableau `tab` de *n* éléments, il suffit d'évaluer l'appel :

```
chercher(r, tab, 0, n - 1)
```

Le programme suivant utilise la fonction `chercher` pour chercher si un nombre appartient au tableau qui nous a servi d'exemple.

```
#include <stdio.h>
définition de la fonction chercher
int main(void)
   {
   int mon_tableau[9] = {1, 4, 6, 11, 17, 25, 42, 78, 93}, r;
   printf("Nombre à chercher ? ");
   scanf("%d", &r);
   if (chercher(r, mon_tableau, 0, 8))
      printf("%d appartient a mon tableau", r);
   else
      printf("%d n'appartient pas a mon tableau", r);
   return 0;
   }
```

Voici deux exécutions de ce programme :

```
Nombre à chercher ? 25
25 appartient a mon tableau

Nombre à chercher ? 15
15 n'appartient pas a mon tableau
```

Lorsque l'on élabore un algorithme, il est important d'évaluer son coût en temps d'exécution et en espace mémoire. Cela permet notamment de pouvoir le comparer à d'autres algorithmes possibles pour réaliser la même tâche. Nous allons évaluer le coût en temps de la recherche dichotomique et le comparer avec le coût d'une recherche séquentielle dans le même tableau. Pour évaluer ce coût, nous ne choisirons pas comme unité de mesure le temps d'exécution lui-même qui est dépendant de l'ordinateur utilisé, mais le nombre de cases du tableau auxquelles il faut accéder pour savoir si l'élément recherché appartient ou non au tableau.

Commençons par la recherche dichotomique. Soit n la taille du tableau. A chaque étape on accède à la case placée au milieu du tableau. Le nombre d'accès nécessaire est donc égal au nombre de fois plus un qu'il faut couper le tableau en deux pour produire un tableau de taille unitaire. Supposons qu'il faille couper le tableau k fois en deux. A la $1^{ère}$ coupure on obtient un tableau de taille $n / 2$, à la 2^{e} un tableau de taille $n / 4$ et à la k^{e} un tableau de taille $n / 2^k = 1$. On a donc $k = \log_2 n$. Le coût d'une recherche dichotomique en nombre d'accès au tableau est donc égal à $1 + \log_2 n$ arrondi à l'entier immédiatement supérieur.

Calculons maintenant le coût de la recherche séquentielle. Cette recherche consiste à parcourir les cases du tableau jusqu'à en trouver une qui contienne un élément supérieur ou égal à l'élément recherché ou bien jusqu'à la fin du tableau. Le nombre d'accès nécessaire est donc égal à n si l'élément recherché n'appartient pas au tableau et à i si l'élément recherché est supérieur ou égal à l'élément contenu dans la i^e case du tableau. Le coût maximal d'une recherche séquentielle en nombre d'accès au tableau est donc égal à n et le coût moyen est égal à $n / 2$.

Le tableau ci-dessous compare les coûts maximum de ces deux méthodes pour différentes valeurs de n :

n	dichotomotique	séquentielle
10	5	10
100	8	100
1000	11	1000
10000	15	10000

Le gain procuré par la recherche dichotomique est d'autant plus important que *n* est grand. On dit que la recherche dichotomique est de complexité logarithmique et que la recherche séquentielle est de complexité linéaire.

Exercices

Exercice 10.1. La suite de Fibonacci est une suite mathématique de nombres, inventée par le mathématicien italien Leonardo Fibonacci. Elle a la forme suivante :

0, 1, 1, 2, 3, 5, 8, 13, 21...

Le nombre de rang 0 de cette suite est 0, et le nombre de rang 1 est 1. A partir du rang 2, chaque nombre de cette suite est la somme des deux précédents.

1. Définir une fonction récursive `fibonacci` qui appliquée à un nombre entier i ($i \geq 0$), retourne le i^e nombre de la suite de Fibonacci.
2. Ecrire un programme qui saisit un nombre i ($i \geq 0$), puis affiche le i^e nombre de la suite de Fibonacci, calculé par appel de la fonction `fibonacci`.

Exercice 10.2. Le but de l'exercice est d'écrire un programme qui compte à rebours de n ($n \geq 0$) à 1, en utilisant une fonction récursive. Par exemple, si $n = 5$, ce programme devra afficher : 5 4 3 2 1

1. Définir une fonction récursive `compter_a_rebours` qui appliquée à un nombre entier n ($n \geq 1$) affiche les nombres entiers de n à 1 à raison de un par ligne.
2. Ecrire un programme qui lit un nombre ≥ 1 saisi par l'utilisateur et compte à rebours à partir de ce nombre en appelant la fonction `compter_a_rebours`.
3. Montrer qu'en déplaçant une seule instruction dans la fonction `compter_a_rebours`, on peut faire compter le programme de 1 à *n*.

Exercice 10.3. Le plus grand commun diviseur (PGCD) de deux entiers relatifs est le plus grand entier naturel qui divise simultanément ces deux entiers. Il peut être calculé par l'algorithme d'Euclide qui peut s'énoncer ainsi : « Soit a et b deux entiers positifs ou nuls. Si $0 < a \leq b$ et si r est le reste de la division entière de b par a, alors le PGCD de a et b vaut le PGCD de r et a. Le PGCD de 0 et b vaut b. ».

1. Tester cet algorithme en calculant le PGCD de 72 et 132 (réponse : 12).
2. Définir une fonction récursive d'en-tête `int pgcd(int a, int b)` qui calcule le PGCD des entiers `a` et `b` en appliquant l'algorithme d'Euclide. On supposera que la valeur de `a` est inférieure ou égale à celle de `b`.
3. Définir une fonction `main` qui demande à l'utilisateur la saisie de deux entiers relatifs, calcule leur PGCD en faisant appel à la fonction `pgcd` et affiche ce PGCD.

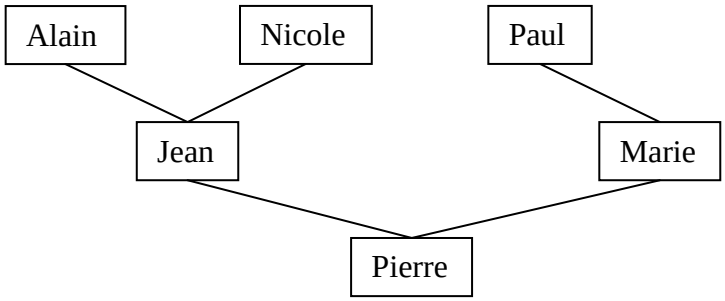


Figure 10.5. Un arbre généalogique

Exercice 10.4. Il s'agit d'écrire un programme qui traduit un nombre romain en un nombre décimal.

Les chiffres romains et leurs valeurs en décimal sont les suivants :

I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000

Voici quelques exemples de nombres romains et leurs valeurs en décimal :

XIV = 14, XV = 15, XLVII = 47, XCVII = 97, CXLIX = 149.

Pour convertir un nombre romain *r* en un nombre décimal, on utilisera l'algorithme récursif suivant :

si *r* n'a qu'un seul chiffre *c* alors
  valeur décimale de *r* = valeur décimale de *c*
si la valeur du 1er chiffre *c* de *r* est plus grande ou égale à la valeur de son 2e chiffre alors :
  valeur décimale de *r* = valeur décimale de *c* + valeur décimale du reste de *r*
si la valeur du 1er chiffre *c* de *r* est plus petite que la valeur de son 2e chiffre alors
  valeur décimale de *r* = valeur décimale du reste de *r* − valeur décimale de *c*

1. Tester l'algorithme sur les nombres donnés en exemple ci-dessus.
2. Définir une fonction d'en-tête `int valeur_chiffre_romain(char c)` qui retourne la valeur décimale du chiffre romain `c`.
3. Définir une fonction d'en-tête `int valeur_nb_romain(char *r, int i)` qui retourne la valeur décimale du nombre romain commençant à la position `i` dans `r`, en appliquant l'algorithme de conversion décrit ci-dessus.
4. Définir une fonction `main` qui demande à l'utilisateur la saisie d'un nombre romain, le convertit en un nombre décimal en faisant appel à la fonction `valeur_nb_romain` et affiche ce nombre.

Exercice 10.5.

Il s'agit d'écrire un programme permettant de créer et d'interroger l'arbre généalogique d'une personne. Par exemple, l'arbre généalogique de la figure 10.5 est celui de la personne nommée Pierre : Jean est le père de Pierre et Marie sa mère. Alain est le père de Jean et Nicole sa mère. Paul est le père de Marie.

Chaque nœud de cet arbre est associé à une et une seule personne. Pierre est la racine de cet arbre. Il a deux sous-arbres généalogiques : celui de son père (Jean) et celui de sa mère

(Marie). Marie n'a qu'un seul sous-arbre généalogique : celui de son père (Paul), celui de sa mère est vide. Les sous-arbres généalogiques d'Alain, Nicole et Paul sont vides.

Une personne sera décrite par une structure à trois champs, instance du type `Personne` :

- `nom`, une chaîne de 20 caractères ;
- `pere`, un pointeur sur la structure décrivant le père de cette personne ;
- `mere`, un pointeur sur la structure décrivant la mère de cette personne.

Une personne sera identifiée par l'adresse (de type `Personne *`) de la structure qui la décrit.

Si le père d'une personne n'est pas spécifié (soit parce qu'il est inconnu, soit parce qu'il n'apparaît pas dans l'arbre), le champ `pere` aura pour valeur `NULL`. De même, si la mère d'une personne n'est pas spécifiée, le champ `mere` aura pour valeur `NULL`.

La variable `racine` est un pointeur sur la structure décrivant la personne racine de l'arbre généalogique.

1. Définir le type `Personne`.
2. Définir une fonction d'en-tête `Personne *creer_personne(char *n)` qui crée une personne de nom `n`, de père et de mère non spécifiés et qui retourne l'identifiant de cette personne.
3. Définir une fonction d'en-tête `void affecter_pere(Personne *p, Personne *pp)` qui affecte au champ `pere` de la personne `p` l'identifiant `pp` du père de `p`.
4. Définir une fonction d'en-tête `void affecter_mere(Personne *p, Personne *mp)` qui affecte au champ `mere` de la personne `p` l'identifiant `mp` de la mère de `p`.
5. Définir une fonction récursive d'en-tête `Personne *chercher (Personne *p, char *n)` qui recherche une personne de nom `n` dans l'arbre généalogique de la personne `p` et qui retourne :
   - l'identifiant de cette personne, si elle existe ;
   - `NULL`, si elle n'existe pas.

   (**Aide** : Une personne appartient à l'arbre généalogique de *p* si elle est *p* ou si elle appartient au sous-arbre généalogique du père de *p* ou si elle appartient au sous-arbre généalogique de la mère de *p*. Une personne n'appartient pas à un arbre généalogique vide.)
6. Définir une fonction récursive d'en-tête `void parcourir(Personne *p)` qui parcoure l'arbre généalogique de la personne `p` et affiche le nom de chaque personne de cet arbre.

   (**Aide** : Parcourir un arbre généalogique vide, c'est ne rien faire. Parcourir l'arbre généalogique de *p* c'est traiter *p* (dans le cas présent : afficher son nom) puis parcourir l'arbre généalogique du père de *p*, puis parcourir l'arbre généalogique de la mère de *p*).
7. Définir une fonction `void nom_des_ancetres_de(char *n)` qui affiche les noms des ancêtres de la personne de nom `n` en faisant appel aux fonctions `chercher` et `parcourir`.
8. Définir une fonction `main` qui :
   - construit un arbre généalogique en faisant appel aux fonctions `creer_personne`, `affecter_pere` et `affecter_mere` (cet arbre devra contenir une dizaine de personnes dont au moins une dont le père est connu, mais pas la mère ou inversement) ;
   - demande à l'utilisateur le nom d'une personne de l'arbre généalogique et affiche les noms des ancêtres de cette personne en faisant appel à la fonction `nom_des_ancetres_de`.

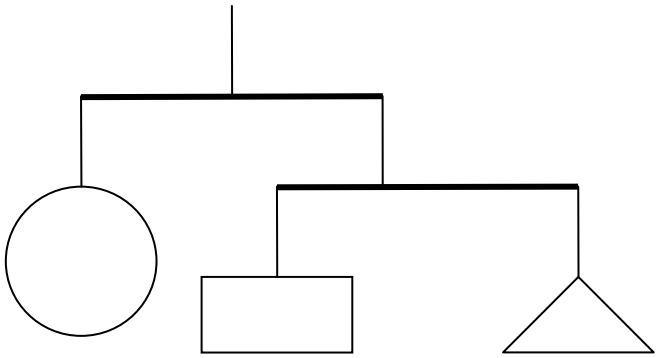


Figure 10.6. *Un mobile décoratif*

Exercice 10.6.

Cet exercice est la poursuite de l'exemple sur l'équilibrage d'un mobile décoratif étudié au paragraphe 7.5.4 du chapitre 7. Il s'agit d'équilibrer un mobile composé d'un nombre quelconque d'éléments, que l'on peut définir récursivement de la façon suivante : « un mobile est soit un cercle, soit un rectangle, soit un triangle, soit une balance formée d'un fléau ayant un mobile accroché à chacune de ses extrémités. ». Par exemple, le mobile de la figure 10.6 est formé d'une balance dont le fléau a un cercle (et donc un mobile) accroché à son extrémité gauche et une balance (et donc un mobile) accrochée à son extrémité droite, balance qui a un rectangle accroché à son extrémité gauche et un triangle accroché à son extrémité droite.

Il s'agit de créer un mobile et de calculer les points d'attache de chacun de ses fléaux de façon à ce qu'ils soient horizontaux. Le poids d'une figure et la position du point d'attache d'un fléau seront calculés de la même façon que dans l'exemple du paragraphe 7.5.4 du chapitre 7.

Un mobile sera représenté par une structure instance du type `Mobile`, ayant un champ `type` de type `char` qui indique le type du mobile (`'c'` pour un cercle, `'r'` pour un rectangle, `'t'` pour un triangle et `'b'` pour une balance) et un champ `u` ayant pour valeur une union des structures décrivant chaque type de mobile. Les structures décrivant un cercle, un rectangle et un triangle seront définies de la même façon que dans l'exemple du paragraphe 7.6.4. La structure décrivant une balance sera composée de trois champs :

- `mobile1` qui pointe sur la variable contenant la représentation du mobile suspendu à l'extrémité gauche ;
- `mobile2` qui pointe sur la variable contenant la représentation du mobile suspendu à l'extrémité droite ;
- `attache` ayant pour valeur la distance entre l'extrémité gauche du fléau et son point d'attache.

Un mobile sera identifié par l'adresse de la structure qui le décrit.

1. Définir le type `Mobile`.
2. Définir une fonction d'en-tête `Mobile *creer_cercle(float r)` qui crée un mobile qui est un cercle de rayon `r` et qui retourne l'identifiant de ce mobile.
3. Définir une fonction d'en-tête `Mobile *creer_rectangle(float l1, float l2)` qui crée un mobile réduit à un rectangle de longueur `l1` et de largeur `l2` et qui retourne l'identifiant de ce mobile.

4. Définir une fonction d'en-tête `Mobile *creer_triangle(float b, float h)` qui crée un mobile qui est un triangle de base `b` et de hauteur `h` et qui retourne l'identifiant de ce mobile.
5. Définir une fonction d'en-tête `Mobile *creer_balance(Mobile *mg, Mobile *md)` qui crée un mobile composé d'un fléau ayant un mobile `mg` suspendu à son extrémité gauche et un mobile `md` suspendu à son extrémité droite et qui retourne l'identifiant de ce mobile.
6. Définir une fonction d'en-tête `float attacher(Mobile *m)` qui :
   - si `m` est une figure, retourne le poids (son aire, en réalité) de cette figure ;
   - si `m` est une balance, affecte au champ `attache` de `m` la position du point d'attache du fléau de cette balance et retourne la somme des poids des figures suspendues à ce mobile.
7. Définir une fonction d'en-tête `void afficher(MOBILE *m)` qui :
   - si `m` est une figure, affiche « cercle », « rectangle » ou « triangle » selon le type de cette figure ;
   - si `m` est une balance, affiche : `(`*affichage de* m_1`,` *position du point d'attache*`,` *affichage de* m_2`)`.

   Par exemple, l'affichage du mobile de la figure ci-dessus sera le suivant :

   ```
   (cercle, 0.49, (rectangle, 0.33, triangle))
   ```
8. Définir une fonction `main` qui :
   - crée un mobile en faisant appel aux fonctions `creer_cercle`, `creer_rectangle`, `creer_triangle` et `creer_balance` ;
   - calcule les points d'attache en faisant appel à la fonction `attacher` ;
   - affiche ce mobile.

Par exemple, le mobile de la figure de la figure 10.6 pourra être créé par l'appel :

```
creer_balance(creer_cercle,
              creer_balance(creer_rectangle(2, 1),
                            creer_triangle(2, 1)));
```

On testera le programme sur plusieurs configurations de mobiles.

11
Un compilateur J pour la pitchoun-machine

Au premier chapitre, nous avons expliqué comment se déroulait la compilation d'un programme au travers d'un langage et d'un processeur jouets : le langage J et la pitchoun-machine, inventés pour la circonstance. Pour conclure ce cours d'introduction à la programmation impérative en C, nous allons entreprendre un exercice très instructif : l'écriture en C d'un compilateur J qui génère du code exécutable par la pitchoun-machine et l'exécute sur une pitchoun-machine virtuelle. Ce n'est pas un exercice facile, mais arrivé à la fin de ce cours, le lecteur a acquis une pratique du C qui lui permet de l'entreprendre.

Après une introduction (paragraphe 11.1) qui apportera les connaissances nécessaires à la compréhension d'un compilateur, nous écrirons pas à pas le programme qui simule la pitchoun-machine (paragraphe 11.2), compile un programme J (paragraphe 11.3) puis lance son exécution sur la pitchoun-machine virtuelle (paragraphe 11.4) et nous l'expérimenterons (paragraphe 11.5).

Avant d'aborder l'étude de ce chapitre, il est nécessaire de relire attentivement le paragraphe 1.2 du chapitre 1 consacré au langage J et à la pitchoun-machine.[1]

11.1 Introduction à la compilation

Un programme est écrit dans un langage particulier dit : « langage de programmation ». Un langage de programmation est un langage formel. Les langages formels se distinguent des langages naturels (anglais, arabe, chinois, français...) qui sont ceux des humains en ce sens qu'ils obéissent à un ensemble de règles strictes qui permettent de construire des énoncés précis et sans ambigüité.

Un langage formel a une syntaxe et une sémantique. Syntaxiquement, un langage formel est un ensemble de chaînes de caractères de longueur finie dont la chaîne vide. Les caractères sont choisis dans un alphabet fini. Ces chaînes de caractères peuvent être en nombre infini car leurs longueurs ne sont pas bornées. La syntaxe d'un langage formel est définie par l'ensemble des règles applicables pour construire les chaînes de caractères de ce langage.

La sémantique d'un langage formel donne un sens aux chaînes de caractères qui appartiennent à ce langage. Par exemple, si la chaîne de caractères « `12 + 37` » appartient à un langage formel qui décrit des expressions arithmétiques, la sémantique de ce langage pourra spécifier : que la chaîne de caractères « `12` » représente le nombre 12 (qui est une abstraction), que la chaîne de caractères « `37` » représente le nombre 37, que la chaîne de caractères « `12 + 37` » représente l'addition du nombre 12 et du nombre 37 dont la valeur est le nombre 49 représentable par la chaîne de caractères « `49` ».

Un langage de programmation est un langage formel dont les chaînes de caractères sont les programmes que l'on peut écrire en ce langage. La sémantique d'un langage de

[1] Pour une étude plus approfondie tout en restant très pratique, de la construction d'un compilateur, on pourra consulter l'excellent livre de Niklaus Wirth, le père du langage Pascal : *Compiler Construction* (Addison-Wesley).

programmation décrit le comportement d'un programme, ce qu'il exécute. Elle peut être définie par un ensemble de règles comme nous l'avons fait pour le langage J (voir chapitre 1, paragraphe 1.2.1).

Compiler un programme, c'est traduire le texte de ce programme : le code source, en un code exécutable par la machine sur laquelle il doit être exécuté. Cette traduction est réalisée par un autre programme : le compilateur.

La compilation d'un programme comporte trois grandes phases :

- l'**analyse lexicale** qui consiste à regrouper les caractères du texte du programme en unités lexicales ou lexèmes. Dans un langage de programmation, ces lexèmes sont les mots réservés, les constantes littérales, les noms de variables, les symboles d'opérateurs, etc.
- l'**analyse syntaxique** qui consiste à reconnaitre les constituants du texte du programme tels qu'ils sont définis par la grammaire. Dans un langage de programmation, ces constituants sont les déclarations, les expressions, les instructions, etc.
- la **génération du code** qui consiste, entre autres, à allouer une place en mémoire à chaque variable du programme et à traduire les expressions ou les instructions du programme en une suite d'instructions exécutables par la machine cible.

11.1.1 **Analyse lexicale**

L'analyse lexicale consiste à transformer le texte du programme en une chaîne de lexèmes. Par la suite, nous représenterons un lexème par une suite de caractères placée entre guillemets simples et une chaîne de lexèmes par une suite de lexèmes placée entre guillemets doubles.

Dans le langage J, les lexèmes sont :

- les mots réservés : '`afficher`', '`saisir`' et '`var`' ;
- les noms de variables qui sont constitués d'une lettre suivie d'une suite quelconque de lettres ou de chiffres ;
- les constantes littérales entières qui sont constituées d'une suite de chiffres (au moins un) ;
- les symboles spéciaux sont : '`+`', '`-`', '`*`' , '`/`', '`%`', '`(`', '`)`', '`:=`', '`,`', '`;`' et '`.`'.

Par exemple, après analyse lexicale, l'expression `2 * (longueur + largeur)` du langage J sera transformée en la chaîne de lexèmes « '`2`' '`*`' '`(`' '`longueur`' '`+`' '`largeur`' '`)`' ».

Les caractères dits d'espacement : espace, tabulation ou nouvelle ligne, n'appartiennent pas aux unités lexicales mais ils doivent être pris en compte pour séparer ces unités.

Les unités lexicales d'un langage de programmation sont généralement décrites sous la forme d'une expression régulière (ou expression rationnelle). Une expression régulière décrit la façon dont une chaîne de caractères est composée à partir d'un ensemble de caractères : l'alphabet et de trois opérateurs : la concaténation, l'union et la fermeture de Kleene. Elle peut avoir l'une des formes suivantes :

- ε qui décrit la chaîne de caractères vide ;
- '*c*' qui décrit la chaîne de caractères composée de l'unique caractère *c* ;
- R_1R_2 (concaténation) qui décrit une chaîne de caractères constituée par la concaténation d'une chaîne de caractères décrite par l'expression régulière R_1 et d'une chaîne de caractères décrite par l'expression régulière R_2 ;
- $R_1 \mid R_2$ (union) qui décrit une chaîne de caractères qui est décrite soit par l'expression régulière R_1, soit par l'expression régulière R_2 ;

– *R** (fermeture de Kleene) qui décrit une chaîne de caractères constituée par la concaténation d'un nombre quelconque (≥ 0) de chaînes de caractères décrites par l'expression régulière *R* ;
– $(R) \equiv R$.

On peut nommer une expression régulière et utiliser ce nom à la place de cette expression dans une autre expression régulière.

Par exemple, les unités lexicales d'une expression du langage J sont décrites par l'expression régulière suivante :

constante | *nom* | '+' | '-' | '*' | '/' | '%' | '(' | ')'

où :

chiffre ≡ '0' | '1' | ... | '9'
lettre ≡ 'a' | 'b' | ... | 'z'
constante ≡ *chiffre chiffre*+
nom ≡ *lettre* (*lettre* | *chiffre*)*

11.1.2 **Analyse syntaxique**

L'analyse syntaxique consiste à vérifier si la chaîne de lexèmes produite par l'analyse lexicale est conforme à la grammaire du langage et à produire une représentation de cette chaîne sous la forme d'un arbre syntaxique qui met en évidence les constituants du programme. Dans le langage J les constituants d'un programme sont : le programme lui-même, la définition des variables, les instructions et les expressions.

Grammaires

Pour décrire la syntaxe d'un langage de programmation, on utilise une catégorie particulière de grammaires : les grammaires BNF (voir chapitre 1, note 1). Une grammaire BNF est constituée :

– d'un ensemble de règles de la forme $x \rightarrow y_1 \ldots y_n$ où x est un symbole non terminal et les y_i sont des symboles terminaux ou non terminaux ;
– d'un symbole de départ : un symbole non terminal.

Un symbole non terminal désigne un constituant générique du texte d'un programme (une expression, une instruction...). Un symbole terminal désigne un lexème. Nous adopterons la convention suivante pour désigner ces symboles :

– un symbole non terminal est représenté par un nom en italiques (par exemple : *expression, instruction* dans la grammaire du langage J) ;
– un symbole terminal est représenté par la suite de ses caractères placée entre apostrophes simples (par exemple : 'saisir' dans la grammaire du langage J) ou bien par un nom en italiques souligné désignant la catégorie à laquelle appartient ce symbole (par exemple : *nom* dans la grammaire du langage J pour désigner un nom de variable) ;
– le symbole de départ désigne le constituant le plus générique (*programme* dans la grammaire du langage J).

Vérifier si une chaîne de lexèmes est conforme à une grammaire, c'est chercher si cette chaîne peut être dérivée à partir du symbole de départ de la grammaire par une suite de réécritures.

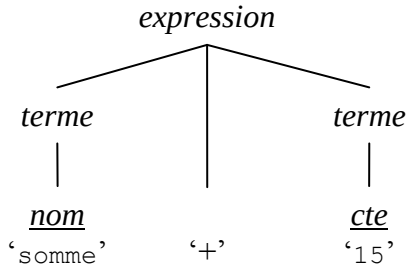


Figure 11.1. *Arbre syntaxique construit par la dérivation de l'exemple 11.2*

La réécriture d'un symbole *x* selon une règle $x \rightarrow \alpha$ de la grammaire dans une chaîne de symboles consiste à remplacer *x* par α dans cette chaîne. Par exemple, la réécriture du symbole *expression* selon la règle « *expression → terme opérateur terme* » dans la chaîne de symboles « ‘(’ *expression* ‘)’ » produit la nouvelle chaîne de symboles « ‘(’ *terme opérateur terme* ‘)’ ».

La dérivation d'une chaîne de lexèmes à partir d'un symbole d'une grammaire est une suite de réécritures qui produisent cette chaîne.

Exemple 11.1. Soit la grammaire suivante :

(1) *expression → terme opérateur terme*
(2) *terme →* ***nom***
(3) *terme →* ***cte***

où le symbole terminal ***nom*** désigne un nom de variable et le symbole terminal ***cte*** désigne une constante littérale entière (ce qui sera le cas dans tous les exemples de ce chapitre).

Une dérivation de la chaîne de lexèmes « ‘somme’ ‘+’ ‘15’ » à partir du symbole *expression* est la suivante :

expression	
terme ‘+’ *terme*	par réécriture du symbole *expression* selon la règle 1
nom ‘+’ *terme*	par réécriture du symbole *terme* de gauche selon la règle 2
nom ‘+’ ***constante***	par réécriture du symbole *terme* selon la règle 3

□

Une dérivation construit l'arbre syntaxique de la chaîne dérivée. La racine de cet arbre est le symbole à partir duquel la dérivation a débuté, chaque nœud interne est un symbole qui a été réécrit et a pour fils, placés dans le même ordre, les symboles par lesquels il a été remplacé et chaque feuille est un symbole terminal. Par exemple, la dérivation de l'exemple 11.1 construit l'arbre syntaxique de la figure 11.1.

Plusieurs dérivations de la même chaîne de lexèmes peuvent exister. Une dérivation est dite « dérivation à gauche » si c'est toujours le symbole non terminal le plus à gauche qui est réécrit et « dérivation à droite » si c'est toujours le symbole non terminal le plus à droite qui est réécrit. La dérivation de l'exemple 11.1 est une dérivation à gauche.

Quelque soit l'ordre dans lequel sont effectuées les réécritures, la dérivation d'une chaîne de lexèmes doit produire le même arbre syntaxique. Si ce n'est pas le cas, on dit que la grammaire est ambiguë. Une grammaire ambigüe n'est pas acceptable car c'est à partir de son arbre syntaxique qu'est généré le code d'un programme. On comprendra facilement que ce

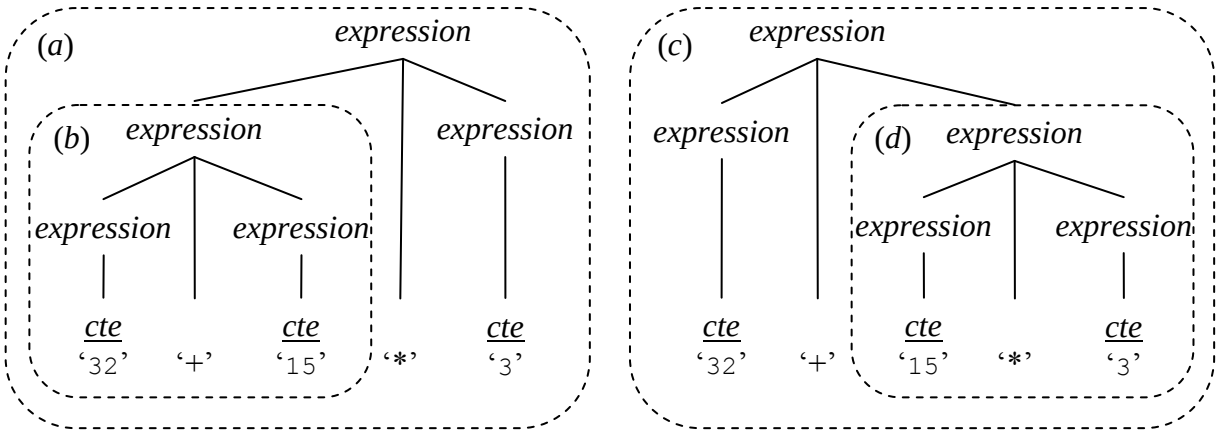


Figure 11.2 *Deux arbres syntaxiques pour la même chaîne de lexèmes*

code, et donc ce qu'exécute le programme, ne doit pas dépendre de l'ordre dans lequel est effectuée l'analyse syntaxique.

Exemple 11.2. La grammaire suivante :

(1) *expression* → *expression opérateur expression*
(2) *expression* → *cte*
(3) *opérateur* → '+'
(4) *opérateur* → '*'

est ambigüe. En effet, considérons la chaîne de lexèmes : « '32' '+' '15' '*' '3' ». Deux arbres syntaxiques peuvent être produits : l'arbre *a* de la figure 11.2 si la dérivation de cette chaîne est une dérivation gauche et l'arbre *c* de cette même figure si la dérivation de cette chaîne est une dérivation droite. Le code généré à partir de l'arbre *a* sera : ajouter 32 à 15 (sous-arbre *b*) puis multiplier le résultat par 3 et le code généré à partir de l'arbre *c* sera : multiplier 15 par 3 (sous-arbre *d*) puis ajouter 32 au résultat. L'exécution du premier code produira la valeur 141 et celle du second produira la valeur 77. □

Grammaires LL(1)

Il existe deux grandes familles d'analyseurs syntaxiques pour les langages de programmation :

- les analyseurs descendants qui construisent l'arbre syntaxique de la chaîne de lexèmes à analyser à partir du symbole de départ de la grammaire et en « descendant » les règles de grammaire de leur partie gauche vers leur partie droite ;
- les analyseurs ascendants qui construisent l'arbre syntaxique de la chaîne de lexèmes à analyser à partir des lexèmes de cette chaîne et en « remontant » les règles de grammaire de leur partie droite vers leur partie gauche.

Parmi ceux-ci, les analyseurs descendants de type *LL*(1) sont les plus simples à mettre en œuvre. Un analyseur *LL*(1) est caractérisé par les propriétés suivantes :

- il effectue l'analyse en lisant les lexèmes de la chaîne à analyser de gauche à droite sans retour en arrière ;
- il construit une dérivation à gauche ;
- il lit un lexème en avance en avance : le prochain lexème ;
- il détermine la règle qui doit être appliquée au vu du prochain lexème.

D'où le nom de ces analyseurs : le premier *L* signifie « *L*eft to right parsing », le second *L* signifie « *L*eftmost derivation » et le (1) signifie qu'un lexème est lu en avance.

Pour bien comprendre le principe d'un analyseur *LL*(1), considérons la grammaire très simple suivante :

(1) *expression* → *cte opérateur cte*
(2) *opérateur* → '+'
(3) *opérateur* → '*'

et la chaîne de lexèmes à analyser : « '3' '+' '4' ». Pour effectuer cette analyse, un analyseur *LL*(1) lit le lexème '3' (c'est le prochain lexème), puis applique la règle 1. D'après cette règle, il faut tout d'abord reconnaître une constante, c'est le cas : le lexème '3' est une constante. Il est donc « consommé » puis le lexème '+' est lu (c'est le prochain lexème). Il faut ensuite reconnaître un opérateur. Or il y a deux règles ayant le symbole *opérateur* pour partie gauche. Le fait que le prochain lexème soit '+' indique à l'analyseur que c'est la règle 2 qui doit être appliquée. Il l'applique. Le lexème '+' est alors consommé et le lexème '4' est lu (c'est le prochain lexème). Il faut enfin reconnaître une constante, c'est le cas : le lexème '4' est une constante.

Une grammaire analysable par un analyseur *LL*(1) est dite grammaire *LL*(1). Pour être *LL*(1) une grammaire :

- ne doit pas être récursive à gauche, c.-à-d. ne doit pas comporter de règles dont la partie droite commence directement ou indirectement par le symbole de la partie gauche, afin que l'analyse syntaxique ne boucle pas indéfiniment ;
- doit être factorisée à gauche, c.-à-d. ne doit pas comporter plusieurs règles ayant la même partie gauche et dont la partie droite commence directement ou indirectement par la même suite de symboles, afin de ne pas avoir à revenir en arrière dans la chaîne à analyser.

Une grammaire qui n'est pas *LL*(1) peut être transformée en une grammaire *LL*(1) qui génère les mêmes chaînes de lexèmes et pour les grammaires simples, cette transformation peut-être effectuée manuellement comme le montre les exemples 11.3 et 11.4. □

Exemple 11.3. La grammaire *G* suivante :

(1) *expression* → *expression* '+' *terme*
(2) *expression* → *terme*
(3) *terme* → *nom*
(4) *terme* → *cte*

n'est pas *LL*(1) car la règle 1 est récursive à gauche. On comprend aisément que l'analyse descendante d'une expression bouclera sur l'application de cette règle.

Notons que selon *G*, la chaîne « '3' '+' '4' » est une expression puisque qu'elle est formée (règle 1) de la chaîne « '3' » qui est un terme (règle 4) et donc une expression (règle 2) suivie de la chaîne « '+' » suivie de la chaîne « '4' » qui est un terme (règle 4).

On peut supprimer la récursion à gauche en transformant cette grammaire en la grammaire *G*' suivante :

(1) *expression* → *terme suite-expression*
(2) *suite-expression* → ‘+’ *terme suite-expression*
(3) *suite-expression* → *ε*
(4) *terme* → *nom*
(5) *terme* → *cte*

qui est *LL*(1) et qui génère les mêmes expressions que *G*. En effet, selon *G*’ la chaîne « ‘3’ ‘+’ ‘4’ » est aussi une expression puisque formée (règle 1) de la chaîne « ‘3’ » qui est un terme (règle 5) suivi de la chaîne « ‘+’ ‘4’ » qui est une suite d’expressions puisque formée (règle 2) de la chaîne « ‘+’ » suivie de la chaîne « ‘4’ » qui est un terme (règle 5) suivie de la chaîne vide qui est une suite d’expression (règle 3). □

Exemple 11.4. La grammaire *G* suivante :

(1) *liste-instructions* → *instruction* ‘;’ *liste-instructions*
(2) *liste-instructions* → *instruction*

n’est pas *LL*(1) car les règles 1 et 2 ont une partie droite qui commence par le même symbole *instruction*. Si on cherche à analyser une liste d’instructions en appliquant la règle 1 et qu’après l’analyse de l’instruction on ne rencontre pas le lexème ‘;’, il faudra revenir en arrière dans la chaîne à analyser jusqu’au lexème qui précède cette instruction et recommencer l’analyse à partie de ce lexème.

On peut factoriser à gauche le terme *instruction* en transformant cette grammaire en la grammaire *G*’ suivante :

(1) *liste-instructions* → *instruction suite-liste-instructions*
(2) *suite-liste-instructions* → ‘;’ *instruction suite-liste-instructions*
(3) *suite-liste-instructions* → *ε*

qui est *LL*(1) et qui génère les mêmes listes d’instructions que *G*.

Lors de l’analyse syntaxique, c’est le fait que le prochain lexème est ou n’est pas un ‘;’ qui déterminera si la règle à appliquer est la règle 2 ou la règle 3. Dans le deuxième aucun lexème ne sera consommé. □

Prise en compte de la priorité et de l’associativité des opérateurs

Afin d’éviter de surcharger les expressions par un trop grand nombre de parenthèses, il est traditionnel, en langage mathématique ou en langage de programmation, d’attribuer à chaque opérateur un degré de priorité et un sens d’associativité qui permettent d’omettre certains couples de parenthèses en fixant l’ordre dans lequel doivent être effectuées les opérations. Par exemple, si les opérateurs arithmétiques sont associatifs de gauche à droite et que les opérateurs multiplicatifs sont prioritaires sur les opérateurs additifs, l’expression `3 + 4 * 5` sera équivalente à `3 + (4 * 5)` et l’expression `5 - 3 - 2` sera équivalente à `(5 - 3) - 2`.

Remarquons que le sens de l’associativité est important dans le cas d’opérateurs de même priorité non associatifs mathématiquement. En effet, que l’opérateur d’addition soit déclaré associatif de gauche à droite ou associatif de droite à gauche, l’expression `5 + 3 + 2` aura la valeur 10, car l’addition est associative. Par contre l’expression `5 - 3 - 2` aura la valeur 0 si l’opérateur de soustraction est déclaré associatif de gauche à droite et la valeur 4 s’il est déclaré associatif de droite à gauche, car la soustraction n’est pas associative.

La priorité et l’associativité des opérateurs du langage C ont été présentées au chapitre 3. Celles du langage J sont identiques pour les opérateurs qu’ils ont en commun. La priorité et l’associativité des opérateurs doivent être connues de l’analyseur syntaxique, afin de générer

le bon arbre syntaxique et d'éviter les ambigüités (voir ci-dessus exemple 11.4). Elles peuvent être énoncées à part ou bien être intégrées dans la grammaire elle-même en décomposant une expression en autant de sous-expressions qu'il y a de niveaux de priorité, comme le montre l'exemple 11.5.

Exemple 11.5. La grammaire *LL*(1) suivante :

expression → *facteur suite-expression*
suite-expression → *opérateur-additif facteur suite-expression*
suite-expression → *ε*
facteur → *cte suite-facteur*
suite-facteur → *opérateur-multiplicatif cte suite-facteur*
suite-facteur → *ε*
opérateur-additif → '+'
opérateur-additif → '-'
opérateur-multiplicatif → '*'
opérateur-multiplicatif → '/'

décrit des expressions arithmétiques formées à l'aide de constantes entières et des quatre opérateurs classiques en intégrant le fait que les opérateurs multiplicatifs sont prioritaires sur les opérateurs additifs et que ces opérateurs sont associatifs de gauche à droite.

L'analyse syntaxique selon cette grammaire, de la chaîne « '3' '+' '4' '*' '5' » identifiera le facteur « '3' » et le facteur « '4' '*' '5' » permettant la génération d'un code qui effectuera l'évaluation de ces deux facteurs avant d'évaluer leur somme conformément à la priorité supérieure de l'opérateur de multiplication sur l'opérateur d'addition.

L'analyse syntaxique selon cette grammaire de la chaîne « '5' '-' '3' '-' '2' » identifiera le premier facteur de la première soustraction ('5') puis le second facteur de chacune des deux soustractions ('3' et '2'), permettant la génération du code : « soustraire 3 à 5 puis soustraire 2 au nombre obtenu » conformément à l'associativité de gauche à droite de l'opérateur de soustraction. □

Analyse *LL*(1)

Un analyseur *LL*(1) opère sur :

- la chaîne à analyser : une chaîne de lexèmes, dont la fin est marquée par un lexème spécial que nous désignerons par FF ;
- une chaîne de symboles dite « chaîne dérivée » ;
- le prochain lexème.

Au début de l'analyse, la chaîne dérivée est constituée du symbole de départ de la grammaire et le prochain lexème (le premier) de la chaîne à analyser est lu. L'analyse consiste en une suite d'actions de deux types : réécriture et consommation du prochain lexème :

- une réécriture est effectuée si le symbole de gauche *x* de la chaîne dérivée est un symbole non terminal. Elle consiste à remplacer ce symbole par la suite de symboles *α* si la grammaire contient un ensemble de règles ayant *x* pour partie gauche et que parmi celles-ci celle qui doit être appliquée au vu du prochain lexème est $x \rightarrow \alpha$.
- la consommation du prochain lexème est effectuée si le symbole de gauche *t* de la chaîne dérivée est un symbole terminal et si le prochain lexème est conforme à *t*. Elle consiste à lire le prochain lexème dans la chaîne à analyser et à effacer le symbole *t* de la chaîne dérivée.

Etat	Chaîne à analyser	Chaîne dérivée	Action
0	‘somme’ ‘+’ ‘45’ FF	*expression*	réécriture selon la règle 1
1	‘somme’ ‘+’ ‘45’ FF	*terme suite-expression*	réécriture selon la règle 4
2	‘somme’ ‘+’ ‘45’ FF	*nom suite-expression*	consommation
3	‘+’ ‘45’ FF	*suite-expression*	réécriture selon la règle 2
4	‘+’ ‘45’ FF	‘+’ *terme suite_expression*	consommation
5	‘45’ FF	*terme suite_expression*	réécriture selon la règle 4
6	‘45’ FF	*cte suite-expression*	consommation
7	FF	*suite-expression*	réécriture selon la règle 3
8	FF		**analyse réussie !**

Figure 11.3. *Analyse LL(1)*

L’analyse se termine si aucune des deux actions (dérivation ou consommation) ne peut être effectuée. Elle a réussi si le lexème FF a été lu et si la chaîne dérivée est vide. Dans le cas contraire, elle a échoué.

Exemple 11.6. Supposons que l’on cherche à analyser si la chaîne : « ‘somme’ ‘+’ ‘45’ FF » est conforme à la grammaire suivante :

(1) *expression* → *terme suite-expression*
(2) *suite-expression* → ‘+’ *terme expression*
(3) *suite-expression* → *ε*
(4) *terme* → *nom*
(5) *terme* → *cte*

dont le symbole de départ est *expression*.

La figure 11.3 montre les étapes d’une analyse *LL*(1) de cette chaîne.

Chaque ligne est associée à un état de l’analyse. La première colonne contient le numéro de cet état, la deuxième colonne contient le reste de la chaîne à analyser dans laquelle le prochain lexème apparaît en premier, la troisième colonne contient la chaîne dérivée et la quatrième colonne contient l’action effectuée qui fait passer l’analyse dans l’état suivant.

Observons par exemple l’état 5. Le symbole de gauche de la chaîne dérivée est le symbole *terme*. Deux règles ont ce symbole en partie gauche : les règles 4 et 5. Laquelle choisir ? C’est le fait de savoir que le prochain lexème est ‘45’ et donc une constante littérale entière qui permet de décider que c’est la règle 5 qui doit être appliquée produisant la nouvelle chaîne dérivée « *cte suite-expression* ». On peut vérifier que l’application de la règle 6 produirait la chaîne dérivée « *nom suite-expression* » et conduirait donc à un échec de l’analyse puisque le lexème ‘45’ n’est pas une constante littérale entière. □

Par la suite, nous dirons que le **symbole** *x* **a été effacé** si partant d’un état *i* où la chaîne dérivée était *x α* l’analyseur a atteint un état *j* (*j* > *i*) où la chaîne dérivée est *α*. Dire que le symbole *x* a été effacé c’est aussi dire qu’un constituant conforme à l’une des règles de la grammaire ayant le symbole *x* en partie gauche, a été reconnu à partir du caractère courant du texte source du programme.

Par exemple, dans l’état 3 de la figure 11.3, le symbole *terme* a été effacé car l’analyseur a atteint cet état à partir de l’état 1 où la chaîne dérivée était *terme suite-expression* et donc un terme a été reconnu : le terme ‘somme’ en l’occurrence.

Règles	Actions sémantiques
expression → *terme*$_1$ '+' *terme*$_2$	Soit r_1 le registre contenant la valeur de *terme*$_1$. Soit r_2 le registre contenant la valeur de *terme*$_2$. Générer le code « `add` r_1, r_1, r_2 ». La valeur de *expression* est contenue dans le registre r_1.
terme → *nom*	Soit *a* l'adresse de la variable *nom*. Générer le code « `load` *r*, *a* ». La valeur de *terme* est contenue dans le registre *r*.
terme → *cte*	Soit *r* un registre libre. Soit *v* la valeur de *cte*. Générer le code « `addi` *r*, `r0`, *v* ». La valeur de *terme* est contenue dans le registre *r*.

Figure 11.4. *Actions sémantiques associées aux règles d'une grammaire*

11.1.3 **Génération de code**

La génération de code consiste à traduire l'arbre syntaxique :

- soit en un code exécutable par la machine cible ;
- soit en un code abstrait qui pourra faire l'objet d'optimisations avant d'être traduit en un code exécutable par la machine cible.

La génération de code s'appuie sur les règles qui définissent la sémantique du langage de programmation. Elle est réalisée selon une technique appelée : « traduction dirigée par la syntaxe » qui consiste à associer à chaque règle de la grammaire une action dite « sémantique » qui vérifiera la sémantique et générera la portion du code correspondant au constituant reconnu. La génération du code d'un programme consiste alors à parcourir l'arbre syntaxique de ce programme depuis sa racine en appliquant la procédure récursive suivante : « Générer le code d'un arbre syntaxique dont la racine est associée au symbole non terminal *x* et ayant les sous-arbres y_1, ..., y_n, c'est générer le code associé à chacun de ces sous-arbres puis effectuer l'action associée au symbole *x* ».

La figure 11.4 montre les actions sémantiques associées aux règles de la grammaire de l'exemple 11.1, pour générer le code calculant la valeur de cette expression en termes d'une suite d'instructions de la pitchoun-machine.

On vérifiera facilement qu'en appliquant ces actions selon la procédure décrite ci-dessus, le code généré pour l'arbre syntaxique de la figure 11.1, construit par l'analyse de l'expression `somme + 15`, est le suivant :

```
load r1, adr(somme)   (la valeur de la variable somme est affectée au registre r1)
addi r2, r0, 15       (le nombre 15 est affecté au registre r2)
add r1, r1, r2        (le contenu du registre r1 est ajouté au contenu du registre r2,
                       le résultat est affecté au registre r1)
```

L'exécution de ce code aura pour effet d'affecter la valeur 77 au registre `r1`.

11.1.4 **Trois en une !**

En général un compilateur effectue l'analyse lexicale, l'analyse syntaxique et la génération de code au fur et à mesure de la lecture du programme source. On dit que ces trois phases sont effectuées en une seule passe. La chaîne de lexèmes et l'arbre syntaxique ne sont donc pas matérialisés.

C'est l'analyseur syntaxique qui pilote la compilation de la façon suivante, dans le cas d'une analyse *LL*(1) :

- il fait appel à l'analyseur lexical pour lire le prochain lexème (le premier) ;
- il lance la réécriture du symbole de départ de la grammaire ;
- chaque fois qu'un lexème a été consommé, il fait appel à l'analyseur lexical pour lire le prochain lexème ;
- chaque fois qu'un constituant a été effacé, les informations nécessaires à la génération du code sont mémorisées et si les informations déjà mémorisées le permettent, la partie de code correspondant à ce constituant est générée.

Un compilateur d'un langage de programmation peut être construit manuellement à partir de la définition des unités lexicales, de la grammaire *LL*(1) et des règles sémantiques de ce langage. C'est ce que nous ferons dans la suite de ce chapitre pour le langage J, car c'est un bon exercice de programmation. Mais il est beaucoup plus simple d'utiliser des outils tels que Lex et Yacc[2] (disponibles dans toute distribution d'UNIX) ou leurs variantes qui construisent automatiquement l'analyseur lexical et l'analyseur syntaxique et facilitent l'écriture des actions sémantiques. Yacc construit un analyseur syntaxique ascendant à partir d'une grammaire BNF dont les règles peuvent être récursives à gauche et non factorisées à gauche.

11.2 Le simulateur de la pitchoun-machine

Pour simuler la pitchoun-machine en C, il faut tout d'abord définir une représentation de sa mémoire, de ses registres et de ses instructions, puis définir la fonction qui simule l'exécution d'un programme.

11.2.1 **Représentation de la mémoire, des registres et des instructions**

Rappelons (voir chapitre 1, paragraphe 1.2.2) :

- que la mémoire de la pitchoun-machine est composée de 32 768 mots de 32 bits numérotés de 0 à 65 535 (l'adresse d'un mot est son numéro) ;
- qu'une instruction est découpée en 4 champs : le code de l'opération et ses arguments (0 à 3 selon l'instruction) qui peuvent être des numéros de registres, l'adresse d'une variable ou un nombre entier dans l'intervalle [0, 65 535] ;
- que l'unité arithmétique et logique comporte un jeu de 32 registres de 32 bits chacun ;
- que l'unité de contrôle comporte 2 registres : le registre `CO` qui contient l'adresse de la prochaine instruction à exécuter et le registre `RI` qui contient l'instruction à exécuter.

Lors de l'exécution d'un programme J, la mémoire de la pitchoun-machine sera découpée en deux zones :

[2] Acronyme de « Yet Another Compiler Compiler »

1. la zone des variables : une suite contiguë de mots dont chacun contient la valeur d'une variable qui est un entier relatif dans l'intervalle [−2 147 483 647; +2 147 483 647]. L'adresse d'une variable est l'adresse du mot qui contient sa valeur.
2. la zone du code : une suite contiguë de mots dont chacun contient une instruction du code exécutable du programme. Si ce code est rangé à partir de l'adresse a, le mot d'adresse $a + i - 1$ ($i > 0$) contient la i^e instruction.

Pour des raisons de simplicité et de lisibilité, nous avons choisi de représenter une instruction par une structure regroupant les champs de cette instruction plutôt que par un mot de 32 bits dont il aurait fallu extraire chacun de ces champs. Le type de cette structure est le suivant :

```
struct instruction
   {
   int op, a, b, c;
   };
```

où `op` est le code opération, `a` et `b` sont des numéros de registre et `c` est soit un numéro de registre (son rang dans le tableau `reg` défini ci-dessous), soit l'adresse d'une variable (son rang dans le tableau `memvar` défini ci-dessous), soit un nombre désigné par une constante littérale entière. Le code opération est un entier ≥ 0 qui sera désigné par l'une des constantes symboliques suivantes : `ADD`, `SUB`, `MULT`, `DIV`, `MOD`, `ADDI`, `SUBI`, `MULTI`, `DIVI`, `MODI`, `LOAD`, `STORE`, `READ`, `WRITE`, `STOP`.

En conséquence, la mémoire est représentée par deux tableaux :

– le tableau `memcode` qui contient le code du programme et dont la définition est la suivante :

  ```
  struct instruction memcode[NBINST];
  ```

  où la constante symbolique `NBINST` désigne le nombre maximum d'instructions d'un programme. L'adresse d'une instruction est son rang dans ce tableau. L'élément `memcode[`i`]` contient la $(i + 1)^e$ instruction du programme.

– le tableau `memvar` dont les éléments sont les variables du programme :

  ```
  long memvar[NBVAR];
  ```

  où la constante symbolique `NBVAR` désigne le nombre maximum de variables que peut contenir la mémoire. L'adresse d'une variable est son rang dans ce tableau.

Les registres de l'unité arithmétique et logique sont représentés par le tableau `reg` défini par :

```
int reg[NBREG];
```

où la constante symbolique `NBREG` désigne le nombre de registres. L'élément `reg[`i`]` contient la valeur du registre `r`i.

Les registres `CO` (compteur ordinal) et `RI` (registre instruction) de l'unité de contrôle sont représentés par les variables `co` et `ri` dont les définitions sont les suivantes :

```
int co;
struct instruction ri;
```

11.2.2 **Exécution d'un programme**

La pitchoun-machine exécute une instruction par cycle d'horloge. L'exécution d'un programme s'effectue selon l'algorithme suivant :

`CO` := 0 (adresse de la 1ère instruction à exécuter)
répéter
  charger dans le registre `RI` l'instruction dont l'adresse est contenue dans le registre `CO`
  `CO` := `CO` + 1 (adresse de la prochaine instruction à exécuter)
  décoder l'instruction contenue dans le registre `RI`
  si le code opération est `stop` **alors arrêter**
  exécuter l'instruction

Cette exécution est simulée par la fonction `executer` dont la définition est la suivante :

```
void executer(void)
   {
   int op, arg1, arg2;
   co = 0;
   do
      {
      ri = memcode[co];
      co++;
      op = ri.op;
      arg1 = reg[ri.b];
      if (op == ADD || op == SUB || op == MULT || op == DIV || op == MOD)
         arg2 = reg[ri.c];
      else
         arg2 = ri.c;
      switch (op)
         {
         case LOAD:
            reg[ri.a] = mem[arg2];
            break;
         case STORE:
            mem[arg2] = reg[ri.a];
            break;
         case ADD:
         case ADDI:
            reg[ri.a] = arg1 + arg2;
            break;
         case SUB:
         case SUBI:
            reg[ri.a] = arg1 - arg2;
            break;
         case MULT:
         case MULTI:
            reg[ri.a] = arg1 * arg2;
            break;
         case DIV:
         case DIVI:
            reg[ri.a] = arg1 / arg2;
            break;
         case MOD:
         case MODI:
            reg[ri.a] = arg1 % arg2;
            break;
         case READ:
            printf("? ");
            scanf("%d", &reg[1]);
            break;
         case WRITE:
            printf("= %d\n", reg[1]);
            break;
```

programme → *déclaration liste-instructions* ‘`.`’
déclaration → ‘`var`’ *liste-noms* ‘`;`’
liste-noms → *nom suite_liste-noms*
suite_liste-noms → ‘`,`’ *nom suite_liste-noms*
suite_liste-noms → ε
liste-instructions → *instruction suite_liste-instructions*
suite_liste-instructions → ‘`;`’ *instruction suite_liste-instructions*
suite_liste-instructions → ε
instruction → *nom* ‘`:=`’ *expression*
instruction → ‘`saisir`’ *nom*
instruction → ‘`afficher`’ *nom*
expression → *facteur suite-expression*
suite-expression → *opérateur-additif facteur suite-expression*
suite-expression → ε
facteur → *terme suite-facteur*
suite-facteur → *opérateur-multiplicatif terme suite-facteur*
suite-facteur → ε
terme → *nom*
terme → *constante*
terme → ‘`(`’ *expression* ‘`)`’
terme → ‘`-`’ *terme*
opérateur-additif → ‘`+`’
opérateur-additif → ‘`-`’
opérateur-multiplicatif → ‘`*`’
opérateur-multiplicatif → ‘`/`’
opérateur-multiplicatif → ‘`%`’

Figure 11.5. *Grammaire LL(1) du langage J*

```
        case STOP:
           break;
        }
     }
while (op != STOP);
}
```

La variable `op` contient le code opération et les variables `arg1` et `arg2` contiennent la valeur des opérandes.

11.3 Le compilateur J

Le compilateur que nous allons programmer en C opère en une seule passe sur le texte du programme (voir ci-dessus, paragraphe 11.1.4). Il est basé sur un analyseur syntaxique par « descente récursive ». En quelques mots, cette technique consiste à associer à chaque symbole non terminal *x* de la grammaire une fonction de même nom qui essaie d’effacer le symbole *x* et qui retourne vrai si elle réussit et faux sinon.

La technique de descente récursive nécessite une grammaire *LL*(1). Il faut donc tout d’abord transformer la grammaire du langage J telle qu’elle a été définie au chapitre 1 (voir figure 1.2)

en une grammaire *LL*(1). Le résultat de cette transformation est présenté sur la figure 11.5. Les transformations appliquées ont consisté :

– à éliminer la récursivité à gauche et à appliquer la factorisation à gauche dans les règles décrivant les listes de noms, les listes d'instructions et les expressions comme cela a été expliqué dans les exemples 11.3 et 11.4 ;
– à intégrer la prise en compte de la priorité des opérateurs multiplicatifs sur les opérateurs additifs comme cela a été expliqué dans l'exemple 11.5.

Afin de simplifier l'écriture de l'analyseur syntaxique, nous avons introduit une contrainte supplémentaire. Dans chaque règle de la forme :

$x \rightarrow y\ \alpha$

où α peut être vide, le symbole *y* n'est pas annulable, c.-à-d. que la chaîne vide ne peut pas être dérivée à partie du symbole *x*. Cela signifie que si, en application de cette règle, le symbole *y* a été effacé et que l'un des symboles de α ne peut pas l'être, il y a une erreur de syntaxe dans le texte du programme. En effet, si le symbole *y* a été effacé, il l'a été au vu du prochain lexème et donc, par définition d'une grammaire *LL*(1), il ne peut pas exister d'autre règle permettant de d'effacer le symbole *x*.

Supposons, par exemple, qu'une instruction doit être reconnue et que le prochain lexème soit '`saisir`', la seule règle applicable est la règle :

instruction → '`saisir`' *nom*

Si une fois ce lexème consommé, le prochain lexème n'est pas un nom de variable, alors c'est une erreur de syntaxe car il n'y a pas d'autre règle applicable.

Le compilateur opère dans un environnement défini par les variables globales suivantes :

– La variable `source` définie par :

```
FILE *source;
```

contient le descripteur du fichier source (celui qui contient le texte du programme).

– La variable `noligne` définie par :

```
int noligne = 1;
```

contient le numéro de la ligne courante dans le fichier source.

– La chaîne de caractères `message` définie par :

```
char message[50];
```

contient un message d'erreur.

– Un lexème est représenté par une structure dont le type est le suivant :

```
struct lexeme
   {
   int type;
   char nom[LNOM + 1];
   long nb;
   };
```

Le champ `type` contient le type du lexème désigné par l'une des constantes symboliques suivantes `ADD` ('+'), `DIV` ('/'), `FF` (*fin de fichier*), `INCONNU` (*lexème inconnu*), `MOD` ('%'), `MULT` ('*'), `NB` (*constante littérale*), `NOM` (*nom de variable*), `PARFER` (')'), `PAROUV` ('('), `PTVIRG` (';'), `READ` ('saisir'), `STOP` ('.'), `STORE` (':='), `SUB` ('+'), `VIRG` (','), `WRITE` ('afficher'). Si le lexème est un nom de variable, le champ `nom` contient ce nom (`LNOM` est

le nombre maximum de caractères d'un nom). Si le lexème est une constante littérale, de variable, le champ `nb` contient le nombre représenté par cette constante (≤ `CTEMAX` qui désigne la valeur maximum d'une constante).

- Les variables `prochain_lexeme` et `lexeme_courant` définies par :

```
struct lexeme prochain_lexeme, lexeme_courant;
```

  contiennent le prochain lexème et le lexème courant (celui qui a été consommé).

- La correspondance entre un nom de variable et son adresse est mémorisée dans le tableau `tabvar` défini par :

```
char tabvar[NBVAR][LNOM + 1];
```

  où `NBVAR` est le nombre maximum de variables d'un programme. L'élément `tabvar[`*i*`]` (0 ≤ *i* < `NBVAR`) contient le nom de la variable d'adresse *i*.

- La variable `ivar` définie par :

```
int ivar = 0;
```

  contient le nombre de variables déjà enregistrées dans le tableau `tabvar`.

Lorsqu'une erreur est détectée, un message est affiché indiquant le numéro de la ligne où cette erreur a été détectée et le type de cette erreur et l'exécution du programme est interrompue. Cette action est réalisée par la fonction `erreur` définie de la façon suivante :

```
void erreur(char *m)
   {
   printf("Erreur ligne %d : %s\n", noligne, m);
   fclose(source);
   exit(1);
   }
```

11.3.1 **L'analyseur lexical**

L'analyseur lexical est appelé par l'analyseur syntaxique chaque fois que le prochain lexème doit être lu. Il lit le prochain lexème à partir de la position courante dans le fichier source du programme.

Les unités lexicales du langage J sont décrites par l'expression régulière suivante :

| *constante*
| *nom*
| *affectation*
| '+' | '-' | '*' | '/' | '%' | '(' | ')' | ',' | ';' | '.'

où :

chiffre ≡ '0' | '1' | ... | '9'
lettre ≡ 'a' | 'b' | ... | 'z'
constante ≡ *chiffre chiffre+*
nom ≡ *lettre* (*lettre* | *chiffre*)*
affectation ≡ ':' '='

De cette définition, on peut déduire :

- qu'une constante littérale entière commence par un chiffre ;
- qu'un nom de variable commence par une lettre ;
- que le symbole d'affectation commence par un deux-points ;

– que chaque opérateur ou chaque séparateur est représenté par un caractère différent qui n'est ni une lettre, ni un chiffre, ni un deux-points ;

et donc qu'au vu du premier caractère qui n'est pas un caractère d'espacement, il est possible de reconnaitre le type de lexème à lire.

S'appuyant sur cette déduction, la lecture du prochain lexème sera réalisée selon l'algorithme suivant :

sauter les caractères d'espacement (blanc, tabulation, nouvelle ligne)
soit *plex* la structure décrivant le prochain lexème
soit *c* le 1er caractère qui n'est pas un espacement
si *c* est un chiffre (début de constante littérale entière) **alors**
  lire les chiffres suivants de cette constante, traduire la suite de chiffres en un nombre,
  affecter le type `NB` et ce nombre aux champs `type` et `nb` de la structure *plex*
sinon si *c* est une lettre (début de nom de variable) **alors**
  lire les lettres ou les chiffres suivants de ce nom,
  si ce nom est un mot-réservé **alors**
    affecter le type correspondant (`DEBUT READ STOP VAR WRITE`)
    au champ `type` de la structure *plex*
  sinon affecter `NOM` et ce nom aux champs `type` et `nom` de la structure *plex*
sinon si *c* est un caractère « `:` » **alors**
  si le caractère suivant est « `=` » (début du symbole d'affectation) **alors**
    affecter `STORE` au champ `type` de la structure *plex*
  sinon affecter `INCONNU` au champ `type` de la structure *plex*
sinon si *c* est l'un des caractères `+ - * / % ( ) , ; .` (opérateur, séparateur ou fin de programme) **alors**
  affecter le type correspondant (`ADD SUB MULT DIV MOD PAROUV PARFER VIRG PTVIRG STOP`) au champ `type` de la structure *plex*
sinon affecter le type `INCONNU` au champ `type` de la structure *plex*

Si aucun lexème n'est reconnu, un lexème de type `INCONNU` est retourné à l'analyseur syntaxique qui émettra le message d'erreur indiquant le type de lexème attendu.

Il est important de remarquer que c'est la lecture du premier caractère qui n'est pas un chiffre qui marque la fin de la lecture d'une constante littérale entière. De même, c'est la lecture du premier caractère qui n'est pas une lettre ou un chiffre qui marque la fin de la lecture d'un nom de variable. Or ce caractère lu en plus, s'il n'est pas un caractère d'espacement, appartient au lexème suivant. Il faudra donc le « délire » c.-à-d. le remettre dans le fichier, afin qu'il puisse être pris en compte lors de la lecture suivant.

Par exemple, si le texte du programme est « `x1:=35` » la lecture du prochain lexème retournera le nom de variable `x1` mais aura lu le caractère « `:` » en plus. Il faudra donc remettre ce caractère dans le fichier pour que la lecture du prochain lexème retourne bien le symbole d'affectation « `:=` ».

Pour effectuer une « délecture », il faut utiliser la fonction `ungetc` d'en-tête :

```
int ungetc(int c, FILE *f)
```

déclaré dans le fichier `stdio.h`. L'appel `ungetc(`*c*`,` *f*`)` remet le caractère *c* dans le fichier *f* afin que le prochain `fgetc(`*f*`)` retourne ce caractère.

La lecture du prochain lexème est réalisée conformément à l'algorithme ci-dessus par la fonction `lire_prochain_lexeme` dont la définition est la suivante :

```
void lire_prochain_lexeme()
   {
   int c, i;
   long nb ;
   c = lire_car();
   while (isspace(c))
      c = lire_car();
   /*
    * Reconnaissance d'une constante littérale entière
    */
   if (isdigit(c))
      {
      nb = 0;
      do
         {
         if (nb > (CTEMAX - c + '0') / 10)
            erreur("constante trop grande");
         nb = 10 * nb + c - '0';
         c = lire_car();
         }
      while (isdigit(c));
      ungetc(c, source);
      prochain_lexeme.type = NB;
      prochain_lexeme.nb = nb;
      }
   /*
    * Reconnaissance d'un nom
    */
   else if (isalpha(c))
      {
      i = 0;
      do
         {
         prochain_lexeme.nom[i++] = c;
         c = lire_car();
         }
      while (i < LNOM && isalnum(c));
      prochain_lexeme.nom[i++] = '\0';
      if (strcmp(prochain_lexeme.nom, "saisir") == 0)
         prochain_lexeme.type = READ;
      else if (strcmp(prochain_lexeme.nom, "var") == 0)
         prochain_lexeme.type = VAR;
      else if (strcmp(prochain_lexeme.nom, "afficher") == 0)
         prochain_lexeme.type = WRITE;
      else
         prochain_lexeme.type = NOM;
      }
   /*
    * Reconnaissance des opérateurs et des séparateurs
    */
   else if (c == '+')
      prochain_lexeme.type = ADD;
   else if (c == '-')
      prochain_lexeme.type = SUB;
   else if (c == '*')
      prochain_lexeme.type = MULT;
```

```
   else if (c == '/')
      prochain_lexeme.type = DIV;
   else if (c == '%')
      prochain_lexeme.type = MOD;
   else if (c == '(')
      prochain_lexeme.type = PAROUV;
   else if (c == ')')
      prochain_lexeme.type = PARFER;
   else if (c == ',')
      prochain_lexeme.type = VIRG;
   else if (c == ';')
      prochain_lexeme.type = PTVIRG;
   else if (c == '.')
      prochain_lexeme.type = PTVIRG;
   else if (c == ':')
      {
      c = lire_car();
      if (c == '=')
         prochain_lexeme.type = STORE;
      else
         prochain_lexeme.type = INCONNU;
      }
   else if (c == EOF)
      prochain_lexeme.type = FF;
   else
      prochain_lexeme.type = INCONNU;
   }
```

La fonction `lire_car` lit le prochain caractère dans le fichier source et incrémente le compteur de ligne (`noligne`) si ce caractère est le caractère « nouvelle ligne ». La définition de cette fonction est la suivante :

```
int lire_car(void)
   {
   int c;
   c = fgetc(source);
   if (c == '\n')
      noligne++;
   return c;
   }
```

11.3.2 **L'analyseur syntaxique et le générateur de code**

L'analyseur syntaxique est un analyseur par descente récursive. A chaque symbole non terminal de la grammaire est associée une fonction qui cherche à effacer ce symbole et qui, si elle a réussi, génère la partie du code exécutable correspondant au constituant reconnu.

Définir ces fonctions, nécessite de bien comprendre les points suivants : effacement d'un symbole, allocation des variables en mémoire et codage des expressions.

Effacement d'un symbole

Le résultat de l'effacement d'un symbole est 0 si il a échoué ou un entier positif si il a réussi (1 ou le code de l'opérateur si c'est un opérateur qui a été reconnu).

L'effacement d'un symbole terminal *t* est effectué par l'appel de la fonction `accepter` dont la définition est la suivante :

```
int accepter(int t)
   {
   if (prochain_lexeme.type == t)
      {
      lexeme_courant = prochain_lexeme;
      lire_prochain_lexeme();
      return 1;
      }
   else
      return 0;
   }
```

Cet appel teste si le prochain lexème est conforme au symbole terminal *t* et si oui, lit le prochain lexème.

L'effacement d'un symbole non terminal *x* est effectué par l'appel de la fonction *x* d'en-tête :

```
int x(void)
```

Si le paquet de règles ayant le symbole *x* en partie gauche a la forme suivante :

$x \rightarrow y_1\ z_{11}\ z_{12}$ ...
...
$x \rightarrow y_n\ z_{n1}\ z_{n2}$...
$x \rightarrow \varepsilon$ (éventuellement)

où y_1, ..., y_n sont des symboles non annulables, la définition de la fonction *x* est conforme au schéma suivant :

```
int x(void)
   {
   if (effacer y1)
      {
      if (!effacer z11)
         erreur("z11 attendu");
      ...
      return entier positif;
      }
   ...
   if (effacer yn)
      {
      if (!effacer zn1)
         erreur("zn1 attendu");
      ...
      return entier positif;
      }
   return 0 si la règle x → ε n'est pas présente, return 1 si elle est présente
   }
```

L'analyseur essaye successivement chaque règle du paquet de règles :

- Si un symbole y_i a été effacé et que l'un des symboles z_{ij} ne peut pas l'être, c'est qu'il y a une erreur de syntaxe car il n'y a pas, par construction de la grammaire, d'autre règle applicable pour effacer *x* : le message d'erreur « z_{ij} attendu » est affiché et la compilation est interrompue. Si tous les symboles z_{ij} ont été effacés, c'est que *x* a été effacé.
- Si aucun symbole y_i ne peut être effacé, c'est que seule la chaîne vide a été reconnue et donc que :

- *x* a été effacé si la règle $x \rightarrow \varepsilon$ est présente ;
- *x* n'est pas été effacé si la règle $x \rightarrow \varepsilon$ n'est pas présente.

Par exemple, la fonction qui reconnaît une déclaration de variables dont la syntaxe doit être conforme à la règle :

déclaration → '`var`' *liste-noms* '`;`'

est définie de la façon suivante

```
int declaration(void)
   {
   if (accepter(VAR))
      {
      if (!liste_noms())
         erreur("liste-noms attendue");
      if (!accepter(PTVIRG))
         erreur("';' attendu");
      return 1;
      }
   return 0;
   }
```

Autre exemple, la fonction qui reconnaît un opérateur additif dont la syntaxe doit être conforme aux règles :

opérateur-additif → '+'
opérateur-additif → '-'

est définie de la façon suivante :

```
int operateur_additif(void)
   {
   if (accepter(ADD))
      return ADD;
   if (accepter(SUB))
      return SUB;
   return 0;
   }
```

Allocation des variables en mémoire

Lorsqu'un nom de variable apparaît dans une instruction, l'analyseur doit connaître l'adresse de cette variable, c.-à-d. le rang de l'élément du tableau `memvar` qui contient la valeur de cette variable. La correspondance entre un nom de variable et son adresse est mémorisée dans le tableau `tabvar` qui est rempli lors de l'analyse de la déclaration des variables.

Par exemple, si la déclaration des variables est la suivante :

```
var longueur, largeur, perimetre;
```

l'adresse 0 sera affectée à la variable `longueur`, l'adresse 1 à la variable `largeur` et l'adresse 2 à la variable `perimetre`. L'élément `tabvar[0]` contiendra donc la chaîne de caractères « longueur », l'élément `tabvar[1]` la chaîne de caractères « largeur » et l'élément `tabvar[2]` la chaîne de caractères « perimetre ».

L'ajout d'une variable est réalisé par la fonction `ajouter_variable` dont la définition est la suivante :

```
void ajouter_variable(char *nom)
   {
   int i = 0;
```

```
   while (i < ivar && strcmp(nom, tabvar[i]) != 0)
      i++;
   if (i == ivar)
      if (ivar < NBVAR)
         strcpy(tabvar[ivar++], lexeme_courant.nom);
      else
         erreur("memoire saturee");
   else
      erreur("variable deja definie");
   }
```

S'il existe déjà une variable de même nom ou si le nombre maximum de variables a été atteint, une erreur est signalée et la compilation est interrompue.

La recherche de l'adresse d'une variable dont le nom *n* est donné est très simple. Elle consiste à rechercher dans le tableau `tabvar` le rang de l'élément qui contient la chaîne de caractères *n*. Cette recherche est réalisée par la fonction `adresse_variable` dont la définition est la suivante :

```
int adresse_variable(char *nom)
   {
   int i = 0;
   while (i < ivar && strcmp(nom, tabvar[i]) != 0)
      i++;
   if (i == ivar)
      {
      sprintf(message, "variable %s non declaree", nom);
      erreur(message);
      }
   return i;
   }
```

Codage d'une expression

Pour calculer la valeur d'une expression comportant plusieurs opérations, il est nécessaire de mémoriser des résultats intermédiaires puisque l'unité arithmétique et logique ne réalise qu'une seule opération à la fois. Par exemple, pour calculer la valeur de l'expression `2 * (longueur + largeur)`, il faut tout d'abord calculer la valeur de l'expression `longueur + largeur`, la mémoriser puis la multiplier par 2. Dans une machine à registres, comme la pitchoun-machine, les résultats intermédiaires sont stockés dans les registres et s'il n'y en a plus de disponibles, en mémoire centrale.

Une façon simple de générer le code permettant de calculer la valeur d'une expression est d'appliquer le principe des machines à pile. Dans une machine à pile, la mémoire est organisée sous la forme d'une pile. Les valeurs des opérandes d'une opération arithmétique *n*-aire sont les *n* valeurs placées au sommet de la pile. Une fois cette opération effectuée, sa valeur vient remplacer celles de ses opérandes et se trouve donc au sommet de la pile. Afin d'illustrer ce principe, considérons une machine à pile très simple, munie des quatre opérations suivantes :

`push_var` *a*	place la valeur de la variable d'adresse *a* au sommet de la pile
`push_cte` *c*	place le nombre désigné par la constante *c* au sommet de la pile
`add`	remplace les deux nombres au sommet de la pile par leur somme
`mult`	remplace les deux nombres au sommet de la pile par leur produit

La partie gauche de la figure 11.5 montre le code qui pourrait être généré pour le calcul par cette machine de la valeur de l'expression `2 * (longueur + largeur)` et son exécution.

Machine à pile		Pitchoun-machine	
`push_cte 2`	► 2	`addi r1, r0, 2`	r3 ; r2 ; ►r1 2 ; r0 0
`push_var` *adr*(`longueur`)	► 8 ; 2	`load r2,` *adr*(`longueur`)	r3 ; ►r2 8 ; r1 2 ; r0 0
`push_var` *adr*(`largeur`)	► 5 ; 8 ; 2	`load r3,` *adr*(`largeur`)	►r3 5 ; r2 8 ; r1 2 ; r0 0
`add`	► 5 ; 13 ; 2	`add r2, r2, r3`	r3 5 ; ►r2 13 ; r1 2 ; r0 0
`mult`	► 5 ; 13 ; 26	`mult r1, r1, r2`	r3 5 ; r2 8 ; ►r1 26 ; r0 0

Figure 11.5. *Machine à pile (► pointeur de sommet de pile)*

La pitchoun-machine n'est pas une machine à pile mais une machine à registres. Cependant, il est possible de la faire fonctionner comme telle en utilisant les registres comme pile. Il faut tout d'abord définir une variable `ireg` jouant le rôle de pointeur de sommet de pile : sa valeur est le numéro du registre sommet de la pile. Au début de l'analyse d'une expression `ireg` est initialisé à 0 (la pile est vide). La génération du code se déroule alors selon le processus suivant :

– lorsqu'une constante littérale a été reconnue, `ireg` est incrémentée et l'instruction suivante est générée :

  `addi r`*ireg*`, r0,` *c*

– lorsqu'un nom de variable *v* a été reconnu, `ireg` est incrémentée de 1 et l'instruction suivante est générée :

  `load r`*ireg*`,` *adr*(*v*)

– lorsque l'expression –*t* a été reconnue, l'instruction suivante est générée :

  `sub r`*ireg*`, r0, r`*ireg*

– lorsque l'expression *terme*$_1$ *op terme*$_2$ (où *op* est un opérateur multiplicatif) a été reconnue, l'instruction suivante est générée :

  op `r`*ireg-1*`, r`*ireg*`, r`*ireg*

et `ireg` est décrémentée ;

– lorsque l'expression *facteur*$_1$ *op facteur*$_2$ (où *op* est un opérateur additif) a été reconnue, l'instruction suivante est générée :

op `r`*ireg-1*`, r`*ireg*`, r`*ireg*

et `ireg` est décrémentée.

C'est ce processus qui est mis en œuvre dans le compilateur J.

La partie droite de la figure 11.5 montre le code généré pour le calcul de la valeur de l'expression `2 * (longueur + largeur)` en appliquant ce processus, ainsi que son exécution. Il faut noter que ce code n'est optimal ni en ce qui concerne le nombre de registres nécessaires, ni en ce qui concerne le nombre d'instructions. Un compilateur plus sophistiqué aurait généré le code suivant :

```
load r1, adr(longueur)
load r2, adr(largeur)
add r1, r1, r2
multi r1, r1, 2
```

qui n'utilise que 2 registres au lieu de 3 et ne comporte que 4 instructions au lieu de 5.

L'analyseur syntaxique

Tous les éléments sont maintenant réunis pour programmer l'analyseur syntaxique. Il est composé des fonctions qui effacent un symbole non terminal de la grammaire et donc qui reconnaissent un constituant conforme à ce symbole et génèrent la portion de code exécutable correspondante. Les définitions de ces fonctions sont données ci-dessous.

– Reconnaissance d'un programme

```
int programme(void)
   {
   if (declaration())
      {
      if (!liste_instructions())
         erreur("liste-instructions attendue");
      if (!accepter(STOP))
         erreur("'.' attendu");
      if (prochain_lexeme.type != FF)
         erreur("'fin de fichier' attendue");
      ajouter_instruction(STOP, 0, 0, 0);
      return 1;
      }
   return 0;
   }
```

– Reconnaissance d'une déclaration de variables

```
int declaration(void)
   {
   if (accepter(VAR))
      {
      if (!liste_variables())
         erreur("liste-noms attendue");
```

```
      if (!accepter(PTVIRG))
         erreur("';' attendu");
      return 1;
      }
   return 0;
   }
```

– Reconnaissance d'une liste de variables :

```
int liste_variables(void)
   {
   if (accepter(NOMVAR))
      {
      ajouter_variable(lexeme.nom);
      if (!suite_liste_variables())
         erreur("suite-liste-noms attendue");
      return 1;
      }
   return 0;
   }

int suite_liste_variables(void)
   {
   if (accepter(VIRG))
      {
      if (!accepter(NOMVAR))
         erreur("nom de variable attendu");
      ajouter_variable(lexeme.nom);
      if (!suite_liste_variables())
         erreur("liste-noms attendue");
      }
   return 1;
   }
```

– Reconnaissance d'une liste d'instructions :

```
int liste_instructions(void)
   {
   if (instruction())
      {
      if (!suite_liste_instructions())
         erreur("suite-liste-instructions attendue");
      return 1;
      }
   return 0;
   }

int suite_liste_instructions(void)
   {
   if (accepter(PTVIRG))
      {
      if (!instruction())
         erreur("instruction attendue");
      if (!suite_liste_instructions())
         erreur("suite-liste-instructions attendue");
      }
   return 1;
   }
```

– Reconnaissance d'une instruction :

```
int instruction(void)
   {
   int a;
   if (accepter(NOMVAR))
      {
      a = adresse_variable(lexeme.nom);
      if (!accepter(STORE))
         erreur("':=' attendu");
      if (!expression())
         erreur("expression attendue");
      ajouter_instruction(STORE, ireg, 0, a);
      ireg--;
      return 1;
      }
   if (accepter(READ))
      {
      if (!accepter(NOMVAR))
         erreur("nom de variable attendu");
      a = adresse_variable(lexeme.nom);
      ajouter_instruction(READ, 0, 0, 0);
      ajouter_instruction(STORE, 1, 0, a);
      return 1;
      }
   if (accepter(WRITE))
      {
      if (!accepter(NOMVAR))
         erreur("nom de variable attendu");
      a = adresse_variable(lexeme.nom);
      ajouter_instruction(LOAD, 1, 0, a);
      ajouter_instruction(WRITE, 0, 0, 0);
      return 1;
      }
   return 0;
   }
```

– Reconnaissance d'une expression :

```
(1)  int expression(void)
(2)     {
(3)     if (facteur())
(4)        {
(5)        if (!suite_expression())
(6)           erreur("suite-expression attendue");
(7)        return 1;
(8)        }
(9)     return 0;
(10)    }

(11) int suite_expression(void)
(12)    {
(13)    int op;
(14)    if ((op = operateur_additif()))
(15)       {
(16)       if (!facteur())
(17)          erreur("facteur attendu");
(18)       ajouter_instruction(op, ireg - 1, ireg - 1, ireg);
(19)       ireg--;
```

```
(20)        if (!suite_expression())
(21)           erreur("suite-expression attendu");
(22)        return 1;
(23)        }
(24)     return 1;
(25)     }
```

Supposons que l'expression soit de la forme f_1 *op* f_2 *op* f_3 et que la variable `ireg` qui contient le numéro du registre sommet de pile ait la valeur 0. L'appel `facteur()` (ligne 3) allouera le registre `r1` à la valeur de f_1. Puis la fonction `suite_expression` sera appelée une 1ère fois (ligne 5). L'appel de la fonction `facteur` (ligne 16) allouera le registre `r2` à la valeur de f_2, puis le registre `r1` sera alloué à la valeur de f_1 *op* f_2 (ligne 18). La fonction `suite_expression` sera appelée une 2e fois (ligne 20). L'appel de la fonction `facteur` (ligne 16) allouera le registre `r2` à la valeur de f_2, puis le registre `r1` sera alloué à la valeur de f_1 *op* f_2 *op* f_3 (ligne 18). Lors de l'évaluation de cette expression, le registre `r1` servira d'accumulateur, puisqu'il contiendra successivement la valeur de f_1, puis la valeur de f_1 *op* f_2, puis la valeur de (f_1 *op* f_2) *op* f_3.

– Reconnaissance d'un facteur :

```
int facteur(void)
   {
   if (terme())
      {
      if (!suite_facteur())
         erreur("suite-facteur attendu");
      return 1;
      }
   return 0;
   }

int suite_facteur(void)
   {
   int op;
   if ((op = operateur_multiplicatif()))
      {
      if (!terme())
         erreur("terme attendu");
      ajouter_instruction(op, ireg - 1, ireg - 1, ireg);
      ireg--;
      if (!suite_facteur())
         erreur("suite-facteur attendu");
      return 1;
      }
   return 1;
   }
```

L'allocation des registres se déroule de la même façon que lors de la reconnaissance d'une expression.

– Reconnaissance d'un terme :

```
int terme(void)
   {
   if (accepter(PAROUV))
      {
      if (!expression())
         erreur("expression attendue");
```

```
      if (!accepter(PARFER))
         erreur("')' attendu");
      return 1;
      }
   if (accepter(SUB))
      {
      if (!terme())
         erreur("terme attendu");
      ajouter_instruction(SUB, ireg, 0, ireg);
      return 1;
      }
   if (accepter(NB))
      {
      if (ireg == NBREG)
         erreur("plus de registres disponibles");
      ireg++;
      ajouter_instruction(ADDI, ireg, 0, lexeme.nb);
      return 1;
      }
   if (accepter(NOMVAR))
      {
      if (ireg == NBREG)
         erreur("plus de registres disponibles");
      ireg++;
      ajouter_instruction(LOAD, ireg, 0, adresse_variable(lexeme.nom));
      return 1;
      }
   return 0;
   }
```

Le registre `ireg` est affecté à la valeur du terme.

– Reconnaissance d'un opérateur additif :

```
int operateur_additif(void)
   {
   if (accepter(ADD))
      return ADD;
   if (accepter(SUB))
      return SUB;
   return 0;
   }
```

– Reconnaissance d'un opérateur multiplicatif :

```
int operateur_multiplicatif(void)
   {
   if (accepter(MULT))
      return MULT;
   if (accepter(DIV))
      return DIV;
   if (accepter(MOD))
      return MOD;
   return 0;
   }
```

Ces fonctions font appel aux fonctions : `erreur`, `accepter`, `ajouter_variable` et `adresse_variable` dont les définitions ont déjà été données et à la fonction `ajouter_instruction` qui ajoute une instruction dans la mémoire code et qui est définie de la façon suivante :

```
void ajouter_instruction(int op, int a, int b, int c)
   {
   if (icode == NBINST)
      erreur("programme trop long");
   memcode[icode].op = op;
   memcode[icode].a = a;
   memcode[icode].b = b;
   memcode[icode].c = c;
   icode++;
   }
```

11.4 L'interface : compilation et exécution d'un programme J

La compilation et l'exécution d'un programme J est lancée par la fonction `main` qui reçoit en argument le nom du fichier contenant le code source de ce programme : le fichier source.

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

```
int main(int argc, char *argv[])
   {
   int r;
   source = fopen(argv[1], "r");
   if (source == NULL)
      {
      printf("Fichier source inconnu !");
      exit(1);
      }
   analex();
   r = programme();
   if (!r)
      erreur("programme attendu");
   fclose(source);
   afficher_code();
   printf("\nExecution\n");
   executer();
   return 0;
   }
```

Si le fichier source existe, il est ouvert, sinon une erreur est signalée et l'exécution est interrompue. Le premier lexème est lu, puis l'effacement du symbole *programme* est lancée. Si elle a échoué, une erreur est signalée et l'exécution est interrompue. Si elle a réussi, c'est que le code exécutable du programme a pu être généré : ce code est affiché, après fermeture du fichier source. L'exécution du code exécutable est alors lancée.

L'affichage du code exécutable est effectué par la fonction `afficher_code` dont la définition est la suivante :

```
void afficher_code(void)
   {
   int i, a, b, c;
   printf("\nCode executable\n");
   for (i = 0; i < icode; i++)
      {
      a = memcode[i].a;
      b = memcode[i].b;
      c = memcode[i].c;
```

```
        switch (memcode[i].op)
           {
           case ADD:
              printf("add r%d, r%d, r%d\n", a, b, c);
              break;
           case SUB:
              printf("sub r%d, r%d, r%d\n", a, b, c);
              break;
           case MULT:
              printf("mult r%d, r%d, r%d\n", a, b, c);
              break;
           case DIV:
              printf("div r%d, r%d, r%d\n", a, b, c);
              break;
           case MOD:
              printf("mod r%d, r%d, r%d\n", a, b, c);
              break;
           case ADDI:
              printf("addi r%d, r%d, %d\n", a, b, c);
              break;
           case SUBI:
              printf("subi r%d, r%d, %d\n", a, b, c);
              break;
           case MULTI:
              printf("multi r%d, r%d, %d\n", a, b, c);
              break;
           case DIVI:
              printf("divi r%d, r%d, %d\n", a, b, c);
              break;
           case MODI:
              printf("modi r%d, r%d, %d\n", a, b, c);
              break;
           case LOAD:
              printf("load r%d, %d\n", a, c);
              break;
           case STORE:
              printf("store r%d, %d\n", a, c);
              break;
           case READ:
              printf("read\n");
              break;
           case WRITE:
              printf("write\n");
              break;
           case STOP:
              printf("stop\n");
              break;
           }
        }
     }
```

11.5 Quelques exemples

Il est temps d'expérimenter notre travail. Nous le ferons en compilant puis exécutant deux programme : le programme *Périmètre* et le programme *Polynôme* qui calcule la valeur d'une fonction polynôme du second degré.

11.5.1 **Programme *Périmètre***

Le texte du programme *Périmètre* est, rappelons le, le suivant :

```
var longueur, largeur, perimetre;
saisir longueur;
saisir largeur;
perimetre := 2 * (longueur + largeur);
afficher perimetre.
```

Lançons son exécution en supposant qu'il est enregistré dans le fichier `perimetre.j` :

```
j perimetre.j↵

Code executable
read             (r1 contiendra la valeur saisie)
store r1, 0      (longueur à l'adresse 0 contiendra la valeur saisie)
read             (r1 contiendra la valeur saisie)
store r1, 1      (largueur à l'adresse 0 contiendra la valeur saisie)
addi r1, r0, 2   (r1 contiendra 2)
load r2, 0       (r2 contiendra la valeur de longueur)
load r3, 1       (r3 contiendra la valeur de largeur)
add r2, r2, r3   (r2 contiendra la valeur de longueur + largeur)
mult r1, r1, r2  (r1 contiendra la valeur de 2 * (longueur + largeur))
store r1, 2      (perimetre à l'adresse 0 contiendra la valeur de
                    2 * (longueur + largeur))
load r1, 2       (r1 contiendra la valeur de perimetre)
write            (affichage de la valeur de perimetre)
stop

Execution
? 8↵             (saisie de la longueur)
? 5↵             (saisie de la largeur)
= 26             (valeur du périmètre)
```

Afin de tester la détection d'erreur, modifions le texte du programme *Périmètre* de la façon suivante :

```
var longueur, largeur perimetre;
saisir longueur;
saisir largeur;
perimetre := 2 * (lngueur + largeur);
afficher perimetre.
```

c.-à-d. en oubliant une virgule dans la déclaration des variables de la 1ère ligne et le « o » de `longueur` dans la 4e ligne. Lançons l'exécution de ce programme :

```
j perimetre.j↵
Erreur ligne 1 : ';' attendu
```

Pourquoi un tel message au lieu de « `Erreur ligne 1 : ',' attendu` » puisque c'est la virgule qui semble avoir été oubliée. Après avoir reconnu le nom de variable `largeur`, l'analyseur a sauté le caractère blanc et lu le prochain lexème qui est le nom `perimetre`. N'ayant pas lu une virgule, il a appliqué la règle :

suite_liste-noms → ε

et donc fini de reconnaître une liste de variables. Il s'attend donc à ce que le prochain lexème soit le point-virgule qui sépare la déclaration de variables de la 1ère instruction du programme. Ce n'est pas le cas, puisque ce prochain lexème est un nom. Il émet donc le message « `Erreur ligne 1 : ';' attendu` ». Le signalement des erreurs est un problème difficile en compilation.

Après avoir corrigé, cette première erreur, relançons l'exécution du programme :

```
j perimetre.j↵
Erreur ligne 4 : variable lngueur non déclarée
```

Dans ce cas, le message est parfaitement clair : la variable `lngueur` n'a pas été déclarée puisque c'est la variable `longueur` qui l'a été ! Après correction de cette erreur, le programme s'exécutera correctement.

11.5.2 **Programme *Polynôme***

Le programme Polynôme calcule la valeur d'une fonction polynôme $p : \mathbb{Z} \rightarrow \mathbb{Z}$ de degré 2 à coefficients dans $\mathbb{Z}$:

$$p(x) = ax^2 + bx + c$$

(plus exactement $\mathbb{Z}$ restreint à l'intervalle [–16 383, +16 383] qui est celui des nombres manipulés par la pitchoun-machine).

Le texte du programme *Polynôme* est le suivant :

```
var a, b, c, x, v;
saisir a;
saisir b;
saisir c;
saisir x;
v := a * x * x + b * x + c;
afficher v.
```

Ce programme demande à l'utilisateur de saisir les valeurs des coefficients *a*, *b* et *c* ainsi que la valeur de *x* pour laquelle *p*(*x*) doit être calculée. Lançons son exécution en supposant qu'il est enregistré dans le fichier `polynome.j` :

```
j polynome.j↵

Code executable
read            (r1 contiendra la valeur saisie)
store r1, 0     (a à l'adresse 0 contiendra la valeur saisie)
read            (r1 contiendra la valeur saisie)
store r1, 1     (b à l'adresse 1 contiendra la valeur saisie)
read            (r1 contiendra la valeur saisie)
store r1, 2     (c à l'adresse 2 contiendra la valeur saisie)
read            (r1 contiendra la valeur saisie)
store r1, 3     (x à l'adresse 3 contiendra la valeur saisie)
```

```
load r1, 0        (r1 contiendra la valeur de a)
load r2, 3        (r2 contiendra la valeur de x)
mult r1, r1, r2   (r1 contiendra la valeur de a * x)
load r2, 3        (r2 contiendra la valeur de x)
mult r1, r1, r2   (r1 contiendra la valeur de a * x * x)
load r2, 1        (r2 contiendra la valeur de b)
load r3, 3        (r3 contiendra la valeur de x)
mult r2, r2, r3   (r2 contiendra la valeur de b * x)
add r1, r1, r2    (r2 contiendra la valeur de a * x * x + b * x)
load r2, 2        (r2 contiendra la valeur de c)
add r1, r1, r2    (r1 contiendra la valeur de a * x * x + b * x + c)
store r1, 4       (v à l'adresse 4 contiendra la valeur de a * x * x + b * x + c)
load r1, 4        (r1 contiendra la valeur de v)
write             (affichage de la valeur de v)
stop

Execution
? 1
? -2
? 3
? 2
= 3
```

Index