Programmation II, Langage C

Cours de Licence 2 Sciences de l'ingénieur

année 2013/2014, deuxième semestre

Faicel CHAMROUKHI

chamroukhi@univ-tln.fr

http ://chamroukhi.univ-tln.fr

Table des matières

Chapitre 1

Introduction

Avant propos

Ce cours est dispensé à des étudiants de deuxième année (L2) Sciences de L'Ingénieur (SI) qui n'ont pas de fait de programmation C avant. Les étudiants ont eu comme prérequis des enseignements en algorithmique et programmation sous Python (le langage enseigné dès la première année de la licence SI).

Cet enseignement est pour un volume horaire de 15 heurs de Cours Magistral (CM) (accompagnés de 8 heurs de Travaux Dirigés (TD) et de 27 heures de Travaux Pratiques (TP)).

Je cite en particulier le livre de Jacques Le maître et celui d'Anne Canteaut qui ont servi de base pour ce cours :

- Jacques Le Maître, *Programmer en langage C* , ISBN : 978-2-7298-7228-1, 2012, Technosup , Ellipses.
- *Programmation en langage C*, Anne Canteaut, INRIA

Quel langage pour débuter en programmation

En première année de la licence SI, nos étudiants commencent à étudier l'algorithmique et la programmation en langage **Python**. De façon générale, d'autres langages par exemple comme Caml, Pascal, Python ou C peuvent être enseigné à des étudiants débutants en programmation. Ici, sont donnés quelques arguments pour justifier le cas pour nos étudiants SI qui commencent par Python en première année et voient ensuite le langage C en deuxième année.

1. **Python** : Syntaxe beaucoup plus simple que celle de C ou du Java (proche du C), Le programmeur ne perd pas de temps en déclaration de types, de variables, etc, ce qui améliore de façon les temps de développement. Pour cela, Python est enseigné en première année SI. L'inconvénient de Python, comme pour tous les autres langages de haut niveau, est que son interprétation nécessite de nombreux autres fichiers, appelés "bibliothèques" ; moins puissant que le langage C.
2. **Caml** : *Avantages* : son interactivité, rigueur de typage et la facilité avec laquelle des données structurellement complexes peuvent être construites et manipulées.

Inconvénients pour les débutants : le côté abstrait de la programmation fonctionnelle.

3. **Pascal** : *Avantages* : syntaxe claire et rigoureuse, qui en fait un très bon langage sur le plan pédagogique. *Inconvénients* : très peu utilisé dans les applications professionnelles contrairement au langage C et dont il est proche
4. **Langage C** : Ce que l'on retient pour cet enseignement (de deuxième année) pour les raisons ci-après.

Avantages du Langage C

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

Inconvénients du langage C

- Il ne permet pas la manipulation globale des tableaux ou des chaînes de caractères. Il demande au programmeur une certaine agilité à manipuler les adresses des données : les pointeurs, ce qui n'est pas toujours évident pour les débutants.
- L'utilisation de certains opérateurs peut conduire à des programmes peu lisibles.

⇒ Cependant, on peut contourner ces défauts par une écriture rigoureuse des programmes et par l'utilisation d'options de compilation fournissant un maximum d'avertissements (e.g., `-Wall`) sur les incorrections dans le texte d'un programme.

Dans cet enseignement, Le langage C décrit est conforme à la norme ANSI dite C89.

Chapitre 2

Présentation des différents paradigmes de programmation

Contents

Le rôle d'un langage de programmation est de décrire des tâches à soumettre à un ordinateur. Il met en œuvre/implémente un algorithme donné. Il existe à l'heure actuelle plusieurs centaines de langages de programmation.

Ces langages se distinguent par

- leur *niveau*,
- leur *degré de généralité* ou *de spécialisation*
- et leur *paradigme de programmation*.

Dans la section suivante, nous allons voir la classification des langages de programmation les plus marquants.

2.1 Classification des langages de programmation les plus marquants

2.1.1 Classement par niveau

2.1.1.1 Langage de programmation bas niveau

Un langage de programmation est dit de bas niveau s'il est proche du langage machine de l'ordinateur sur lequel il est exécuté. C'est le cas par exemple, du langage assembleur.

2.1.1.2 Langage de programmation haut niveau

Inversement, un langage de programmation est dit de haut niveau s'il n'est pas lié à un ordinateur particulier et s'il permet au programmeur de décrire de façon aussi concise que possible ce que doit faire une tâche sans décrire en détail comment elle doit être exécutée.

$\Rightarrow$ Le Langage C est un langage de haut niveau (le plus bas que quasiment tous les langages de haut niveau).

Les langages d'interrogation de bases de données, par exemple, sont des langages de haut niveau (SQL $\ll$ Structured Query Language : l'utilisateur pose une question (requête) et le système de gestion de bases de données établit la stratégie la plus efficace pour y répondre. On peut citer aussi Pascal, C (de plus bas niveau que les autres), Python, C++, Java, Fortran, OCaml, Cobol

2.1.2 Classement par leur degré de généralité : Langage de programmation généraliste VS spécialisé

Un langage de programmation généraliste est conçu pour programmer tout type d'application. $\Rightarrow$Le langage C est un langage généraliste. C++ etc

Un langage de programmation spécialisé est conçu, au contraire, pour une famille d'application particulière. Par exemple, un langage pour la création de sites web (HTML, PHP, etc).

⚠ **Remarque 2.1.1.** *Les langages spécialisés sont souvent des langages de plus haut niveau que les langages généralistes.*

Le lange C est un langage de *programmation impérative*. Dans la partie suivante, nous donnons les différents paradigmes de programmation.

2.1.3 Classement par paradigme de programmation

Nous consacrons la section suivante au différents paradigmes de programmation.

2.2 Les différents paradigmes de programmation

On distingue quatre grands paradigmes de programmation

1. la programmation impérative,
2. la programmation fonctionnelle,

3. la programmation logique,
4. la programmation orientée objet.

2.3 La programmation impérative (ou procédurale)

Un programme impératif simple se présente comme une suite d'instructions adressées à l'ordinateur. Chacune des instructions a pour effet de changer l'état de la mémoire ou de communiquer avec les unités périphériques de l'ordinateur (clavier, écran ou imprimante, par exemple). La mémoire d'un programme impératif contient un ensemble de variables. Une variable a un nom et une valeur qui est rangée dans une case de la mémoire associée à cette variable. Cette valeur peut changer au cours de l'exécution du programme.

▷ **Exemple 2.3.1.** *Exemple de programme impératif (calcul et affichage de la somme de deux entiers) :*

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

Description du programme :

- *La ligne 1 contient la définition de trois variables :* `x`*,* `y` *et* `z` *dont les valeurs sont des nombres entiers.*
- *Les lignes 2, 3 et 4 contiennent chacune une instruction d'affectation qui demande l'enregistrement de la valeur de l'expression qui suit le symbole* `:=` *dans la case mémoire associée à la variable dont le nom précède ce symbole.*
- *La ligne 5 contient une instruction qui demande l'affichage à l'écran de la valeur de l'expression qui suit le mot-clé* `afficher`*.*

⇒ *L'exécution d'un programme impératif consiste à exécuter les instructions les unes après les autres, ce qui dans le cas du programme ci-dessous produira l'affichage à l'écran du nombre 7.*

⇒C, pascal, Python sont des langages de programmation impérative.

Caml se prête à la programmation dans un style fonctionnel, impératif ou orienté objets.

2.4 La programmation fonctionnelle

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

▷ **Exemple 2.4.1.** *Exemple d'une programme fonctionnel (calcul de l'aire d'un disque de rayon* `r`*) :*

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

Description du programme :

- La ligne 1 définit `pi` comme étant la fonction constante $3,14$.
- La ligne 2 définit `carre` comme étant la fonction qui appliquée à une valeur `x` retourne cette valeur multipliée par elle-même.
- La ligne 3 définit `aire_disque` comme étant la fonction qui, appliquée à une valeur `r` retourne la valeur de la multiplication de `pi` par le carré de la valeur de `r`.

$\Rightarrow$ L'exécution d'un programme fonctionnel consiste à poser une question sous la forme d'une application de fonction. La réponse est la valeur retournée par cette application.

Par exemple, pour connaître l'aire d'un disque de rayon 5, l'utilisateur posera la question :

```
aire_disque(5);
```

La réponse sera `78,5` car :

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

$\Rightarrow$ Par exemple Ocaml est un langage de programmation fonctionnelle.

2.5 La programmation logique

La programmation logique est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes.

▷ **Exemple 2.5.1.** *Exemple de programme logique :*

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


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

Description du programme :

- Les lignes 1 à 6 contiennent des relations qui sont des faits : Alain est le père de Jean, Nicole est la mère de Jean, etc.
- La ligne 7 contient une relation qui est une règle stipulant que `x` est un parent de `y` si `x` est le père de `y`.
- La ligne 8 contient une relation qui est une règle stipulant que `x` est un parent de `y` si `x` est la mère de `y`.
- La ligne 9 contient une relation qui est une règle stipulant que `x` est un grand-parent de `z` s'il existe un `y` tel que `x` est le parent de `y` et `y` est le parent de `z`.

⇒ L'exécution d'un programme logique est lancée en posant une question sous la forme d'une relation à vérifier.

Par exemple, pour connaître les grands-parents de François, l'utilisateur posera la question :

```
grand-parent(x, "François");
```

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

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

Par exemple, Prolog est un langage de programmation logique.

2.6 La programmation orientée objet

Un programme orienté objet est caractérisé par la façon dont sont représentées les données : un ensemble de *classes* dont les instances sont des *objets*.

Notion de classe en programmation orientée objet Une classe définit les propriétés des objets de cette classe et les opérations que l'on peut leur appliquer.

▷ **Exemple 2.6.1.** *Par exemple, la classe* `Personne` *possédant les propriétés* `nom` *et* `annee_de_naissance` *et l'opération âge définie par* `age(p) = annee courante - annee_de_naissance(p)` *où* `p` *est un objet (une instance) de la classe* `Personne`.

Notion d'objet en programmation orientée objet Un objet est défini par un *identificateur unique et invariable* et par les valeurs de ses propriétés qui, elles, sont variables.

Héritage Une classe peut être fille d'une autre classe. Dans ce cas, les objets de la classe fille « héritent » de toutes les propriétés et de toutes les opérations de la classe mère.

▷ **Exemple 2.6.2.** *Par exemple, si la classe* `Etudiant` *est une sous-classe de la classe* `Personne` *alors, un étudiant est aussi une personne qui a un* `nom` *et une* `année de naissance` *et dont l'*`âge` *peut être calculé par l'opération* `age` *définie dans la classe* `Personne`. *Un étudiant pourra avoir par ailleurs des propriétés supplémentaires telle que* `numero_etudiant`, *etc.*

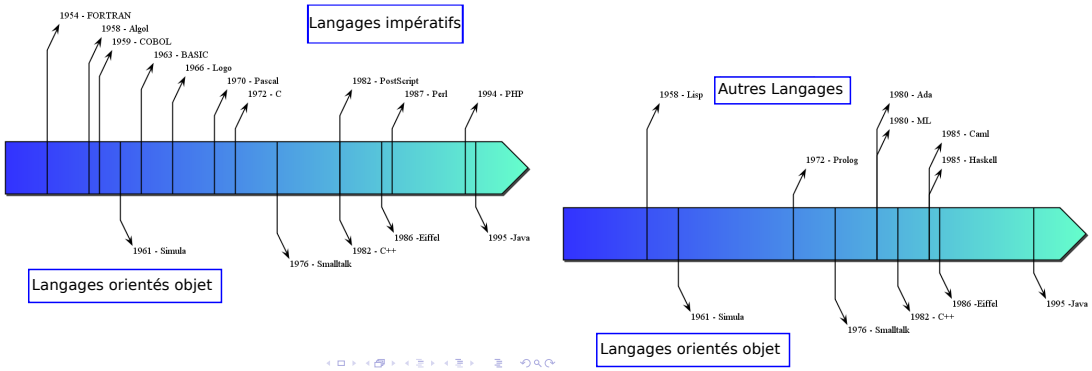
⇒ En programmation orientée objet, la manipulation des données peut, elle, être réalisée selon le paradigme impératif ou fonctionnel.

⚠ **Remarque 2.6.1.** *Les langages de programmation utilisent en général les principes de plusieurs de ces quatre paradigmes. Par exemple, la notion de fonctions, notamment, est présente dans pratiquement tous les langages de programmation.*

Résumé : Le Langage C est un langage de haut niveau (de plus bas que quasiment tous les langages de haut niveau), généraliste et impératif, ç-a-d conforme au paradigme de la programmation impérative.

historique : Le langage C a été créé au début des années 1970 par des chercheurs du laboratoire de la compagnie Bell aux USA. Il a été conçu, à l'origine, pour être le langage de programmation du système d'exploitation UNIX. C est un langage très largement diffusé. Il a de plus servi de base au langage orienté objet C++ et au langage Java très utilisé notamment dans les applications web.

2.7 Chronologie des langages de programmation les plus marquants



Chapitre 3

Structure générale et compilation d'un programme C

Contents

Ici nous allons commencé par voir comment est structuré un programme C simple. Ensuite nous verrons quelles sont les différentes étapes par lesquelles ont doit passer pour exécuter un tel programme : de la conception, en passant par la compilation et à l'exécution.

3.1 Structure générale d'un programme C

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

Un programme C est composé d'un ensemble de fonctions. Ces fonctions s'appellent les unes les autres en se transmettant des valeurs.

La définition d'une fonction spécifie son nom, le type des valeurs qui peuvent lui être transmises, le type des valeurs ou de la valeur qu'elle retourne et son corps. Le corps d'une fonction est une suite d'instructions dont l'exécution, retourne une valeur à la fonction appelante et éventuellement produit des effets de bord tels que des entrées/sorties.

⚠ **Remarque 3.1.1.** *une fonction est dite à effet de bord si elle modifie un état autre que sa valeur de retour. Par exemple, une fonction peut modifier une variable statique*

ou globale, modifier un ou plusieurs de ses arguments, écrire des données vers un écran ou un fichier ou lire des données provenant d'autres fonctions à effet de bord.

Les effets de bord rendent souvent le comportement des programmes plus difficile à comprendre et nuisent à la réutilisabilité des fonctions et procédures.

Tout programme C doit comporter au moins une fonction : la fonction `main` (fonction principale). L'exécution d'un programme C commence par l'appel de cette fonction.

Un programme C peut appeler des fonctions prédéfinies telles que les fonctions qui permettent de lire ou d'écrire des données ou les fonctions mathématiques usuelles. Ces fonctions constituent la bibliothèque standard de C. (par exemple `stdio.h` `stdlib.h` `math.h`)

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

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

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

3.1.1 Exemple d'un programme C monofichier

Nous présenterons et dériverons un un programme simple écrit en langage C, pour avoir une vue d'ensemble du langage C.

▷ **Exemple 3.1.1.** *Le programme suivant que nous appellerons* `Maximum` *est écrit en C. Il calcule le plus grand des nombres entiers (25 et 30) et affiche ce nombre à l'écran.*

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

(2) #include <stdio.h>

(3) int max(int x, int y)
(4) {
(5) if (x > y)
(6) return x;
(7) else
(8) return y;
(9) }

(10) int main(void)
(11) {
(12) int m;
(13) m = max(25, 30);
(14) printf("Le maximum de 25 et 30 est %d.\n", m);
(15) return 0;
(16) }
```

Ce programme est construit de la façon suivante :

– La ligne 1 contient un commentaire qui indique ce qu'effectue le programme.

- La ligne 2 indique que le programme appellera une fonction de la bibliothèque standard `stdio.h` : la fonction `printf` pour afficher la valeur du maximum à l'écran (ligne 14).
- Les lignes 3 à 9 constituent la définition de la fonction `max`. La ligne 3 est l'entête de cette fonction. Elle indique que cette fonction : retourne un nombre entier (type `int`), s'appelle `max`, et est appelée avec deux nombres entiers qui seront affectées aux variables `x` et `y` de type int (nombre entier).
- Le corps de la fonction `max` est le bloc qui commence à la ligne 4 et se termine à la ligne 9.
  L'instruction `if` qui commence à la ligne 5 et se termine à la ligne 8 demande la comparaison des valeurs des variables `x` et `y`. Si la valeur de `x` est supérieure à celle de `y`, alors (instruction return de la ligne 8) la valeur de `x` sera retournée à la fonction qui a appelé la fonction `max`. Sinon (ligne 7), c'est la valeur de `y` qui sera retournée (instruction return de la ligne 8).
- Les lignes 10 à 16 constituent la définition de la fonction principale `main`. Le corps de cette fonction est le bloc qui commence à la ligne 11 et se termine à la ligne 16.
- La ligne 12 contient la définition d'une variable `m` de type `int` (nombre entier) à laquelle sera affecté le maximum calculé (pour stocker le résultat).
- L'instruction de la ligne 13 appelle la fonction `max` pour calculer le maximum des nombres 25 et 30 qui lui sont transmis. La valeur retournée par cet appel sera affectée à la variable `m` (opérateur =).
- L'instruction de la ligne 14 appelle la fonction `printf` pour afficher à l'écran la chaîne de caractères : "Le maximum de 25 et 30 est *valeur de la variable m.*".
- L'instruction de la ligne 15 demande que la valeur 0 soit retournée au système d'exploitation pour lui indiquer que le programme s'est bien déroulé.

3.1.2 Exécution de ce programme

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

1. La fonction main est appelée par le système d'exploitation.
2. La fonction max est appelée avec les valeurs 25 et 30 (ligne 13) qui sont affectées respectivement aux variables `x` et `y` de la fonction max.
3. La condition `x > y` est testée (ligne 5) : elle est fausse car la valeur 25 de `x` n'est pas supérieure à la valeur 30 de `y` ; l'instruction (ligne 8) introduite par le mot-clé `else` (ligne 7) est donc exécutée et la valeur 30 est retournée à la fonction `main`.
4. La valeur 30 est affectée à la variable `m` (ligne 13).
5. La chaîne de caractères : "Le maximum de 25 et 30 est 30." est affichée (ligne 14).
6. L'exécution de l'instruction `return 0;` (ligne 15) rend la main au système d'exploitation en lui indiquant (retour de la valeur 0) que le programme s'est bien déroulé.

3.2 Cycle de vie d'un programme

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

1. **Conception du programme.** Analyser l'application à programmer pour identifier les données à manipuler et les opérations à réaliser, choisir le ou les algorithmes pour réaliser ces opérations, décider d'une présentation des résultats adaptée aux utilisateurs du programme, etc.

   C'est une étape cruciale mais difficile lorsque l'application est complexe.

2. **Écriture du programme.** Il s'agit de traduire le résultat de la conception (algorithme) en un programme écrit dans le langage de programmation choisi. C'est l'objectif de ce cours où en utilisera le Langage C.

3. **Compilation.** Un programme *"compilé"*, est un programme traduit à partir du langage de programmation en langage machine en une suite d'instructions exécutables par un ordinateur. En effet, pour qu'un programme puisse être exécuté par un ordinateur, il faut le traduire dans le langage machine de cet ordinateur. C'est le rôle du compilateur, qui

   (i) lit le programme,

   (ii) vérifie s'il est lexicalement, syntaxiquement et sémantiquement correct et
   - si c'est le cas, le traduit en langage machine produisant le code exécutable du programme.
   - si le programme n'est pas correct, le compilateur signale les erreurs en les localisant aussi précisément que possible dans le texte du programme. Il faut alors retourner à l'étape 2 (écriture du programme).

4. **Exécution.** Le code exécutable produit à l'issue de la compilation, est soumis au système d'exploitation qui l'exécute.

   Dans la plupart des cas, un programme est fait pour être exécuté plusieurs fois. Tant qu'aucune modification n'a été apportée à ce programme, il n'est pas nécessaire de le recompiler avant chaque exécution. On conserve le code exécutable et on le soumet au système d'exploitation chaque fois que l'on a besoin. Il se peut, que l'exécution du programme déclenche des erreurs non détectées à la compilation ou ne satisfasse pas ses utilisateurs, à cause d'une mauvaise écriture ou d'une mauvaise conception du programme. Il faut alors retourner à l'étape 2 ou à l'étape 1 (écriture ou conception).

3.2.1 Compilation et exécution d'un programme C monofichier

Nous verrons dans le chapitre dédiée à la compilation séparée, comment écrire un programme C complet dont le texte consiste en plusieurs fichiers comportant des fonctions (`.c`) et/ou des fichiers d'entête `.h`, etc.

Ici nous montrons la marche à suivre pour compiler et exécuter un programme dont le texte est enregistré dans un fichier unique. La compilation que l'on montre ici s'effectue dans un environnement Unix et avec le compilateur GCC,

1. Saisir le texte du programme sous un éditeur de texte et le sauvegarder dans un fichier ayant l'extension `.c`, par exemple `prog.c`, du répertoire courant (`prog` est le nom du programme).

2. Compiler le programme en lançant la commande :

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

   Si le compilateur détecte des erreurs, elles seront affichées et il faudra les corriger. S'il n'y pas d'erreurs, le fichier exécutable `prog` est créé.

L'option facultative `-Wall` est recommandée car elle génère une liste très complète d'avertissements ("warnings") sur des incorrections éventuelles ou des oublis dans le texte du programme qui pourraient provoquer des problèmes lors de l'exécution.

L'option facultative `-o prog` permet d'indiquer le nom du programme éxécutable à construire (ici `prog`) après la compilation du programme source `prog.c`

3. Exécuter le programme en lançant la commande :

   ```
   ./prog
   ```

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

⚠ **Remarque 3.2.1.** *Dans le cas où l'option* `-o prog` *n'est pas présente, le fichier exécutable généré aura pour nom* `a.out` *et l'exécution du programme sera lancée par la commande* `./a.out`.

▷ **Exemple 3.2.1.** *Par exemple, si le programme est le programme* `Maximum` *de l'exemple 3.1.1 dont le texte est enregistré dans le fichier source* `maximum.c`*, sa compilation sera lancée par la commande :*

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

et son exécution par la commande :

```
./maximum
```

Chapitre 4

Représentation des données en mémoire

Contents

4.1 Types numériques, constantes et variables

4.1.1 Données en langage C

Un programme C manipule des données. Une donnée a un *type* et une *valeur*. Elle peut avoir un *nom* et elle peut être *constante* ou *variable*.

La valeur d'une donnée variable, ou plus simplement d'une *variable*, peut changer au cours de l'exécution d'un programme alors que celle d'une donnée *constante*, ou plus simplement d'une constante, ne le peut pas.

4.1.1.1 Donnée variable (variable)

Une variable est une case de la mémoire qui est identifiée par son *adresse* et qui contient la *valeur* de cette variable.

▷ **Exemple 4.1.1.** *Supposons, par exemple, que l'on veuille écrire un programme qui calcule l'aire d'un disque ($\pi\, r^2$) dont le rayon est saisi au clavier de son poste de travail par l'utilisateur du programme. Le nombre π est une constante. Le rayon et l'aire du disque sont des variables. Il y aura donc dans la mémoire une case nommée* `rayon` *qui contiendra le rayon du disque une fois qu'il aura été saisi par l'utilisateur et une case nommée* `aire` *qui contiendra l'aire de ce disque une fois qu'elle aura été calculée.*

4.1.1.2 Types de variables

Une valeur peut être :

1. un nombre *entier* ;
2. un nombre *flottant* (une approximation d'un nombre réel) ;
3. une *structure* composée d'un ensemble variables (appelés champs de la structure) qui peuvent être de types différents (par exemple, une structure représentant un point d'un plan, composée du nom, de l'abscisse et de l'ordonnée de ce point) ;
4. une *union* qui est une variable dont les valeurs peuvent être de types différents (mais en général de même taille). (par exemple, une union dont la valeur pourra être un nombre entier ou un nombre flottant ou un caractère et un entier, etc) ;

   Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.
5. l'*adresse* d'une variable ou d'une fonction (les fameux pointeurs).

⚠ **Remarque 4.1.1.** *Les valeurs sont classées par types. Toute valeur a un et un seul type.*

Tableaux Il est de plus possible de définir des tableaux qui, en C, ne sont pas des variables mais des suites de variables de même type rangées de façon contigüe dans la mémoire.

▷ **Exemple 4.1.2.** *Par exemple, le tableau des nombres d'étudiants des classes de Licence à la fac.*

⚠ **Remarque 4.1.2.** *Il n'existe pas de type spécifique pour les booléens, les caractères et les chaînes de caractères.* Le booléen « faux » est représenté par le nombre zéro et le booléen « vrai » par toute valeur différente de zéro (par exemple 1).

Un caractère est codé par un nombre entier (conformément au code ASCII1, en général). Une chaîne de caractères est représentée comme un tableau de caractères (c.-à-d. comme un tableau de nombres entiers).

Les différents types sont désignés par les mots-clefs suivants :

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

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

Nous expliquerons ensuite, comment est organisée la mémoire d'un programme et nous proposerons une représentation graphique des variables contenues dans cette mémoire .

4.2 Représentation des nombres

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

4.2.1 Représentation des nombres entiers

Si l'on dispose d'une taille de n bits, on peut représenter tous les entiers positifs ou nuls ($\in \mathbb{N}$) compris entre 0 et $2^n - 1$ ou bien tous les entiers négatifs, positifs ou nul compris entre $-2^{n-1} + 1$ et $2^{n-1} - 1$ ($\in \mathbb{Z}$).

4.2.2 Représentation des nombres réels

Par contre, pour les nombres réels c'est plus complexe, car il faut tenir compte de la partie entière et de la partie décimale qui peut être de longueur infinie.

Si le nombre de bits dont on dispose pour représenter un nombre réel est fini, on ne pourra pas représenter tous les nombres réels qui appartiennent à un intervalle donné. Dans un ordinateur, un nombre réel est donc représenté par le nombre décimal qui en est le plus proche et qui constitue donc une *approximation de ce nombre réel*.

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

4.2.2.1 Représentation en virgule fixe

Dans la représentation en virgule fixe, on fixe le nombre de chiffres après la virgule.

Un nombre décimal est représenté par un nombre entier n. Si b est la base et k le nombre de chiffres après la virgule, le nombre décimal N représenté par l'entier n sera

$$N = n \times b^{-k}.$$

▷ **Exemple 4.2.1.** *Par exemple, si* $k = 3$*, en base 10 le nombre* $17,648$ *sera représenté par le nombre* 17648 *(on a* $17648 = 17,648 \times 10^{-3}$*). Le problème posé par cette représentation est que la partie décimale devra être tronquée si le nombre de chiffres*

après la virgule est supérieur à k. Par exemple, le nombre $6,4817$ *sera représenté par* 6481 *et donc tronqué car* $6481 \times 10^{-3} = 6,481$.

Remarquons que bien que les nombres $6,4817$ et $17,648$ aient le même nombre de chiffres significatifs ($k = 3$), il a fallu tronquer le premier mais pas le second. Pour résoudre ce problème, une autre représentation a été proposée : la représentation en virgule flottante qui consiste à introduire *un exposant*.

4.2.2.2 Représentation en virgule flottante

La représentation d'un nombre décimal en virgule flottante est un doublet (m, e) où m est la mantisse et e est l'exposant. Si b est la base, le nombre N représenté par le doublet (m, e) sera égal à

$$N = m \times b^e.$$

▷ **Exemple 4.2.2.** *Par exemple, si* $b = 10$*, le nombre* $17,648$ *sera représenté par*

$$17,648 = m \times 10^e \text{ avec } (m; e) = (17648; -3)$$

et le nombre $6,4817$ *sera représenté par*

$$6,4817 = m \times 10^e \text{ avec } (m; e) = (64817; -4).$$

⇒ C'est l'exposant qui détermine la place de la virgule : d'où le nom de cette représentation.

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

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

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

4.3 Types numériques

On distingue les types numériques *entiers* et les types numériques *flottants*.

4.3.1 Les types entiers

Les types entiers sont

- le type `char`

– les types entiers *signés* : `signed char`, `short int`, `int`, `long int` et les types entiers *non signés* : `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`.

On notera que les noms des types entiers ont plusieurs synonymes (la paire de crochets autour d'un mot indique que ce mot est facultatif). Par exemple, `short` (le plus employé), `signed short` et `signed short` int sont des synonymes de `short int`. Le type `char` est destiné à la manipulation des caractères du jeu de caractères de la machine utilisée (en général, ceux du code ASCII ou d'un code ASCII étendu).

Le code ASCII de base prend en compte tous les caractères nécessaires à l'écriture de l'anglais : il leur attribue un code numérique de 0 à 127 (7 bits). Les codes ASCII étendus profitent du 8e bit pour attribuer un code numérique aux caractères des langues européennes non pris en compte par le code ASCII de base, les caractères accentués entre autres.

⚠ **Remarque 4.3.1.** *Attention ! Dans ce cours, nous n'utiliserons que les caractères du code ASCII de base dans les textes des programmes et dans les fichiers lus par ces programmes. Les caractères accentués ne seront donc pas utilisés.*

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

⚠ **Remarque 4.3.2.** ⇒ *L'intérêt des types non signés est de permettre la représentation dans le même espace mémoire de nombres positifs ou nuls deux fois plus grands que s'ils pouvaient aussi être négatifs.*

Le tableau 4.1 présente les types entiers du langage C.

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

TABLE 4.1 – Types numériques entiers du Langage C.

4.3.2 Les types flottants

Les types flottants sont : `float`, `double` et `long double`. Ils sont destinés à la représentation des nombres en *virgule flottante* simple précision, double précision et double précision étendue.

Le tableau 4.2 présente les types entiers du langage C.

La représentation d'un nombre flottant est caractérisée par son signe, sa mantisse et son exposant :

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

TABLE 4.2 – Types numériques flottants du Langage C.

```
Nombre = signe x mantisse x 2^exposant
```

Notez que l'extension et la précision de ces types sont dépendantes de chaque implantation (architecture de la mémoire de la machine utilisé), mais les valeurs minimales indiquées dans les tableaux 4.1 et 4.2 ainsi que le respect des contraintes suivantes sont garanties par la norme ANSI :

1. extension(`signed char`) $\subseteq$ extension(`signed short int`) $\subseteq$ extension(`int`)$\subseteq$ extension(`signed long int`)
2. extension(`float`) $\subseteq$ extension(`double`) $\subseteq$ extension(`long double`)
3. précision(`float`) $\leq$ précision(`double`) $\leq$ précision(`long double`)
4. extension(`signed type`) $\subseteq$ extension(`unsigned type`) où `type` $\in$ {`char, short int, int, long int`}
5. La place mémoire occupée par un entier de type `signed type` est la même que celle occupée par un entier de type `unsigned type` et la représentation d'un même entier dans chacun de ces deux types est la même.

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

4.3.3 Constantes symboliques INT_MIN et INT_MAX

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

⚠ **Remarque 4.3.3.** *Attention ! Pour assurer la portabilité d'un programme, il faudra choisir un type numérique par rapport à son extension et à sa précision minimales et non par rapport à celles de ce type dans l'implantation de C sur laquelle est développé le programme. Notamment, il ne faut utiliser le type* `char` *que pour la manipulation des caractères et le type* `int` *que pour des entiers compris entre -32 767 à 32 767.*

4.4 Identificateurs

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

identificateur est une suite contiguë de caractères dont chacun peut être une lettre, un chiffre ou le caractère _ (blanc souligné). Par convention, les identificateurs commençant par un blanc souligné sont réservés à la bibliothèque standard. Tous les identificateurs d'un programme utilisateur devront donc commencer par une lettre. Un identificateur peut avoir de 1 à 31 caractères. Une lettre majuscule est un caractère différent de la même lettre en minuscule. Le premier caractère d'un identificateur ne peut pas être un chiffre.

▷ **Exemple 4.4.1.** *Par exemple :* `x var1 y1 ma_variable tab_23` *sont des identificateurs valides. par contre* `1i` *et* `i:j` *ne le sont pas.*

4.4.1 Les mots-clefs du Langage C

Les mots suivants dits ≪ mots réservés ≫ qui jouent un rôle spécifique dans la syntaxe de C, *ne peuvent pas être utilisés comme identificateurs*. Le langage C en norme ANSI compte 32 mots clefs

```
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
```

que l'on peut ranger en catégories

- les spécificateurs de stockage
  `auto register static extern typedef`
- les spécificateurs de type
  `char double enum float int long short signed struct union unsigned void`
- les qualificateurs de type
  `const volatile`
- les instructions de contrôle
  `break case continue default do else for goto if switch while`
- divers
  `return sizeof`

4.5 Constantes

Les valeurs sont des abstractions, mais il faut pouvoir les représenter (les écrire) dans le texte d'un programme.

Dans un langage de programmation, on appelle constante la représentation d'une valeur. En C, on distingue :

- les constantes littérales, (entiers, réels, caractères)
- les constantes symboliques
- et les expressions constantes.

4.5.1 Constantes littérales

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

4.5.1.1 Constantes entières

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

Par exemple : `0 234112` sont des constantes littérales de type entier qui représentent respectivement les entiers 0 et 234112.

Une constante littérale de type entier est la représentation d'une valeur du plus petit des types `int, long, unsigned long` qui contient cette valeur.

Par exemple, si le type `int` est codé sur 2 octets, les constantes littérales 17 et 12519 représentent des valeurs de type `int`.

4.5.1.2 Constantes réelles (flottants)

Une constante littérale de type flottant est formée :
- d'une partie entière : une suite de chiffres ;
- suivie d'un point ;
- suivi d'une partie fractionnaire : une suite de chiffres ;
- suivie d'un exposant : la lettre `e` ou `E`, suivie du signe + ou −, suivi d'une suite de chiffres.

On peut omettre :
- la partie entière ou la partie fractionnaire, mais pas les deux ;
- le point ou l'exposant, mais pas les deux.

Par exemple :

```
5341.3e-1 .17 2e3
```

sont des constantes littérales qui représentent respectivement les nombres : $534,13$ $0,17$ et 2000.

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

4.5.1.3 Constantes caractères

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

```
'A', '9', ';' '$'
```

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

⚠ **Remarque 4.5.1.** *Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par* `\\` *et* `\'`.

⇒ Les caractères non imprimables peuvent être désignes par `'\code-octal'` où `code-octal` est le code en octal du caractère. On peut aussi écrire `'\xcode-hexa'` où `code-hexa` est le code en hexadécimal du caractère.

Par exemple, `'\33'` et `'\x1b'` désignent le caractère escape. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

Entre autres :

\n nouvelle ligne	\r retour chariot
\t tabulation horizontale	\f saut de page
\v tabulation verticale	\a signal d'alerte
\0 caractère nul	\' apostrophe
\\ anti-slash	\" guillemet \? point d'interrogation

4.5.1.4 Chaîne de caractères

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

```
"Bonjour"  "ceci est une chaîne de caractères"
```

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

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

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple, `"Ceci est une chaîne de caractères"` Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple, `"ligne 1 \n ligne 2"` A l'intérieur d'une chaîne de caractères, le caractère " doit être désigné par `\"`. Enfin, le caractère \ suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple,

```
"ceci est une trèèèèèèèèèèèèèèèèès longue longue longue \
chaîne de caractères"
```

4.5.2 Constantes symboliques

Une constante symbolique est un nom donné à une valeur. Il y a deux façons de donner un nom à une valeur : les énumérations et les définitions de macros : Par exemple :

```
#define PI 3.14
```

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

4.5.3 Expressions constantes

Une expression constante est une expression construite à partir de constantes et d'opérateurs (les opérateurs arithmétiques, par exemple).

Une expression constante ne contient pas de variables. Elle a donc pour particularité de ne pas dépendre de l'exécution du programme. Sa valeur peut donc être calculée lors de la compilation du programme.

Par exemple, si `M` est une constante symbolique qui représente l'entier 20 et `N` est une constante symbolique qui représente l'entier 30, alors : `N + M + 1` est une expression constante qui a pour valeur l'entier 51.

4.6 Variables

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

```
type var1 = exp1,.., varn = expn;
```

où :
- `type` est l'un des types numériques des tableaux 4.1 et 4.2 ;
- `var1`, ... , `varn` sont les noms des variables définies (des identificateurs) ;
- `exp1`, ... , `expn` sont des expressions dont les valeurs constituent les valeurs initiales respectives des variables définies.

Cette définition **déclare** n variables de type `type`, nommées `var1`,..., `varn`, et les **crée**.

1. La **déclaration** d'une variable consiste à lui associer un type
2. la création d'une variable consiste à lui attribuer une case de la mémoire et à y enregistrer sa valeur initiale (**initialisation**), si elle est indiquée.

L'indication d'une valeur initiale est facultative. L'expression d'une valeur initiale ainsi que la valeur initiale affectée à une variable lorsque cette valeur n'est pas spécifiée dans la définition de cette variable, diffèrent selon que cette variable est **globale** ou **locale** comme nous le verrons plus tard.

Par exemple, les définitions :

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

définissent respectivement :
- trois variables `i`, `j` et `k` de type `int` ;
- deux variables `longueur` et `largeur` de type `float` et de valeurs initiales respectives 10,5 et 6,34 ;
- une variable `une_lettre` de type `char` intialisée au caractère 'a'.

⚠ **Remarque 4.6.1.** *On peut indiquer au compilateur qu'une variable ne changera pas de valeur au cours de l'exécution d'un programme, en préfixant sa définition par le mot-clé* `const`. *En ce cas, sa valeur devra être fournie dans la définition de cette variable. Cette indication pourra être utilisée par C pour contrôler que le programme ne modifie pas cette valeur.* Par exemple, la définition :

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

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

4.7 Mémoire d'un programme

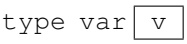
La mémoire d'un programme contient les informations dont il a besoin pour son exécution. Elle est constituée de 4 parties :

1. le code exécutable du programme qui est constitué de la suite des instructions machine à exécuter. Il n'est pas modifié lors de l'exécution du programme.

2. la mémoire statique qui contient les *variables globales*.
3. la pile, qui contient les *variables locales* (celles qui sont définies dans le corps des fonctions). La pile évolue par l'ajout d'une variable à son sommet ou par la suppression de la variable placée à son sommet.
4. le tas, qui contient des variables allouées dynamiquement au cours de l'exécution du programme. Lors de l'exécution d'un programme le tas évolue par l'allocation d'une variable dans un emplacement disponible au moment de cette allocation, ou par la suppression d'une variable allouée dynamiquement et la restitution de l'emplacement qu'elle occupait.

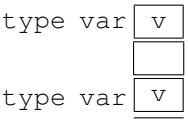
Dans ce cours, nous utiliserons la visualisation graphique suivante des variables et de la pile.

– une variable var, de type T et de valeur v sera représentée par : où le rectangle



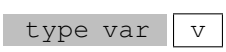
représente la case mémoire dans laquelle est placée la valeur de la variable (le type pourra être omis) ;

– un extrait de la pile ou de la mémoire statique apparaitra comme une suite de variables rangées dans l'ordre de leur déclaration, de bas en haut ou éventuellement de gauche à droite dans le cas des tableaux et le fond de la pile sera marqué par un double trait :



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

– une variable `var`, de type `type` et de valeur `v` sera représentée par :

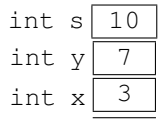


▷ **Exemple 4.7.1.** *Considérons, par exemple, le programme C suivant qui définit trois variables x, y et s (ligne 3), demande à l'utilisateur de saisir la valeur des variables x et y (lignes 4 et 5), affecte à la variable s la somme des valeurs de x et de y (ligne 6) et enfin affiche la valeur de s (ligne 7).*

```
(1) int main(void)
(2) {
(3) int x, y, s;
(4) scanf("%d", &x); /*saisir la valeur de x;*/
(5) scanf("%d", &y); /*saisir la valeur de y;*/
(6) s = x + y;
(7) printf("somme : %d\n",s);/*afficher la valeur de s;*/
```

```
(8) return 0;
(9) }
```

Quand le bloc qui constitue le corps de la fonction main est exécuté (lignes 2 à 8) les variables qui y sont définies (x, y et s dans cet exemple) sont empilées dans l'ordre présentée dans la figure ci-après. Quand la valeur de x a été saisie par l'utilisateur, elle est lue et affectée dans la case associée à x. Il en est de même pour y. Enfin, la valeur de l'expression `x + y` *est calculée et rangée dans la case associée à s. L'état de la pile immédiatement après l'exécution de l'instruction 6 sera le suivant, si les valeurs saisies pour x et y sont 3 et 7 :*



Chapitre 5

Expressions et opérateurs

Contents

5.1 Expressions

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

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

sont des expressions qui ont pour valeurs respectives 12, 8, 17 et 1 (≪ vrai ≫). En C, toute expression a obligatoirement un type et une valeur. Elle peut avoir une adresse et un effet de bord qui change l ?état de la mémoire ou déclenche des entrées-sorties.

Soit `expr` une expression, nous noterons :
- `type(expr)` son type ;
- `val(expr)` sa valeur ;
- `adr(expr)` son adresse.

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

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

5.2 Expressions simples

Une expression simple est soit une constante, soit un nom de variable.
Une constante `cste` est une expression qui a les propriétés suivantes :
- `type(cste)` = type de la valeur représentée par `cste` ;
- `val(cste)` = valeur représentée par `cste`.

Par exemple :
- 125 est une expression de type int qui a pour valeur le nombre 125 ;
- 3.14 est une expression de type double qui a pour valeur le nombre 3,14.

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

⚠ **Remarque 5.2.1.** *Attention ! Un même nom peut être associé à des constantes ou à des variables différentes dans un même programme. La conséquence est que lorsqu'un nom apparaît dans une expression, il faut pouvoir déterminer de façon univoque quelle est la valeur ou la variable associée à ce nom. Il ne peut pas y avoir deux entités (type, constante ou variable) de même nom qui sont visibles en un même point du texte d'un programme. Les règles de visibilité des déclarations se résument de la façon suivante : un nom dans une expression se rapporte à sa déclaration la plus récente dans l'ordre de lecture du programme.*

5.3 Opérateurs

On distingue

1. l'affectation.
2. les opérateurs arithmétiques ; opérateurs relationnels (de comparaison)
3. les opérateurs logiques booléens ;
4. la conversion de type ;
5. Les opérateurs d'affectation composée
6. Les opérateurs d'incrémentation et de décrémentation
7. L'opérateur virgule

8. L'opérateur conditionnel ternaire
9. L'opérateur d'appel de fonction
10. L'opérateur adresse

L'opérateur d'appel de fonction et les opérateurs de manipulation de structures de tableaux et des adresses seront étudiés plus tard dans des chapitres qui suivront.

Les opérateurs arithmétiques ou de comparaison peuvent s'appliquer à des opérandes de types différents, mais pour que l'opération puisse être effectuée, il faut au préalable convertir ces opérandes en un même type que nous appellerons le type commun.

Intuitivement, il est clair que le type commun doit être le plus grand, au sens de son extension et de sa précision, des types des opérandes. Mais le fait que C impose que ce type ne soit pas plus petit que le type `int` et le fait que le type `int` oscille entre le type `short` et le type `long` selon les implantations, rend un peu plus compliqué le choix du type commun. Ce choix s'appuie sur l'ordre suivant des types numériques :

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

5.3.1 Opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont

(i) l'opérateur moins unaire − (changement de signe) qui calcule l'opposé d'un nombre et on a

```
type(-expr) = type(expr)
val(-expr) = -val(expr)
```

(ii) ainsi que les opérateurs *binaires*

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste de la division de deux nombres entiers (modulo)

TABLE 5.1 – Opérateurs arithmétiques binaire.

• Si `expr1` et `expr2` sont des expressions de type numérique, et `op` est l'un des opérateurs `+ - * / %` alors :
- `type(expr1 op expr2)` = type commun de `type(exp1)` et de `type(exp2)`
- `valeur(expr1 op expr2) = val1 op val2`, où `val1` et `val2` sont le résultat des conversions de `valeur(expr1)` et de `valeur(expr2)` en type commun

• Si `expr1` et `expr2` sont de type entier, l'opérateur `/` effectue la division entière de leurs valeurs.

• Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'elévation à la puissance. Il y a cependant la fonction `pow` de la librairie `math.h` telle que `pow(x,y)` calcule x^y.

▷ **Exemple 5.3.1.** *Par exemple*
- *l'expression* `((4.0 + 2.25) * 2.0)` *est de type* `double` *(flottant) et a la valeur* $12,5$ *;*

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

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

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

a la valeur $5,5$.

5.3.2 Comparateurs

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

`==`	égal
`!=`	différence
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal

TABLE 5.2 – Opérateurs de comparaison.

La syntaxe est

```
expr1 op expr2
```

Les deux expressions sont évaluées puis comparées. Rappelons qu'il n'y a pas de type booléen en C. La valeur rendue est de type `int`. En C, le booléen « faux » est représenté par 0 et le booléen « vrai » par toute valeur différente de 0.

Si `expr1` et `expr2` sont des expressions de type numérique et `op` est l'un des opérateurs de comparaison `==, !=, <, <=, > ou >=`, alors :

```
expr1 op expr2
```

sont des expressions telles que :

(i) `type(expr1 op expr2) = int`

(ii) `valeur(expr1 op expr2) = val1 op val2`, où `val1` et `val2` sont le résultat de la conversion de `valeur(expr1)` et de `valeur(expr2)` en type commun

(iii) `val1 op val2` est égal à 1 si la comparaison est vraie, 0 sinon

&&	et logique
||	ou logique
!	négation logique

TABLE 5.3 – Opérateurs logiques booléens.

5.3.3 Les opérateurs logiques booléens

Sont :

5.3.3.1 Négation

L'opérateur `!` calcule la négation d'un booléen. Si `expr` est une expression, alors :

```
!exp
```

est une expression qui a les propriétés suivantes :

(i) `type(exp) = int`

(ii) `valeur(!exp) = 1` si `valeur(expr) = 0` et `0` sinon.

5.3.3.2 Conjonction et disjonction

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

Si `expr1` et `expr2` sont des expressions et `op` est l'un des opérateurs || et &&, alors :

```
expr1 op expr2
```

sont des expressions telles que :

(i) `type(expr1 op expr2) = int` et

(ii) `valeur(expr1||expr2)` =
si `valeur(expr1)` $\neq 0$ alors 1
sinon,
si `valeur(expr2)` $\neq 0$ alors 1 sinon 0 fin-si
fin-si

(iii) `valeur(expr1 && expr2)` =
si `valeur(expr1)` = 0 alors 0
sinon,
si `valeur(expr2)` $\neq 0$ alors 1 sinon 0 fin-si
fin-si

⚠ **Remarque 5.3.1.** *L'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Le deuxième opérande n'est donc pas évalué si le premier opérande a la valeur 1 dans le cas de l'opérateur || ou la valeur 0 dans le cas de l'opérateur &&.*

▷ **Exemple 5.3.2.** *Par exemple dans*

```
int i;
int T[10];
if ((i >= 0) && (i <= 9) && !(T[i] == 0))
...
```

la dernière condition ne sera pas évaluée si `i` *n'est pas entre 0 et 9.*

5.3.4 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (`short`, `int` ou `long`), signés ou non. Les opérateurs `&`, `|` et `~` consistent à appliquer bit à bit les opérations suivantes

`&` et	`|` ou inclusif
`^` ou exclusif	`~` complément à 1
<< décalage à gauche	>> décalage à droite

`&`	0	1
0	0	0
1	0	1

`|`	0	1
0	0	1
1	1	1

`^`	0	1
0	0	1
1	1	0

L'opérateur unaire `~` change la valeur de chaque bit d'un entier.

Le décalage à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2 égale à 2^n où n est le nombre de bit décalés. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît).

5.3.5 Changement de type (Conversion)

L'opérateur de conversion de type, `(T)`, appelé *cast*, où `T` est un nom de type, convertit une valeur en une valeur de type `T`. Il permet de modifier explicitement le type d'une valeur. On écrit

```
(T) expr
```

où `(T)` est un nom de type et `epxr` est une expression dont la valeur est convertible en `(T)` et on a :

(i) `type((T) expr) = T`

(ii) `valeur((T) expr) = valeur(expr)` convertie en `T`

Par exemple,

```
main()
{
int i = 3, j = 2;
printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

⚠ **Remarque 5.3.2.** *Attention ! Il n'est pas possible de changer le type d'une structure ou d'une union.*

5.3.6 Affectation

L'opérateur = affecte une valeur à une variable. Sa syntaxe est la suivante

```
variable = expression
```

L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche.

Si `expr1` est une valeur gauche et `expr2` est une expression, alors :

```
expr1 = expr2
```

est une expression qui a les propriétés suivantes :

- `type(expr1 = expr2) = type(expr1)`
- `val(expr1 = expr2) = val(expr2)` convertie en `type(expr1)`
- `adr(expr1 = expr2) = adr(expr1)`,
- elle a un effet de bord : `val(expr2)` est enregistrée dans la case mémoire qui contient la valeur de `expr1` et remplace cette valeur.

▷ **Exemple 5.3.3.** *Par exemple le programme*

```
int main()
{
int i, j = 2;
float x = 2.5;
i = j + x;
x = x + i;
printf("\n %f \n",x);
return 0;
}
```

imprime pour `x` *la valeur 6.5 (et non 7), car dans l'instruction* `i = j + x;`*, l'expression* `j + x` *a été convertie en entier.*

⚠ **Remarque 5.3.3.** *Attention ! Ne faut pas confondre l'opérateur d'affectation = avec l'opérateur d'égalité ==.*

Signalons les abréviations classiques, obtenues avec les opérateurs

```
++ −− += −= *= /= %=        &= ^= |= <<= >>=
```

5.3.6.1 Les opérateurs d'incrémentation et de décrémentation

Soit `expr` une expression numérique :

1. l'expression : `exp++` a le même type que `expr`, le même effet de bord que `expr = expr + 1` et a pour valeur `valeur(expr)` ;
2. l'expression : `++expr` a le même type que `exp`, le même effet de bord que `exp = exp + 1` et a pour valeur `valeur(expr) + 1`.

▷ **Exemple 5.3.4.** *Par exemple, si* `x` *a la valeur 5 :*

1. *après l'évaluation de l'expression* `y = x++`*,* `x` *aura la valeur 6 et* `y` *aura la valeur 5 ;*
2. *après l'évaluation de l'expression* `y = ++x`*,* `x` *aura la valeur 6 et* `y` *aura la valeur 6.*

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

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

5.3.6.2 Les opérateurs d'affectation composée

Soit `expr1` une expression qui est une valeur gauche et `expr2` une expression, l'expression :

```
expr1 op= expr2
```

où `op` est l'un des opérateurs `+ - * / %  &= ^= |= <<= >>=` est équivalente à

```
expr1 = expr1 op expr2
```

▷ **Exemple 5.3.5.** *Par exemple, si* `x` *a la valeur 5, après l'évaluation de l'expression* `x += 7`, `x` *aura la valeur 12.*

5.3.7 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

```
expr1, expr2, ... , exprn
```

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme

```
int main()
{
int a, b;
b = ((a = 4), (a + 1));
printf("\n b = %d \n",b);
return 0;
}
```

affiche `b = 5`.

5.3.8 L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. L'opérateur ternaire accepte trois arguments Sa syntaxe est la suivante :

```
condition ? expr1 : expr2
```

Cette expression est égale à `expr1` si condition est satisfaite, et à `expr2` sinon.

▷ **Exemple 5.3.6.** *Par exemple, l'expression*

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre.

5.3.9 L'opérateur adresse

L'opérateur d'adresse `&` appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est

```
& expression
```

On le verra plus en détails dans le chapitre dédié aux pointeurs.

5.4 Priorité et associativité des opérateurs

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

Le tableau de la figure 3.1 indique la priorité (par ordre décroissant) et l'associativité des opérateurs du C :

Opérateurs	Associativité
`() [] . ->`	de gauche à droite
`! ~ ++ --` −(unaire) ⋆(unaire) `(type) sizeof &`(adresse)	de gauche à droite
`* / %`	de droite à gauche
`+` −(binaire)	de gauche à droite
`<< >>`	de gauche à droite
`< <= > >=`	de gauche à droite
`== !=`	de gauche à droite
`&`(et bit à bit)	de gauche à droite
`^` (ou exclusif)	de gauche à droite
`|` (ou bit à bit)	de gauche à droite
`?:`	de droite à gauche
`&&`	de gauche à droite
`||`	de gauche à droite
`= += -= *= /= %= &= ^= |= <<= >>=`	de droite à gauche
`,`	de gauche à droite

TABLE 5.4 – Priorité et associativité des principaux opérateurs de C

Les règles suivantes appliquées à une expression parcourue de gauche à droite, permettent de rétablir les paires de parenthèses rendues implicites par l ?application des règles de priorité et d ?associativité. Inversement, elles permettent de supprimer les paires de parenthèses rendues inutiles par l ?application de ces mêmes règles. Si `e`, `f` et `g` sont des expressions simples ou parenthésées et si `op1` et `op2` sont des opérateurs infixes (placés entre leurs opérandes), on a :

- `e op1 f op2 g` ≡ `(e op1 f) op2 g`, si la priorité de `op1` est supérieure à la priorité de `op2` ou si la priorité de `op1` est égale à la priorité de `op2` et que `op1` est associatif de gauche à droite
- `e op1 f op2 g` ≡ `e op1 (f op2 g)`, sinon

Si `e` et `f` sont des expressions simples ou parenthésées, si `op1` est un opérateur préfixe (placé devant son opérande) et si `op2` est un opérateur infixe, on a :

- `op1 e op2 e` ≡ `(op1 e) op2 f`, si la priorité de `op1` est supérieure à la priorité de `op2` ou bien si la priorité de `op1` est égale à la priorité de `op2` et que `op1` est associatif de gauche à droite
- `op1 e op2 f` ≡ `op1 (e op2 f)`, sinon

Par exemple :

(i) `6 * 3 + 4` ≡ `(6 * 3) + 4`, car la priorité de ⋆ est supérieure à celle de +

(ii) `6 + 3 * 4` ≡ `6 + (3 * 4)`, car la priorité de + n'est pas supérieure à celle de ⋆

(iii) `!a && b || x > 3`

≡`(!a) && b|| x > 3`, car la priorité de `!` est supérieure à celle de `&&`

≡`((!a) && b)|| x > 3`, car la priorité de `&&` est supérieure à celle de `||`

≡ `((!a) && b)|| (x > 3)`, car la priorité de `||` n'est pas supérieure à celle de >

(iv) `3 + 4 - 5` ≡ `(3 + 4) - 5`, car + et − ont la même priorité et + est associatif de gauche à droite

(v) `x = y = 5` ≡ `x = (y = 5)`, car = est associatif de droite à gauche

⚠ **Remarque 5.4.1.** *En cas de doute, il est plus sûr de spécifier l'ordre des opérations d'une expression en la parenthésant. Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions. Par exemple, il faut écrire* `if ((x ^ y) != 0)`.

5.5 Saisie de données et affichage de résultats

Les fonctions d'entrées-sorties seront étudiées dans le chapitre dédié aux entrées-sorties que nous verrons plus tard (Chapitre 9). En attendant, et afin de pouvoir écrire des programmes qui lisent des données saisies au clavier ou affichent des résultats à l'écran, on fera appel aux fonctions `printf` et `scanf` définies dans la bibliothèque standard de C `<stdio.h>`.

La fonction `printf` écrit des caractères sur l'unité de sortie standard : l'écran.

La fonction `scanf` lit des caractères saisis au clavier (l'unité d'entrée standard) par l'utilisateur.

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

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

soit placée en tête du programme.

5.6 *printf*

L'affichage des valeurs d'une suite d'expressions `expr1`, ..., `exprn` au sein d'une chaîne de caractères qui les nomme ou les commente, est obtenue par l'appel :

```
printf("...%conv1...%convn...", expr1, ..., exprn);
```

Les valeurs des expressions `expr1`, ..., `exprn` sont converties en une suite de caractères selon les spécifications de conversion `%conv1`, ..., `%convn` puis substituées à ces spécifications dans la chaîne de caractères "...`%conv1`...`%convn`...". La chaîne de caractères obtenue est affichée.

5.7 *scanf*

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

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

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

Les spécifications de conversion les plus classiques sont :

- `%c` pour convertir un caractère en un nombre de type `char` ou inversement,
- `%d` pour convertir une suite de caractères en un nombre de type `int` ou inversement
- `%f` pour convertir une suite de caractères en un nombre de type `float` ou inversement.

▷ **Exemple 5.7.1.** *Par exemple :*

- si `x` vaut 5 et `y` vaut 9, `printf("%d + %d = %d\n", x, y, x + y)` affiche "5 + 9 = 14" puis passera à la ligne ;
- printf("Entrer une note ? ") afficher « Entrer une note » ;
- `scanf("%f", &note)` aura pour effet de bord de lire le nombre flottant saisi au clavier et de l'affecter à la variable note.

Chapitre 6

Instructions

Contents

6.1 Instructions

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

En programmation impérative, un algorithme se traduit par une suite d'instructions. Les instructions offertes par C sont les suivantes :

- instruction vide ;
- instruction ≪ expression ≫ ;
- bloc d'instructions ;
- instruction conditionnelle `if---else` ;
- instruction de choix multiple `switch` ;
- instruction d'itération (*boucles* `while`, `do--while`, `for` ;
- instruction de rupture de séquence `break` ;

- instruction de branchement non conditionnel `continue` ;
- `goto` ;
- instruction de retour d'un appel de fonction `return`.
- .

Dans ce chapitre, pour chaque instruction, nous décrirons sa syntaxe et l'action qu'elle effectue.

6.2 Instruction vide

Une instruction vide a la forme suivante :

```
;
```

Elle ne réalise aucune action.

6.3 Instruction « expression »

Une instruction « expression » a la forme suivante :

```
expr;
```

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

```
x = 3 + 5;
```

est une instruction qui déclenche l'action : « affecter 8 à x ». Par contre, l'instruction :

```
3 + 5;
```

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

6.4 Bloc d'instructions

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

```
{
decl1
...
declm
inst1
...
instn
}
```

où `decl1,...,` `declm` sont des déclarations de type, de variables ou de tableaux et `inst1,...,` `instn` sont des instructions.

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

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

La durée de vie d'une variable locale à un bloc est donc celle de l'exécution de ce bloc.

Variables globales Les variables qui sont utilisées dans un bloc et qui n'y sont pas définies doivent avoir été définies dans une autre partie du programme, soit globalement à l'extérieur des fonctions du programme, soit dans un bloc englobant. Ces variables sont dites globales.

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

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

6.5 Instruction conditionnelle $if---else$

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

En C, l'instruction conditionnelle est l'instruction `if`. Sa formulation la plus simple est la suivante :

```
if (expression)
   instruction
```

Elle peut aussi avoir l'une des deux formes suivantes :

```
if (expression)
   instruction1
else
   instruction2
```

ou

```
if (expression1)
   instruction1
else if (expression2)
{
   instruction2
  .
  .
  .
  }
```

```
else if (expressionn)
   instructionn
else
   instruction
```

avec un nombre quelconque de `else if`. Le dernier `else` est toujours facultatif. Chaque instruction peut être un bloc d'instructions.

6.6 Instruction de choix multiple : *switch*

Une instruction de choix multiple permet de choisir une suite d'instructions à réaliser parmi un ensemble de suites d'instructions possibles, en fonction de la valeur d'une expression que nous appellerons « expression de choix ». En C, l'instruction de choix multiple est l'instruction `switch`. Sa forme la plus générale est la suivante :

```
switch (expression)
  {
  case constante1:
     instructions1
     break;
  case constante-2:
     instructions2
     break;
   ...
  case constanten:
     instructionsn
     break;
  default:
     instructions
     break;
}
```

Si la valeur de expression est égale à l'une des constantes, la liste d'instructions correspondant est exécutée. Sinon la liste d'`instructions` correspondant à default est exécutée. L'instruction `break` permet d'éviter d'enchaîner l'exécution de deux cas exclusifs consécutifs, et elle a donc pour effet d'abandonner l'exécution de l'instruction `switch` et d'exécuter l'instruction qui la suit (`break`) immédiatement.
L'instruction `default` est facultative.

▷ **Exemple 6.6.1.** *Soit* `n` *la valeur d'un nombre entier, le bloc suivant affiche si ce nombre est pair ou impair :*

```
switch(n % 2)
   {
   case 0:
      printf("%d est pair !", n);
      break;
   case 1:
      printf("%d est impair !", n);
      break;
}
```

Par exemple, si la valeur de n est 3, « 3 est impair ! » sera affiché.

⚠ **Remarque 6.6.1.** *Attention ! Dans une expression* `switch`*, un cas doit être associé à chacune des valeurs que peut prendre l'expression de choix. Lorsque parmi ces valeurs, il en existe qui ne correspondent pas à une opération valide, alors, on leur associera un cas de traitement d'erreur : le cas default en général.*

6.7 Instructions d'itération (boucles)

Une instruction d'itération, permet de répéter l'exécution d'une instruction tant qu'une condition est vérifiée. Trois instructions d'itération sont disponibles en C : `while`, `do while` et `for`.

6.7.1 Boucle *while*

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

`while` (*expression*)
instruction

où *expression* est une expression booléenne : le test de continuation et *instruction* est une instruction à répéter : le corps de la boucle. *instruction* est exécutée tant que la condition est vérifiée (i.e., *expression* est non nulle),

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

▷ **Exemple 6.7.1.** *Programme qui affiche les entiers de 1 à 9.*

```
i = 1;
while (i < 10)
   {
     printf("\n i = %d",i);
     i++;
    }
```

6.7.2 Boucle *do − −while*

Si l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do--while`. Sa syntaxe est

`do`
instruction
`while`(*expression*);

où, comme dans l'instruction `while`, *instruction* est le corps de la boucle et *expression* est le test de continuation.

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

▷ **Exemple 6.7.2.** *Par exemple, pour saisir au clavier un entier entre 1 et 10 :*

```
int a;
do
 {
   printf("\n Entrez un entier entre 1 et 10 : ");
   scanf("%d",&a);
 }
while ((a <= 0) || (a > 10));
```

6.7.3 Boucle *for*

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

`for` (*expr1* `;` *expr2* `;` *exp3*)
instruction

où *expr1*, *expr2*, *expr3* sont des expressions et *instruction* est une instruction.
Elle est équivalente, par définition, à :

expr1`;`
`while` (*expr2*)
`{`
instruction
expr3`;`
`}`

▷ **Exemple 6.7.3.** *Par exemple, pour afficher tous les entiers de 0 à 9, on écrit :*

```
for (i = 0; i < 10; i++)
  printf("\n i = %d",i);
```

⚠ **Remarque 6.7.1.** *Dans une instruction* `for` (*expr1* `;` *expr2* `;` *exp3*)*, les expressions expr1 et expr3 peuvent être absentes. Le test de continuation (expr2) peut lui aussi être absent, on considère alors qu'il est toujours vrai. Par exemple, l'instruction :*

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

6.7.4 Choisir entre *for* et *while*

Les instructions `for` et `while` sont équivalentes. Le choix entre les deux peut être guidé par des considérations de lisibilité :

- Dans le cas où le nombre d'itérations dépend du calcul fait dans le corps de la boucle, on utilisera les instructions `while` ou `do while` plutôt que l'instruction `for`.

– Dans le cas où le nombre d'itérations dépend d'un paramètre dont les valeurs initiale et finale ainsi que l'incrément sont connus avant l'exécution de la boucle, on utilisera plutôt l'instruction `for`.

▷ **Exemple 6.7.4.** *Par exemple, pour le calcul de la somme des carrés des entiers de 1 à n*

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

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

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

plutôt que l'instruction `while` *:*

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

6.8 Rupture de séquence : $break$

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

6.9 Branchement non conditionnel $continue$

L'instruction `continue` permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      if (i == 3)
        continue;
      printf("i = %d\n",i);
    }
```

```
    printf("valeur de i a la sortie de la boucle = %d\n"→
        ↪ ,i);
}
```

affiche les valeurs 0, 1, 2, 4.

6.10 Retour d'un appel de fonction

Les instructions :

`return;`
`return` *expression*`;`

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

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

Chapitre 7

Fonctions et programmes

Contents

Tout langage de programmation offre le moyen de découper un programme en unités qui peuvent s'appeler les unes les autres en se transmettant des valeurs et qui peuvent partager les mêmes variables. En C, ces unités sont les fonctions et les variables partagées sont appelées *variables globales*, par opposition aux *variables locales* qui ne sont utilisables que dans les blocs où elles sont définies.

Par convention, l'exécution d'un programme commence par l'appel de la fonction `main`, qui doit donc être présente dans tout programme C : c'est elle qui pilote l'exécution du programme.

7.1 Fonctions

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

Une fonction peut être appeler par elle-même, dans ce dernier cas, on dit que la fonction est *récursive*).

Dans ce chapitre, nous étudierons :

– la définition et l'appel des fonctions
– les règles de structuration d'un programme composé de plusieurs fichiers compilés ou non

7.2 Définition d'une fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme

type_fonction nom_fonction (type_1 arg_1 , ... , type_n arg_n)
{
[*déclarations de variables locales*]
instructions
}

où

- La première ligne de cette définition est l'en-tête de la fonction (on dit aussi son prototype). Cet en-tête déclare le nom de la fonction et le type de la fonction *type_fonction*, c'est-à-dire le type de la valeur qu'elle retourne.
- *type_fonction* est le type de la fonction ;
- si la fonction ne possède pas de valeur de retour, on remplace le type de valeur de retour par le mot clef `void`
- *nom_fonction* est le nom de la fonction : un identificateur ;
- Les arguments de la fonction *arg_1,..., arg_n* sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction.
- si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.
- Enfin, le bloc *instructions* constitue le corps de la fonction.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'instruction de retour à la fonction appelante, `return`. Dans le corps de la fonction on doit trouver au moins une instruction :

`return` *expression* ;

dont l'exécution a pour effet de retourner à la fonction appelante la valeur `valeur(expression)` appelée *valeur de retour*.

Si la fonction ne retourne pas de valeur (fonction de type `void`), sa définition se termine par

```
return;
```

⚠ **Remarque 7.2.1.** *Lorsqu'une instruction* `return;` *est la dernière du corps d'une fonction, elle peut être omise. Si la fonction ne retourne pas de valeur (fonction de type* `void`*), l'instruction* `return` *peut donc être omise.*

▷ **Exemple 7.2.1.** *Fonction puissance. La fonction qui calcule x^n où x est un réel et n un entier positif ou nul, peut être définie de la façon suivante :*

```
1  float puissance(float x, int n)
2      {
3      float p;
4      int i;
5      if (n < 0)
6          {
```

```
7           printf("Appel de puissance incorrect !");
8           exit(1);
9           }
10      p = 1;
11      for (i = 1; i <= n; i++)
12          p = p * x;
13      return p;
14  }
```

La ligne 1 constitue l'en-tête de la fonction et le bloc d'instructions 2-14, son corps. Cette fonction a pour nom puissance, pour arguments formels les variables `x` *et* `n` *de types respectifs* `float` *et* `int` *et pour type de retour* `float`.

7.2.1 Appel d'une fonction

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

nom_fonction (*param_1*, ..., *param_n*)
où *param_1*,..., *param_n* sont des expressions dont les valeurs constituent les arguments effectifs de la fonction.

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur virgule.

▷ **Exemple 7.2.2.** *Appel de la fonction puissance vu précédemment.*

```
{
float y;
x = 3;
m = puissance(3, 2);
}
```

7.2.2 Fonctions sans arguments ou sans valeur de retour

Une fonction peut ne pas avoir d'arguments. Elle est alors de la forme :

type_fonction nom_fonction (`void`)
{
    [*déclarations de variables locales*]
    instructions
}
où `void` est le type vide. Par exemple, la fonction suivante lit un entier saisi au clavier et le retourne

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

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

`void` *nom_fonction* `(` *param_1*`, ..., ` *param_n*`)`
`{`
[*déclarations de variables locales*]
instructions
`}`

où `void` est le type vide. Dans ce cas, les instructions `return` présentes dans le corps de cette fonction ne spécifieront pas d'expression de la valeur de retour. Elles seront de la forme `return;`

Remarque : Une fonction sans valeur de retour n'a de sens que si son appel produit un effet de bord. C'est le cas, par exemple, de la fonction suivante qui affiche un entier à l'écran :

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

La définition de la fonction *ecrire_entier* aurait pu aussi s'écrire :

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

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

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

7.2.3 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale `main`. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée.

Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction `main`), elle doit impérativement être déclarée au préalable.

Une fonction secondaire est déclarée par son prototype, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

type_fonction *nom_fonction* `(` *type_1*`, ..., ` *type_n*`);`

Les fonctions secondaires peuvent être déclarées et définies avant la fonction `main` ou déclarés avant et définies après la fonction `main`. Par exemple, on écrira

```
int puissance (int, int );
int puissance (int a, int n)
  {
```

```
    if (n == 0)
    return(1);
    return(a * puissance(a, n-1));
   }

int main()
 {
    int a = 2, b = 5;
    printf("%d\n", puissance(a,b));
    returun 0;
 }
```

On peut également

⚠ **Remarque 7.2.2.** *La déclaration est parfois facultative, par exemple quand les fonctions sont définies avant la fonction* `main` *et dans le bon ordre. Cependant, la déclaration est importante car elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le prototype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.*

7.2.3.1 Durée de vie des variables

La durée de vie d'une variable locale à un bloc est celle de l'exécution de ce bloc. Les variables qui sont utilisées dans un bloc et qui n'y sont pas définies doivent avoir été définies dans une autre partie du programme, soit globalement à l'extérieur des fonctions du programme, soit dans un bloc englobant.

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

La fonction `main` est un peu particulière. Elle est appelée par le système d'exploitation pour lancer l'exécution du programme et elle lui retourne un code d'erreur (un entier) indiquant si le programme s'est bien déroulé ou non. Elle ne peut être appelée ni par elle-même, ni dans le corps d'une autre fonction du programme.
La fonction `main` retourne un entier de type `int` et peut avoir des arguments qui lui sont transmis par le système d'exploitation lors du lancement du programme. Lorsqu'elle n'a pas d'arguments son en-tête doit être :

```
int main(void)
```

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

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

Si le programme est lancé par la commande :

```
nom_commande arg_1  arg_2 ... arg_k
```

où

- `nom_commande` est le nom de cette commande et `arg_1 arg_2 ... arg_k` sont ses arguments ;
- la variable `argc` (argument counter) est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle a la valeur `k` + 1,
- la variable `argv` (argument vector) est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. `argv[0]` pointera sur la chaîne de caractères `nom_commande`, la variable `argv[1]` sur la chaîne de caractères `arg_1` et la variable `argv[k]` sur la chaîne de caractères `arg_k`.

Tout cela sera plus clair lorsque nous aurons étudié les tableaux, les pointeurs et les chaînes de caractères.

L'entier retourné par la fonction `main` est, par convention, égal à 0 si le programme s'est déroulé correctement, différent de 0 sinon (1, en général). En conséquence, l'appel de la fonction `main` doit se terminer par l'exécution de l'instruction

`return` *expression*`;`

où *expression est une expression qui a pour valeur le code d'erreur.*

⚠ **Remarque 7.2.3.** *La fonction* `main` *retourne un entier. Considérer que la fonction* `main()` *est de type* `void` *est toléré par le compilateur. Toutefois l'écriture* `main()` *provoque un message d'avertissement lorsqu'on utilise l'option* `-Wall` *de* `gcc`.

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

`void exit(int` *statut*`)`

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

Pour la fonction `main` *on peut s'affranchir de la fonction* `exit` *en utilisant les constantes symboliques* `EXIT_SUCCESS` *(le programme s'est déroulé correctement) ou* `EXIT_FAILURE` *(une erreur s'est produite) définies dans le fichier d'en-têtes* `stdlib.h`.

▷ **Exemple 7.2.3.** *Par exemple, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :*

```
#include <stdio.h> /* printf*/
#include <stdlib.h> /* atoi, EXIT_SUCCESS, EXIT_FILURE →
    ↪ */
int main(int argc, char *argv[])
{
  int a, b;
  if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
```

```
      return(EXIT_FAILURE);
    }
    /* atoi convertit un char en int */
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLe produit de %d par %d vaut : %d\n", a, b, →
      ↪ a * b);
  return(EXIT_SUCCESS);
}
```

Supposons que ce programme soit enregistré dans le fichier `produit-arguments.c` *et compilé par la commande :*

```
gcc -Wall produit-arguments -o produit
```

l'exécution de la commande

```
./produit 12 3
```

affichera :

```
Le produit de 12 par 3 vaut : 36
```

Chapitre 8

Programmation modulaire et compilation séparée

Contents

Dans ce chapitre, nous étudierons la programmation modulaire, la compilation séparée et l'utilisation du compilateur GCC et de l'outil `make` *pour assembler les différents fichiers d'un programme et générer son code exécutable.*

8.1 Structure d'un programme modulaire et compilation séparée

8.1.1 Programmation modulaire

Les fonctions d'un programme C peuvent être réparties dans plusieurs fichiers. Chacun de ces fichiers peut être vu comme un module qui prend en charge une des composantes de l'application gérée par le programme ou bien comme une bibliothèque qui

rassemble des fonctions d'intérêt général (des fonctions mathématiques, par exemple) qui pourront être utilisées soit par les autres modules du programme, soit par d'autres programmes.

8.1.2 Compilation séparée

De plus, ces fichiers peuvent être compilés séparément, ce qui permet lors de la modification d'un programme de ne recompiler que les fichiers touchés par cette modification.

8.1.3 Fichiers sources

Le texte d'un programme C est contenu dans un ou plusieurs fichiers dits **fichiers sources**

Pour les programmes simples, tels que ceux qui sont proposés dans ce cours à titre d'exercices, un seul fichier source est suffisant. Dès qu'un programme devient volumineux, il faut le découper en plusieurs fichiers sources que l'on peut voir comme des modules. Cela a plusieurs avantages :

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

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

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

Les fonctions définies dans un fichier source peuvent utiliser des variables globales ou des fonctions définies dans d'autres fichiers sources ainsi que des fonctions d'utilité générale dont le code objet est enregistré dans une bibliothèque dont notamment la bibliothèque standard.

8.1.4 Fichiers d'en-têtes

Les éléments d'un fichier source doivent être ordonnés conformément au principe suivant : chaque fois que le compilateur lit un nom dans le texte d'un fichier source, il doit avoir connaissance de l'entité (type, constante ou variable) désignée par ce nom et de son type. Cette entité doit donc avoir été déclarée.

Lorsque les fonctions d'un fichier source utilisent des entités définies dans d'autres fichiers sources ou dans une bibliothèque, il est nécessaire de déclarer ces entités au début de ce fichier.

La pratique est de regrouper les déclarations d'entités utilisables dans plusieurs fichiers sources dans des fichiers dits : **fichiers d'en-têtes**.

$\Rightarrow$ *Le contenu d'un fichier d'en-têtes devra être inclus dans tout fichier source utilisant une entité déclarée dans ce fichier d'en-têtes.*

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

8.1.5 Génération du code exécutable d'un programme modulaire

La génération du code exécutable d'un programme se déroule en deux phases :

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

8.1.6 Directives au préprocesseur

Le préprocesseur est un outil bien pratique.

Le préprocesseur est un programme exécuté lors de la première phase de la compilation.

Il effectue des modifications textuelles sur le fichier source à partir de directives (actions).

Les différentes directives au préprocesseur, introduites par le caractère `#`*, ont pour but :*

- *l'inclusion de fichiers (*`#include`*),*
- *la définition de constantes symboliques et de macros (*`#define`*),*
- *la compilation conditionnelle (*`#if`*,* `#ifndef`*,...).*

8.1.6.1 Substitution de macros

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

`#define` *NOM texte*

par exemple `#define PI 3.14`

On adopte souvent la convention d'écrire les noms de macro en majuscules pour les distinguer des noms de variables ou de fonctions que l'on écrit en général en minuscules. Après avoir lu cette directive, le préprocesseur substituera le texte texte à chaque occurrence de l'identificateur `nom`*. Cette substitution ne s'applique pas à l'intérieur d'une chaîne de caractères, d'un commentaire ou d'un identificateur*

8.1.6.2 Inclusion de fichiers d'en-têtes

Après avoir lu la directive :

`#include` *nom de fichier*

Le préprocesseur écrit dans le fichier à compiler le texte contenu dans le fichier dont le nom est spécifié.
Cette directive est principalement utilisée pour inclure dans un fichier source le contenu d'un fichier d'en-têtes. Elle a alors l'une des deux formes suivantes :

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

qui est à utiliser pour l'inclusion de fichiers d'en-têtes associés à la bibliothèque standard. Sinon, il faut utiliser la seconde forme suivante

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

Le nom de fichier fourni doit être un nom complet précisant le chemin d'accès sauf si ce fichier réside dans le répertoire courant, auquel cas son nom module suffit.

8.1.7 Règles de visibilité

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

1. *les noms de variables, de tableaux et de fonctions, les noms de valeurs introduites par un type énuméré (*`enum`*) et les noms de type introduits par une déclaration* `typedef`*;*

   Un type énuméré est un type dont on énumère, c'est-à-dire dont on donne la liste effective, les éléments de ce type. Cela ne peut concerner que des types correspondants à un ensemble fini d'éléments.
2. *les noms d'énumération, de structure et d'union ;*
3. *les noms des champs de chaque type structure ou de chaque type union.*

(Les tableaux, structures et unions seront étudiées au chapitre 10.)

En C les règles de visibilité sont les suivantes :

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

Le fichier source suivant, illustre certaines de ces règles :

```
1 float x = 5.5, y = 9.1;
2 float min(float x, float y)
3 {
4 if (x < y)
5   return x;
6 else
7   return y;
8 }
9  int main(void)
```

```
10 {
11   float m;
12   m = min(x, y);
13   return 0;
14 }
```

Les variables `x` *et* `y` *définies à la ligne 1 sont des variables globales. Elles sont visibles dans le corps de la fonction* `main` *(bloc 9-14) mais pas dans celui de la fonction* `min` *(bloc 2-8) car elles sont masquées par les arguments formels* `x` *et* `y` *qui sont assimilés à des variables locales au bloc 3-8. Ces deux variables sont visibles dans tout ce bloc. La variable* `m` *définie à la ligne 11 est locale au bloc 9-14, elle est visible dans tout ce bloc. Après l'exécution de l'instruction 12, la variable* `m` *aura la valeur 5,5*

8.1.8 Déclarations externes

La déclaration d'une variable, d'un tableau ou d'une fonction peut être séparée de sa définition. Une telle déclaration est appelée déclaration externe.

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

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

```
extern int nombre;
```

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

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

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

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

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

Les déclarations externes de fonctions peuvent aussi s'avérer nécessaire, dans le fichier source dans lequel elles sont définies, selon l'imbrication de leurs appels.

8.1.9 La bibliothèque standard

La bibliothèque standard contient les codes objet des fonctions utilitaires distribués avec chaque implantation de C : fonctions d'entrées-sorties, fonctions mathématiques, fonctions de manipulation de chaînes de caractères, etc. Associée à cette bibliothèque, un ensemble de fichiers d'en-têtes est également distribué. Ces fichiers contiennent les déclarations des fonctions de la bibliothèque standard, des déclarations de types ou de constantes associés à ces fonctions, etc. Les fichiers d'en-têtes suivants contiennent les déclarations de fonctions de la bibliothèque standard et de définitions de constantes :

1. **ctype.h** *contient les déclarations des fonctions qui testent à quelle catégorie appartient un caractère : alphabétique, numérique, alphanumérique, etc. et celles des fonctions de conversion d'une lettre en minuscule ou en majuscule ;*
   - **classification de caractères**
     - `int isupper(int C)` *: retourne une valeur différente de zéro, si C est une majuscule*
     - `int islower(int C)` *: retourne une valeur différente de zéro, si C est une minuscule*
     - `int isdigit(int C)` *: retourne une valeur différente de zéro, si C est un chiffre décimal*
     - `int isalpha(int C)` *: retourne une valeur différente de zéro, si islower(C) ou isupper(C)*
     - `int isalnum(int C)` *: retourne une valeur différente de zéro, si isalpha(C) ou isdigit(C)*
     - `int isxdigit(int C)` *: retourne une valeur différente de zéro, si C est un chiffre hexadécimal*
     - `int isspace(int C)` *: retourne une valeur différente de zéro, si C est un signe d'espacement*
   - **Conversion de caractères**
     Elles fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de C reste inchangée :
     - `int tolower(int C)` *: retourne C converti en minuscule si C est une majuscule, sinon C*
     - `int toupper(int C)` *: retourne C converti en majuscule si C est une minuscule, sinon C*
2. **float.h** *contient le plus petit et le plus grand nombre représentables ainsi que le nombre de chiffres significatifs pour chaque type numérique flottant de C ;*
3. **limits.h** *contient notamment les valeurs minimale et maximale des instances de chaque type numérique entier de C ;*
4. **math.h** *contient les déclarations des principales fonctions mathématiques : exponentielles, logarithmiques, trigonométriques, hyperboliques, etc. ;*
5. **stddef.h** *contient notamment la définition de la macro* `NULL` *: un pointeur qui ne pointe sur rien ;*
6. **stdio.h** *contient les déclarations des fonctions d'entrées-sorties dont les fonctions* `printf` *et* `scanf` *;*
7. **stdlib.h** *contient les déclarations de diverses fonctions utilitaires : conversions, allocation dynamique de mémoire, sortie d'un programme dont la fonction* `exit`*, génération de nombres aléatoires, tri, recherche dichotomique, etc. ;*
8. **string.h** *contient les déclarations des fonctions de manipulation des chaînes de caractères (*`atoi`*,* `strncmp`*,* `strlen`*, etc).*

8.2 Écriture d'un programme modulaire

L'écriture d'un programme `progm_aire_disque` *qui calcule et affiche l'aire d'un disque de rayon* `r` *saisi au clavier :* $aire\_disque = \pi r^2$*, peut être décomposée en trois fichiers :*

1. *le fichier* `aire_disque.h` *:*

```
#ifndef AIRE_DISQUE
#define AIRE_DISQUE
#define PI 3.14
float aire_disque(float);
#endif
```

qui contient la définition de la constante symbolique `PI` *ainsi que la déclaration de la fonction* `aire_disque` *(la signification des deux premières lignes sera expliquée ci-dessous) ;*

2. *le fichier* `aire_disque.c`

```
#include <math.h> /* pow */
#include "aire_disque.h" /* déclaration de →
    ↪ aire_disque */


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

qui contient la définition de la fonction `aire_disque`*, précédée de l'inclusion des fichiers d'en-têtes* `math.h` *et* `aire_disque.h` *qui contiennent les déclarations des fonctions* `pow`*,* `aire_disque` *et la définition de la constante* `PI`*, utilisées dans les corps de la fonction* `aire_disque`*.*

3. *le fichier* `progm_aire_disque.c` *:*

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

int main(void)
{
  float r, aire;
  printf("Rayon du disque (en cm) ? ");
  scanf("%f", &r);
  aire_disque= aire_disque(r);
  printf("Aire du disque = %.0f cm2\n", aire);
  return 0;
}
```

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

⚠ **Remarque 8.2.1.** *Il se peut que l'inclusion d'un même fichier d'en-têtes soit demandée dans plusieurs fichiers sources d'un même programme. Il faut empêcher que ce fichier soit inclus plusieurs fois afin d'éviter les erreurs dues aux définitions multiples d'une même entité. Cela peut être fait en utilisant la directive de compilation conditionnelle :*

`#ifndef` *MACRO*

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

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

```
#ifndef NOM_MACRO
#define NOM_MACRO
contenu du fichier d'en-têtes
#endif
```

8.3 Génération du code exécutable d'un programme

8.3.1 Compilation des fichiers sources

La compilation d'un fichier source consiste à générer un fichier objet qui contient les codes objets des fonctions définies dans ce fichier. Le code objet d'une fonction contient du code exécutable ainsi que des appels vers des fonctions définies dans d'autres fichiers sources ou dans une bibliothèque (par exemple, les fonctions d'entrées-sorties) mais dont l'adresse en mémoire n'est pas encore connue.

La commande

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

lance la compilation du fichier source fic.c avec l'option `-Wall` *qui fournie une liste très complète d'avertissements (des erreurs ou des oublis qui, bien que n'empêchant pas la compilation, peuvent être la source d'une exécution incorrecte du programme).*

Par exemple, la commande : `gcc -Wall -c progm_aire_disque.c` *génère le fichier objet* `progm_aire_disque.o`.

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

8.3.2 Génération du code exécutable

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

Soit fic1,..., ficn les fichiers source ou objet des modules d'un programme. La commande :

`gcc` *[*`-Wall`*] fic*$_1$ *... fic*$_m$ *[*`-l`*bib*$_1$*] ... [*`-l`*bib*$_n$*]* `-o` *fic*

compile parmi les fichiers fic$_1$ *... fic*$_m$ *et avec l'option* `-Wall` *ceux qui sont des fichiers sources (.c), réalise l'édition de liens et génère, si aucune erreur n'est détectée, le fichier exécutable* `fic` *(ou* `a.out` *si l'option* `-o` *n'est pas présente).*
L'exécution du programme pourra alors être lancée par la commande `./fic` *(ou* `./a.out`*).*
Lorsqu'un programme utilise des fonctions de bibliothèque, l'éditeur de liens doit connaitre le nom du fichier qui contient les codes objet de ces fonctions.

Pour les fonctions de la bibliothèque standard, autres que les fonctions mathématiques, il n'est pas nécessaire de fournir ce nom : l'éditeur de liens le connait.

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

Le fichier de bibliothèque qui contient le code objet des fonctions mathématiques de la bibliothèque standard a pour nom lib`m`*. Si un programme fait appel à certaines des fonctions mathématiques de la bibliothèque standard, il faut ajouter l'option* `-lm` *dans la commande de génération de son code exécutable.*

Par exemple, la commande :

```
gcc -Wall aire_disque.o progm_aire_disque.c -lm -o progm_aire_disque
```

génère le fichier exécutable `progm_aire_disque` *à partir du fichier objet* `aire_disque.o` *(généré par la compilation du fichier objet* `aire_disque.c`*), du fichier source* `progm_aire_disque.c` *et de la bibliothèque mathématique, nécessaire car la fonction* `pow` *(puissance) sont utilisées dans le fichier source* `progm_aire_disque.c`

8.4 Utilitaire *make*

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

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

fic : fic_1 ... fic_n

commandes (précédées chacune d'une tabulation) et dont la signification est la suivante. Le fichier fic dépend des fichiers fic_1 ... fic_n*. Pour que le fichier fic soit à jour, il faut que les fichiers* fic_1 ... fic_n *le soient aussi. Si ce n'est pas le cas, alors les commandes spécifiées devront être exécutées pour mettre à jour le fichier fic.*

Crétation d'un *makefile*

Par exemple, pour générer le fichier exécutable `progm_aire_disque`*, on écrit*

```
progm_aire_disque : progm_aire_disque.o aire_disque.o
     gcc aire_disque.o progm_aire_disque.o -o progm_aire_disque
progm_aire_disque.o : aire_disque.h progm_aire_disque.c
     gcc -Wall -c progm_aire_disque.c
aire_disque.o : aire_disque.h aire_disque.c
     gcc -Wall -c aire_disque.c
```

Les commentaires sont précédés du caractère `#`*.*

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

`make -f` *fic*

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

`make -f` *progm_aire_disque.mk*

Chapitre 9

Entrées/Sorties

Contents

En C, les opérations d'entrées-sorties : saisie au clavier, affichage à l'écran, lecture et écriture sur disque, etc. sont réalisées par des fonctions prédéfinies de la bibliothèque standard de C qui sont déclarées dans le fichier d'en-têtes `stdio.h`.

Les fonctions d'entrées-sorties traitent les données qu'elles lisent ou écrivent comme des flots ou des fichiers d'octets qui représentent soit des caractères, soit des mots de la mémoire. Dans le premier cas les fichiers sont dits « fichiers texte » et dans le second ils sont dits « fichiers binaires ».

Par exemple, le texte source d'un programme est enregistré dans un fichier texte et son code exécutable dans un fichier binaire.

Ici nous ne traiterons que des fichiers texte.

Il existe trois fichiers particuliers toujours disponibles lors de l'exécution d'un programme : le fichier d'entrée standard qui est généralement affecté au clavier du poste de travail, le fichier de sortie standard qui est généralement affecté à l'écran du poste de travail et le fichier d'erreur standard qui est aussi généralement affecté à l'écran du poste de travail.

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

9.1 Saisie et affichage de données dans les E/S standards

9.1.1 Lecture et écriture d'un caractère

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

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

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

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

```
int getchar(void)
```

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

▷ **Exemple 9.1.1.** *Le programme suivant lit un caractère dans le fichier d'entrée standard et l'écrit dans le fichier de sortie standard tant que le caractère lu est différent du caractère « nouvelle ligne » :*

```
#include <stdio.h>

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

équivalent à

```
#include <stdio.h>

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

```
        putchar(c);
    return 0;
    }
```

9.1.2 Lecture et écriture d'une chaîne de caractères

La lecture ou l'écriture d'une chaîne de caractères est réalisée en utilisant les fonctions d'entrées-sorties :

- `getchar` *(resp.* `putchar`*) : lire (resp. écrire) une chaîne, caractère par caractère de l'entrée standard (clavier) (resp. sur la sortie standard (écran)) ;*
- `fgetc` *(resp.* `fputc`*) : lire (resp. écrire) une chaîne contenue dans (resp. dans) une ligne d'un fichier texte, caractère par caractère ;*
- `gets` *(resp.* `puts`*) pour lire (resp. écrire) une chaîne de caractères à partir du fichier d'entrée standard (clavier) (resp. dans le fichier de sortie standard (écran)) ;*
- `fgets`*, (resp.* `fputs` *pour lire (resp. écrire) une chaîne de caractères contenue dans (resp. dans) une ligne d'un fichier texte ;*
- `gets` *(resp.* `puts`*) pour lire (resp. écrire) une chaîne de caractères contenue dans (resp. dans) une ligne d'un fichier texte ;*
- `fgets` *(resp.* `fputs`*) pour lire (resp. écrire) une chaîne de caractères contenue dans (resp. dans) une ligne d'un fichier texte ;*
- `scanf` *(resp.* `printf`*) pour lire (resp. écrire) à partir de (resp. sur) l'écran une chaîne de caractères avec la spécification de conversion* `%s`*.*
- `fscanf` *(resp.* `fprintf`*) pour lire (resp. écrire) une chaîne de caractères contenue dans (resp. dans) un fichier texte*

Nous verrons l'utilisation de ces fonctions en détail ci-après.

9.1.2.1 Ecriture avec format dans le fichier de sortie standard

L'écriture avec format dans le fichier de sortie standard est effectuée par la fonction `printf` *dont l'appel a la forme suivante :*
`printf`(*format, expr1..., expn*)

Le format est une chaîne de caractères qui doit contenir autant de spécifications de conversion qu'il y a de valeurs à écrire. Les valeurs à écrire sont celles des expressions expr1,..., expn. Une spécification de conversion spécifie la façon de convertir une valeur en une suite de caractères.

La première spécification de conversion correspondant à la première valeur à écrire et ainsi de suite. Cet appel retourne le nombre de valeurs écrites si l'écriture s'est déroulée correctement ou un entier négatif sinon.

Comme exemples de spécifications de conversion offertes par C, on a :

- `%d` *pour écrire un nombre de type char, short ou int, signé ou non signé, sous forme décimale signée,*
- `%hd` *pour écrire un nombre de type short, signé ou non signé, sous forme décimale signée ;*
- `%ld` *pour écrire un nombre de type long, signé ou non signé, sous forme décimale signée ;*
- `%f` *pour écrire un nombre de type float ou double en virgule fixe ;*
- `%e` *pour écrire un nombre de type float ou double en virgule flottante ;*

- `%Lf` *pour écrire un nombre de type long double en virgule fixe ;*
- `%Le` *pour écrire un nombre de type long double en virgule flottante ;*
- `%c` *pour écrire un nombre de type char, short ou int, signé ou non signé, sous la forme d'un caractère dont le code est le résultat de la conversion de ce nombre en unsigned char ;*
- `%s` *pour écrire une chaîne de caractères rangée dans un tableau d'éléments de type char.*

9.1.2.2 Lecture avec format dans le fichier d'entrée standard

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

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

`scanf`(*format, expr1..., expn*)

l'appel de la fonction `scanf` *a souvent la forme suivante*
`scanf(format, &var1,..., &varn)`

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

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

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

Comme exemples de spécifications de conversion offertes par C

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

9.2 Lecture et écriture dans des fichiers texte

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

par tous les langages de programmation. Ils sont manipulables par un éditeur de texte et sont imprimables.

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

L'échange de données entre la mémoire d'un programme et un fichier texte se fait au travers d'un tampon (« buffer » en anglais) qui est une zone de la mémoire qui contient une copie d'un morceau du fichier.

Quand un programme lit ou écrit une suite de caractères, il le fait dans le tampon. Parallèlement, un processus est mis en œuvre de façon désynchronisée qui transfère, lorsque cela est nécessaire les caractères à lire ou à écrire de l'unité d'E/S vers le tampon ou inversement.

Une opération de lecture ou d'écriture dans un fichier est réalisée à partir de la position courante dans le fichier, que nous appellerons le curseur. Ce curseur est positionné automatiquement lors de l'ouverture du fichier, soit au début, soit à la fin de celui-ci. Il est déplacé automatiquement lors des opérations de lecture ou d'écriture ou peut l'être à la demande. Un programme peut de plus tester si le curseur est positionné sur la fin du fichier afin de savoir si celle-ci a été atteinte.

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

```
FILE *
```

Par exemple :

```
FILE *mon_fichier;
```

déclare une variable `mon_fichier` *de type adresse d'un descripteur d'un fichier. Un objet de type* `FILE *` *est appelé flot de données (en anglais, stream)*

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

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

Ces fichiers sont déclarés par : `FILE *stdin, *stdout, *stderr;`

9.3 Opérations sur un ficher

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

- *la création : le nom du fichier dont le nom est donné est créé par le système d'exploitation à l'emplacement spécifié par le nom de ce fichier ;*
- *l'ouverture : le tampon est créé, le curseur est positionné en fonction du mode d'accès et l'adresse du descripteur du fichier est retournée ;*
- *la fermeture : les informations contenues dans le tampon sont recopiées dans le fichier, puis le tampon est supprimé ;*
- *les tests de fin de fichier ou d'erreur ;*

– *la lecture ou l'écriture avec ou sans format : les données échangées peuvent être des caractères ou des lignes.*

9.3.1 Création, ouverture et fermeture d'un fichier

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

```
FILE * fopen(char *n, char *m)
```

et sa syntaxe est
`fopen("`*nom-de-fichier*`","`*mode*`")`

L'appel `fopen("`*nom-de-fichier*`","`*mode*`")` *a pour effet de créer un tampon en mémoire, d'ouvrir après l'avoir éventuellement créé, le fichier nom-de-fichier selon le mode mode et de lui associer ce tampon. Cet appel retourne l'adresse du descripteur du fichier si l'ouverture s'estbien déroulé ou* `NULL` *sinon.*

Les modes d'ouverture possibles sont les suivants :

– `"r"` *(lecture) : le fichier nom-de-fichier doit exister, il ne sera accessible qu'en lecture et le curseur est placé en début de fichier ;*
– `"w"` *(écriture) : si un fichier nom-de-fichier existe déjà, il est effacé, puis un fichier de nom nom-de-fichier est créé qui ne sera accessible qu'en écriture et le curseur est placé en début de fichier. Sinon le fichier nom-de-fichier est créé ;*
– `"a"` *(ajout) : si un fichier de nom nom-de-fichier existe déjà, le curseur est placé en fin du fichier (les données seront placées en fin du fichier). Sinon, le fichier nom-de-fichier est créé ; puis un fichier de nom nom-de-fichier est créé qui ne sera accessible qu'en écriture et le curseur est placé en fin de fichier.*

La fermeture d'un fichier est effectuée par l'appel de la fonction `fclose` *d'en-tête :*

```
int fclose(FILE *f)
```

L'appel `fopen(f)` *où* `f` *est le flot de type* `FILE*` *retourné par la fonction* `fopen` *correspondant, a pour effet de transférer le contenu du tampon (on dit : « de vider ») sur l'unité d'entrée/sortie supportant le fichier* `f` *et de libérer la place occupée par ce tampon. Cet appel retourne 0 si la fermeture s'est effectuée correctement ou sinon un entier différent de 0.*

9.3.2 Test de fin de fichier ou d'erreur

Lors de la lecture ou de l'écriture dans un fichier, il peut être nécessaire de savoir si le curseur est positionné en fin de fichier. Ce test peut être réalisé en appelant la fonction `feof` *d'en-tête :*

```
int feof(FILE *f)
```

L'appel `feof(f)` *retourne un entier différent de 0 (vrai) si le curseur est positionné sur la fin du fichier* `f` *ou 0 sinon. Il peut aussi être nécessaire de tester si une lecture ou une écriture s'est effectuée correctement.*

Ce test peut être réalisé en appelant la fonction `ferror` *d'en-tête :*

```
int ferror(FILE *f)
```

L'appel `ferror(f)` *retourne la valeur de l'indicateur d'erreur du fichier* `f` *: un entier différent de 0 (vrai) si une erreur s'est produite ou 0 sinon*

9.3.3 Lecture et écriture d'un caractère

9.3.3.1 Lecture caractère

La lecture d'un caractère dans un fichier est effectuée par l'appel de la fonction `fgetc` *d'en-tête :*

```
int fgetc(FILE *f)
```

L'appel `fgetc(f)` *a pour effet de lire dans le fichier* `f` *le caractère dont la position est indiquée par le curseur, puis d'avancer le curseur d'un caractère. Le caractère lu est retourné, sauf si la fin du fichier a été atteinte ou si une erreur s'est produite, auquel cas* `EOF` *est retourné.*

⚠ **Remarque 9.3.1.** *La lecture d'un caractère dans le fichier d'entrée standard (clavier) peut-être effectuée par l'appel de la fonction* `getchar` *que nous avons étudiée au paragraphe 9.3.1.*
L'appel `getchar()` *est équivalent à* `fgetc(stdin)`.

9.3.3.2 Écriture d'un caractère

L'écriture d'un caractère dans un fichier est effectuée par l'appel de la fonction `fputc` *d'en-tête :*

```
int fputc(int c, FILE *f)
```

L'appel `fputc(c, f)` *a pour effet d'écrire dans le fichier* `f` *le caractère* `c` *à la position du curseur puis d'avancer le curseur d'un caractère. Le caractère* `c` *est retourné, sauf si une erreur s'est produite, auquel cas* `EOF` *est retourné.*

L'écriture d'un caractère dans le fichier standard de sortie (écran) peut-être effectuée par l'appel de la fonction `putchar` *que nous avons étudiée au paragraphe 9.3.1.*

⚠ **Remarque 9.3.2.** *L'appel* `putchar(c)` *est équivalent à* `fputc(c, stdout)`.

▷ **Exemple 9.3.1. : Ecriture et lecture dans un ficher caractère par caractère** *Le programme suivant crée le fichier* `mes_caracteres.txt`*, y écrit un par un les caractères saisis par l'utilisateur jusqu'à la frappe de la touche « Entrée » et ferme ce fichier. Il rouvre ensuite ce fichier en lecture, lit un par un les caractères qui y sont enregistrés et les affiche.*

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

int main(void)
    {
    FILE *f;
    int c;
    f = fopen("mes-caracteres.txt", "w");
```

```
    if (f == NULL)
       {
        printf("Le fichier mes-caracteres.txt ne peut →
           ↪ pas être ouvert !");
        exit(1);
       }
    printf("Saisir une suite de caracteres ? ");
    while((c = getchar()) != '\n')
         fputc(c, f);
    fclose(f);
    f = fopen("mes-caracteres.txt", "r");
    if (f == NULL)
       {
        printf("Le fichier mes-caracteres.txt ne peut →
           ↪ pas être ouvert !");
        exit(1);
       }
    printf("Mes caracteres = ");
    while((c = fgetc(f)) != EOF)
         putchar(c);
    fclose(f);
    return 0;
}
```

9.3.4 Lecture et écriture d'une ligne

9.3.4.1 Lecture d'une ligne

La lecture d'une ligne dans un fichier est effectuée par l'appel de la fonction `fgets` *d'en-tête :*

```
char *fgets(char *s, int n, FILE *f)
```

L'appel `fgets(s, n, f)` *a pour effet de lire dans le fichier* `f` *à partir de la position du curseur les caractères qui suivent et de les placer dans le tableau de caractères* `s` *jusqu'à ce que :*

- *soit* $n-1$ *caractères ont été lus ;*
- *soit le caractère nouvelle ligne (*`'\n'` *a été lu et ajouté à la fin de la chaîne* `s` *;*
- *soit la fin de fichier a été atteinte.*

Le caractère de fin de chaîne (caractère nul `'\0'`*) est ensuite ajouté à la fin de* `s`*. L'adresse* `s` *est retournée, sauf si la fin du fichier a été atteinte sans qu'aucun caractère n'ait été lu ou si une erreur s'est produite, auquel cas* `NULL` *est retourné.*

9.3.4.2 Écriture d'une ligne

L'écriture d'une ligne dans un fichier est effectuée par l'appel de la fonction `fputs` *d'en-tête :*

```
int fputs(char *s, FILE *f)
```

L'appel `fputs(s, f)` *a pour effet d'écrire dans le fichier* `f` *à partir de la position du curseur les caractères de la chaîne de caractères* `s`*. Le caractère de fin de chaîne n'est pas écrit dans le fichier. Un entier* ≥ 0 *est retourné, sauf si une erreur s'est produite, auquel cas* `EOF` *est retourné.*

▷ **Exemple 9.3.2. Ecriture et lecture d'un fichier ligne par ligne** *Le programme suivant crée le fichier* `mon_texte.txt`*, y écrit une suite de ligne, puis le ferme. Il rouvre ce fichier en lecture, lit les ligne qu'il contient, les range dans le tableau* `texte` *et affiche ce tableau. Le tableau* `texte` *peut contenir au maximum* `NBLIGNES` *ayant au maximum* `NBCAR` *caractères (un caractère étant réservé pour le caractère de fin de chaîne).*

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

#define NBLIGNES 100
#define NBCAR 100

int main(void)
{
  FILE *f;
  char texte[NBLIGNES][NBCAR + 1]; /*(il faut compter →
     ↪ le caractère de fin de chaîne)*/
  int i, j;
  f = fopen("mon_texte.txt", "w");
  if (f == NULL)
     {
      printf("Le fichier mon-texte.txt ne peut pas être →
         ↪ ouvert !");
      exit(1);
     }
  fputs("ceci est ma première ligne\n", f);
  fputs("ma deuxième ligne ..\n", f);
  fputs("ma troisième ligne,\n", f);
  fputs("ma quatrième et denière ligne \n", f);
  fclose(f);

  f = fopen("mon_texte.txt", "r");
  if (f == NULL)
     {
      printf("Le fichier mon-texte.txt ne peut pas être →
         ↪ ouvert !");
      exit(1);
     }
  i = 0;
  while (i < NBLIGNES && fgets(texte[i], NBCAR, f))
        i++;
  fclose(f);
  for (j = 0; j < i; j++)
       printf("%s", texte[j]);
  return 0;
```

```
}
```

L'instruction while

```
while (i < NBLIGNES && fgets(texte[i], NBCAR, f))
        i++;
```

lit les vers dans le fichier `mon_texte` *peut-être paraphrasée de la façon suivante : « Tant que le rang de la ligne du texte à lire (i) est inférieur au nombre maximum de lignes (*`NBLIGNES`*) que l'on peut ranger dans le tableau* `texte` *et que cette ligne de texte a été correctement lue par la fonction* `fgets` *et rangée dans la ième ligne du tableau* `texte`*, incrémenter le rang de la ligne à lire et recommencer.*

Cette instruction pourrait être écrite de façon encore plus compacte :

```
while (i < NBLIGNES && fgets(texte[i++], NBCAR, f))
;
```

mais attention à la lisibilité du programme.

9.3.5 Lecture et écriture avec format

Les opérations de lecture et d'écriture avec format dans un fichier sont réalisées par l'appel des fonctions `fscanf` *et* `fprintf` *dont les fonctions* `scanf` *et* `printf` *que nous avons étudiées au paragraphe 9.2 sont des cas particuliers.*

9.3.6 Lecture avec format

La lecture avec format dans un fichier est effectuée par la fonction `fscanf` *d'entête :*

```
int fscanf(FILE *f, char *format, liste d'adresses)
```

L'appel de cette fonction a le même effet que celui de la fonction `scanf` *mais elle lit dans le fichier* `f` *au lieu de lire dans le fichier d'entrée standard. Cet appel retourne le nombre de variables lues si la lecture s'est bien passée, un entier négatif sinon.*
Remarque : On a `scanf(...)` ≡ `fscanf(stdin,...)`

9.3.7 Écriture avec format

L'écriture avec format dans un fichier est effectuée par la fonction `fprintf` *d'entête :*

```
int fprintf(FILE *f, char *format, liste d'expressions)
```

L'appel de cette fonction a le même effet que celui de la fonction `printf` *mais elle écrit dans le fichier* `f` *au lieu décrire dans le fichier d'entrée standard. Cet appel retourne le nombre de caractères écrits si l'écriture s'est bien passée, un entier négatif sinon. On a :* `printf(...)` ≡ `fprintf(stdout,...)`

Au lieu d'écrire avec format dans un fichier, on peut le faire dans une chaîne de caractères. L'écriture avec format dans une chaîne de caractères est effectuée par la fonction `sprintf` *d'en-tête :*

```
int sprintf(char *s, char *format, liste d'expressions)
```

L'appel de cette fonction a le même effet que celui de la fonction `fprintf` *mais elle écrit dans la chaîne* `s` *au lieu d'écrire dans le fichier d'entrée standard. Cet appel retourne le nombre de caractères écrits si l'écriture s'est bien passée, un entier négatif sinon.*

⇒ L'intérêt de cette fonction est de permettre la conversion de nombres en chaînes de caractères ou l'insertion dans une chaîne de caractères de nombres convertis en chaînes de caractères.
Par exemple, si `s` *est un tableau d'au moins 3 caractères et* `x` *est une variable entière de valeur 5, l'appel :*

```
sprintf(s, "%d", x * x)
```

rangera la chaîne de caractères : « *25* » *dans le tableau* `s` *et l'appel :*

```
sprintf(s, "Le carré de %d est %d.", x, x * x)
```

y rangera la chaîne de caractères : « *Le carré de 5 est 25.* ».

Signalons que l'opération inverse : la conversion d'une chaîne de caractères en nombre peut être effectuée en utilisant les fonctions `atoi`, `atol` *et* `atof` *de la bibliothèque standard, d'en-têtes respectives :*

```
int atoi(char *s)
long atol(char *s)
double atof(char *s)
```

Ces fonctions sont déclarées dans le fichier d'en-têtes `stdlib.h`.
Par exemple, l'appel : `atoi("25")` *retournera l'entier 25 de type* `int`.

9.4 Positionnement dans un fichier

Les différentes fonctions d'entrées/sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier.
La fonction `fseek` *permet de se positionner à un endroit précis ; elle a pour prototype :*

```
int fseek(FILE *f, long deplacement, int origine);
```

La variable `deplacement` *détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable origine peut prendre trois valeurs :*

1. `SEEK_SET` *(égale à 0) : début du fichier ;*
2. `SEEK_CUR` *(égale à 1) : position courante ;*
3. `SEEK_END` *(égale à 2) : fin du fichier.*

L'appel `rewind(f)` *de la fonction fonction* `rewind` *de prototype*

```
int rewind(FILE *f);
```

permet de se positionner au début du fichier de tampon `f`. *L'appel de cette fonction* `rewind(f)` *est donc équivalent à*

`fseek(f, 0, SEEK SET);`

L'appel `ftell(f)` *de la fonction* `ftell` *de prototype*

`long ftell(FILE *f);`

retourne la position courante dans le fichier de tampon `f` *(en nombre d'octets depuis l'origine).*

Par exemple, dans le cas d'un fichier binaire

▷ **Exemple 9.4.1.**

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

#define NB 50
#define F_SORTIE "sortie"
int main(void)
{
  FILE *f_in, *f_out;
  int *tab;
  int i;

  tab = (int*)malloc(NB * sizeof(int));
  for (i = 0 ; i < NB; i++)
    tab[i] = i;
  /* ecriture du tableau dans F_SORTIE */
  if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
      fprintf(stderr, "\nImpossible d?ecrire dans le →
          ↪ fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  fwrite(tab, NB * sizeof(int), 1, f_out);
  fclose(f_out);

  /* lecture dans F_SORTIE */
  if ((f_in = fopen(F_SORTIE, "r")) == NULL)
    {
      fprintf(stderr, "\nImpossible de lire dans le →
          ↪ fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  /* on se positionne a la fin du fichier */
  fseek(f_in, 0, SEEK_END);

  printf("\n position %ld", ftell(f_in));
  /* deplacement de 10 int en arriere */
  fseek(f_in, -10 * sizeof(int), SEEK_END);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);
```

```
  /* retour au debut du fichier */
  rewind(f_in);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d", i);

  /* deplacement de 5 int en avant */
  fseek(f_in, 5 * sizeof(int), SEEK_CUR);
  printf("\n position %ld", ftell(f_in));
  fread(&i, sizeof(i), 1, f_in);
  printf("\t i = %d\n", i);
  fclose(f_in);

  return(EXIT_SUCCESS);
}
```

L'exécution de ce programme affiche à l'écran :

```
position 200
position 160    i = 40
position 0      i = 0
position 24     i = 6
```

On constate en particulier que l'emploi de la fonction `fread` *provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.*
Par exemple, dans le cas d'un fichier texte

▷ **Exemple 9.4.2.**

```
#define NB 50
#define F_SORTIE "sortie.txt"
int main(void)
{
  FILE *f_in, *f_out;
  int *tab;
  int i;

  tab = (int*)malloc(NB * sizeof(int));
  for (i = 0 ; i < NB; i++)
    tab[i] = i;
  /* ecriture du tableau dans F_SORTIE */
  if ((f_out = fopen(F_SORTIE, "w")) == NULL)
    {
      fprintf(stderr, "\nImpossible d?ecrire dans le →
          ↪ fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  /* cas d'un fichier texte */
    for (i = 0 ; i < NB; i++){
    fprintf(f_out,"%d ", tab[i]);
    }
```

```
  fclose(f_out);

  /* lecture dans F_SORTIE */
  if ((f_in = fopen(F_SORTIE, "r")) == NULL)
    {
      fprintf(stderr, "\nImpossible de lire dans le →
          ↪ fichier %s\n",F_SORTIE);
      return(EXIT_FAILURE);
    }
  /* on se positionne a la fin du fichier */
  fseek(f_in, 0, SEEK_END);

  printf("\n position %ld", ftell(f_in));
  /* deplacement de 10 caractères en arriere */
  fseek(f_in, -10, SEEK_END);
  printf("\n position %ld", ftell(f_in));
    fscanf(f_in, "%d", &i); /* Cas d'un fichier texte →
        ↪ */

  printf("\t i = %d", i);

  /* retour au debut du fichier */
  rewind(f_in);
  printf("\n position %ld", ftell(f_in));
    fscanf(f_in, "%d", &i); /* Cas d'un fichier texte →
        ↪ */

  printf("\t i = %d", i);

  /* deplacement de 5 caractères en avant */
  fseek(f_in, 5, SEEK_CUR);
  printf("\n position %ld", ftell(f_in));
    fscanf(f_in, "%d", &i); /* Cas d'un fichier texte →
        ↪ */

  printf("\t i = %d\n", i);
  fclose(f_in);

  return(EXIT_SUCCESS);
}
```

L'exécution de ce programme affiche à l'écran :

```
position 140
position 130    i = 47
position 0      i = 0
position 6      i = 3
```

On constate en particulier que l'emploi de la fonction `fscanf` *provoque un déplacement correspondant au caractère lu à partir de la position courante.*

Chapitre 10

Tableaux, structures et unions

Contents

Dans un programme, il est souvent nécessaire de manipuler des données composées d'autres données. Par exemple : un relevé de mesures composé d'une suite de nombres réels ; un point de l'espace composé de son nom (e.g., une lettre) et de ses coordonnées (deux nombres réels) ; une figure géométrique composée d'un ensemble de points. Pour manipuler de telles données, C offre deux structures de données :

- *les tableaux composés de variables ou de tableaux de même type ;*
- *les structures composées de variables ou de tableaux qui peuvent être de types différents.*

Il faut aussi pouvoir manipuler des données qui sont de types différents mais appartiennent à un même type générique. Par exemple : un item qui peut être soit un caractère, soit un nombre entier, soit un nombre flottant. Pour cela, C offre une troisième structure de données : les unions.

10.1 Tableaux

En C, un tableau est une suite de variables ou de tableaux du même type qui constituent les éléments de ce tableau. Dans le premier cas, ce tableau est monodimensionnel et dans le second cas, il est multidimensionnel.

Par la suite, nous appellerons type de base : un type numérique, un type structure, un type union ou un nom de type introduit par une déclaration `typedef`.

10.1.1 Définition d'un tableau monodimensionnel

La déclaration d'un tableau monodimensionnel de variables de type de base `type` *a la forme suivante :*

```
type tab[n];
```

où `tab` *est un identificateur,* `n` *est une expression constante de type entier représente le nombre des éléments du tableau.*

La définition d'un tableau monodimensionnel de variables de type de base `type` *a la forme suivante :*

```
type tab[n] = {exp0,..., expm};
```

ou `exp0,...,expm` *($m \leq n$) sont des expressions.*
Cette définition crée un tableau monodimensionnel de nom `tab` *composé de n variables de type* `tab` *:* `tab[0],..., tab[n-1]`. *Ces variables constituent les éléments terminaux du tableau* `tab`. *Ils sont rangés en mémoire de façon contiguë dans l'ordre suivant :*

`tab[0], tab[1],..., tab[n-1]`

Les `m` *premiers de ces éléments ont pour valeurs initiales respectives les valeurs des expressions* `exp0,..., expm`. *La liste des valeurs initiales est facultative.*

On accède à un élément du tableau en lui appliquant l'opérateur `[]`.

⚠ **Remarque 10.1.1.** *Attention ! Les éléments d'un tableau sont numérotés de 0 à $n-1$ et non de 1 à n.*

⚠ **Remarque 10.1.2.** *Attention ! Un tableau n'est pas une valeur gauche : il ne peut pas être affecté à une variable, ni retourné par une fonction (il n'est pas une lvalue) . Comme nous le verrons par la suite, c'est l'adresse de son premier élément qui identifie un tableau.*

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

▷ **Exemple 10.1.1.** *Le programme suivant définit et affiche les éléments du tableau* `tab` *:*

```
#include<stdio.h>
# define N 5
 int main(void)
  {
   int tab[N] ={1,2,3,4,5};
   int i;
   for (i = 0; i < N; i++)
   printf("tab[%d] = %d\n",i,tab[i]);
   return 0;
  }
```

10.1.2 Définition d'un tableau multidimensionnel

Les éléments d'un tableau peuvent être eux-mêmes des tableaux, ce qui permet de construire des tableaux multidimensionnels. La définition d'un tableau d-dimensionnel est une généralisation de celle d'un tableau monodimensionnel. Elle a la forme suivante :

```
type tab[n1][n2]...[nd] = {exp0,..., expm};
```

où `n1,..., nd` *sont des expressions constantes de type entier.*

Cette définition crée un tableau de nom `tab` *à* `d` *dimensions tel que :*

- *le tableau* `tab` *est composé de n_1 tableaux à* `d-1` *dimensions :* `tab[0],..., tab[n1-1]` *;*
- *chaque tableau* `tab[i1]` *($0 \leq i_1 \leq n_1 - 1$) est composé de n_2 tableaux à $d-2$ dimensions :*
  `tab[i1][0],..., tab[i1][n2- 1]` *;*
- *...*
- *chaque tableau*
  `tab[i1]...[id-1]` $(0 \leq i_1 \leq n_1 - 1, \ldots, 0 \leq i_{d-1} \leq n_{d-1} - 1)$
  est composé de n_d variables de type `type` *:*
  `tab[i1]...[id-1][0],..., tab[i1]...[id-1][nd - 1]`
  qui constituent les éléments terminaux du tableau `tab`*.*

Les éléments terminaux du tableau `tab` *sont rangés en mémoire de façon contiguë en faisant varier l'indice le plus interne d'abord :*

```
tab[0]...[0][0], tab[0]...[0][1],..., tab[0]...[1][0], tab[0]...[1][1],...
```

Les `m` *premiers éléments terminaux ont pour valeurs initiales respectives les valeurs des expressions* `exp0,..., expm`*. La liste des valeurs initiales est facultative. Si elle est omise, la définition d'un tableau se réduit à :*

```
type tab[n1][n2]...[nd];
```

▷ **Exemple 10.1.2.** *Une matrice* `M` *à 2 lignes et 3 colonnes peut être représentée par le tableau* `M` *à 2 dimensions défini par : int* `M[2][3]` *Le tableau* `M[0]` *représente la 1ère ligne de la matrice m dont les éléments sont* `M[0][0]`, `M[0][1]` *et* `M[0][2]` *et le tableau* `M[1]` *représente sa 2e ligne dont les éléments sont* `M[1][0]`, `M[1][1]` *et* `M[1][2]`*. Si les éléments de cette matrice sont connus lors de sa définition, ils peuvent être spécifiés comme valeurs initiales. Par exemple, la matrice :*

$$M = \begin{matrix} 5 & 4 & 3 \\ 6 & 2 & 1 \end{matrix}$$

peut être représentée par le tableau `M`*m défini par :*

```
int M[2][3] = {5, 4, 3, 6, 2, 1};
```

ou mieux par :

```
int M[2][3] = {{5, 4, 3}, {6, 2, 1}};
```

en individualisant chaque sous-tableau.

▷ **Exemple 10.1.3.** *Si m est la matrice de l'exemple précédent, la somme des éléments de cette matrice peut-être calculée et affichée comme suit :*

```
{
 int i, j, s = 0;
 for (i = 0 ; i < 2 ; i++)
 for (j = 0; j < 3; j++)
      s += m[i][j];
 printf("Somme = %d\n", s);
}
```

▷ **Exemple 10.1.4.** *Si m1 et m2 sont deux tableaux définis par : int m1[2][3], m2[2][3] ; l'instruction suivante a pour effet de recopier le contenu du tableau m1 dans le tableau m2 :*

```
for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
        m2[i][j] = m1[i][j];
```

Il n'est pas possible de remplacer cette instruction par :

```
m2 = m1;
```

en pensant recopier globalement le contenu du tableau t2 dans le tableau t1, ou par :

```
for (i = 0; i < 10; i = i + 1)
  m2[i] = m1[i];
```

en pensant recopier globalement le contenu de chaque ligne de m1 dans la ligne correspondante de m2. Ceci, parce que les expressions m2 et m2[i] ne sont pas des valeurs gauche.

10.2 Structures

Une structure est une valeur composée d'une suite de variables ou de tableaux qui constituent les champs de cette structure. Contrairement aux éléments d'un tableau, les champs d'une structure peuvent être de types différents. Toute structure a un type formé des déclarations de ses champs.

10.2.1 Déclaration d'un type structure

La déclaration :

```
struct s
{
type-1 champs-1;
type-2 champs-2;
...
type-n champs-n;
};
```

où `s` *est un identificateur : le nom de la structure (ou son étiquette) et*
`type-1 champs-1;...type-n champs-n;`

sont les déclarations des variables ou des tableaux qui constituent les champs des structures de ce type, déclare un nouveau type : le type `struct s` *qui est un type structure.*

⇒ Les types structure sont des types de base au même titre que les types numériques.

▷ **Exemple 10.2.1.** *Un point du plan décrit par son nom (une lettre), son abscisse et son ordonnée (des réels) dans un repère orthonormé, pourra être représenté par une structure de type :*

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

10.2.2 Définition de variables de type structure

La définition d'une variable de type structure a la forme suivante :

```
struct s var = {expr1,..., expm};
```

où `s` *est le nom d'une structure, varb— est un identificateur et* `expr1,..., expm` *($m \leq$ nombre de champs des structures de type* `struct s`*) sont des expressions.*

La valeur initiale de cette variable est une structure dont les `m` *premiers champs ont pour valeurs initiales respectives les valeurs des expressions* `exp0,..., expm`*. La liste des valeurs initiales est facultative. Si elle est omise, la définition d'une variable de type structure se réduira à :*

```
struct s var;
```

▷ **Exemple 10.2.2.** *définition d'un point dans l'espace :*

```
struct point p = {'A', 1.5, 2.5};
```

provoquera la création des variables suivantes dans la mémoire :

`p.y` *: 2,5*
`p.x` *: 1,5*
`p.nom` *: 65 (code ascii de la lettre 'A')*

10.2.2.1 Sélection de la valeur d'un champ

La sélection de la valeur d'un champ d'une structure est réalisée par l'opérateur `.` *(point).*

▷ **Exemple 10.2.3.** *L'instruction suivante définit deux variables* `a` *et* `b` *de type* `struct point`*, représentant respectivement un point A de coordonnées $(3; 2)$, un point B de coordonnées $(7; 4)$. Elle calcule et affiche les coordonnées du milieu du segment AB.*

```
{
  struct point a = {'A', 3, 2}, b = {'B', 7, 4};
  printf("Coordonnees du milieu du segment AB = (%.2f ; →
     ↪ %.2f)", (a.x + b.x) / 2, (a.y + b.y) / 2);
}
```

10.2.2.2 Affectation d'une valeur à un champ

L'affectation d'une valeur à un champ d'une structure et d'une structure à une variable de type structure est réalisée par l'opérateur =.

⚠ **Remarque 10.2.1.** *Notons, que contrairement aux tableaux, les structures sont des valeurs et peuvent donc être affectées à des variables ou passées en arguments d'une fonction.*

Attention ! Une structure ne peut être affectée qu'à une variable de même type que cette structure. Il n'y a pas de conversion automatique.

▷ **Exemple 10.2.4.** *La fonction suivante retourne le point milieu du segment dont les points extrémités* `a` *et* `b` *lui sont passés en argument et affecte à ce point le nom nom lui-même passé en argument :*

```
struct point_milieu(char nom, struct point a, struct
    ↪ point b)
  {
  struct point m;
  m.nom = nom;
  m.x = (a.x + b.x) / 2;
  m.y = (a.y + b.y) / 2;
  return m;
  }
```

On notera qu'il n'y a pas conflit de nom entre la variable `nom` *et le champ* `nom`*, car ils n'appartiennent pas à la même catégorie de noms.*

10.3 Unions

Une union est une valeur composée d'un ensemble de variables ou de tableaux qui constituent les champs de cette union et qui ont la même adresse. Une variable de type union ne peut donc avoir comme "valeur" que l'un des champs qui la composent.

▷ **Exemple 10.3.1.** *Supposons que nous voulions définir des items dont chacun est soit un caractère, soit un nombre entier, soit un nombre flottant. En C, un tel item pourra être représenté par une union dont le type est déclaré de la façon suivante :*

```
union item
 {
  char c ;
  int e;
  float f;
 };
```

Cette déclaration spécifie qu'une variable `i` *de type* `union item` *est composée de trois variables* `i.c` *de type* `char`*,* `i.e` *de type* `int` *et* `i.f` *de type* `float` *qui constituent ses champs. Si la variable* `i` *a pour valeur un caractère, il sera affecté au champ* `c`*, si elle a pour valeur un entier, il sera affecté au champ* `e` *et si elle a pour valeur un flottant, il sera affecté au champ* `f`*. Puisqu'une variable de type* `union item` *ne peut avoir que l'une de ces trois valeurs, il est possible d'implanter ses champs à partir de la même adresse et c'est ce qui est fait. Une variable* `i` *de type* `union item` *sera définie par :*

`union item i;`

Cette définition provoquera la création des variables suivantes dans la mémoire : Cette

i	i.c i.e i.f	

représentation fait clairement apparaître que i est une variable composée des 3 variables : i.c, i.e et i.f qui ont la même adresse. Si la taille d'une valeur de type char est de 1 octet et celle d'une valeur de type int ou float est de 4 octets, la taille de la case mémoire d'une valeur de type union item sera de 4 octets.

Si l'on souhaite affecter à la variable `i` *: le caractère 'z', on l'affectera au champ c : i.c = 'z' : (122 est le code ASCII de « z »). La valeur entière 50, on l'affectera au*

i	i.c i.e i.f	122

champ i : i.e = 50 et la valeur réelle 3,14, on l'affectera au champ f : i.f = 3.14

⇒ *Les unions sont définies et manipulées de la même façon que les structures.*

▷ **Exemple 10.3.2.** *Reprenons les items (caractère, entier ou flottant) de l'exemple précèdent. Ils pourront être représentés par des structures de type* `struct item` *est déclaré de la façon suivante :*

```
struct item
   {
     char type;
     union
         {
          char c;
          int e;
          float f;
         } u;
   };
```

Si l'item a pour valeur un caractère c, le champ `type` *aura pour valeur le caractère « C » et le champ* `u.c` *aura la valeur* `c`*. Si l'item est un nombre entier* `e`*, le champ type aura pour valeur le caractère « E » et le champ* `u.e` *aura la valeur* `e`*. Si l'item est un nombre flottant* `f`*, le champ* `type` *aura pour valeur le caractère « F » et le champ* `u.f` *aura la valeur* `f`*. L'affichage d'un item pourra être réalisé en lui appliquant la fonction* `afficher_item` *dont la définition est la suivante :*

```
void afficher_item(struct item i)
    {
      switch (i.type)
        {
         case 'C':
              printf("%c\n", i.u.c);
              break;
```

```
        case 'E':
            printf("%d\n", i.u.e);
            break;
        case 'F':
            printf("%f\n", i.u.f);
            break;
        }
    }
```

10.4 Définition de types composés avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de `typedef` *:*

`typedef` *type synonyme ;*

Par exemple, soit la structure `point`

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

la déclaration suivante :

```
typedef struct point Point;
```

déclare que `Point` *est un synonyme de l'expression de type* `struct point`*. Ainsi, une variable* `p` *dont les instances sont des structures de type* `struct point` *pourra être déclarée indifféremment* `struct point p` *ou* `Point p`*.*
De même, la déclaration suivante :

```
typedef struct point Point;
typedef Point Triangle[3];
```

déclare que `Triangle` *est un synonyme de l'expression de type « tableau de 3 variables de type* `Point` *» et donc de l'expression de type « tableau de 3 variables de type* `struct point` *». Ainsi, un tableau* `tab` *de 3 variables de type* `struct point` *pourra être déclaré indifféremment :* `struct point tab[3]`, `Point tab[3]` *ou* `Triangle tab`*.*

Chapitre 11

Adresses et pointeurs

Contents

11.1 Introduction

Dans les programmes que nous avons étudiés jusqu'à présent, l'accès aux variables, aux tableaux et aux fonctions s'est fait par leur nom. Mais il y a des cas où il est nécessaire de le faire par leur adresse pour, entre autres, accéder aux éléments d'un tableau, définir des fonctions qui accèdent directement à la valeur d'une variable définie dans le corps de la fonction appelante, passer un tableau ou une fonction en argument d'une fonction, accéder à des variables anonymes définies dynamiquement pendant l'exécution du programme.

Rappelons :

- *qu'une variable est une case de la mémoire dans laquelle est rangée une valeur. L'adresse d'une variable est l'adresse de cette case.*
- *qu'un tableau est constitué d'éléments qui sont des tableaux ou des variables et qui sont rangés de façon contiguë en mémoire. L'adresse d'un tableau est celle de son premier élément (l'élément de rang 0 de ce tableau).*
- *que le code exécutable d'un programme est lui-même rangé dans la mémoire. A chaque fonction de ce programme, il correspond un point d'entrée dans ce code qui est l'adresse de la première instruction machine à exécuter quand cette fonction est appelée. Cette adresse est l'adresse de la fonction.*

En C, les adresses des variables, des tableaux et des fonctions sont des valeurs au même titre que les nombres, les structures ou les unions. Elles peuvent être la valeur d'un argument ou la valeur de retour d'un opérateur ou d'une fonction. Il est possible d'obtenir l'adresse d'une variable, d'un tableau ou d'une fonction à partir de son nom, d'accéder à la valeur d'une variable ou à un élément de tableau dont l'adresse est donnée, d'appeler une fonction en indiquant son adresse.

⚠ **Remarque 11.1.1.** *Attention ! Si les adresses des fonctions sont des valeurs, les fonctions elles ne le sont pas. Il n'est pas possible d'écrire un programme qui crée une nouvelle fonction ou modifie le code d'une fonction.*

11.2 Notion d'adresses (pointeurs)

11.2.1 Définition d'un pointeur

On appelle « pointeur » une expression ou une variable dont la valeur est l'adresse d'une variable, d'un tableau ou d'une fonction. On dit d'une telle expression qu'elle « pointe » sur cette variable, ce tableau ou cette fonction qui, eux, sont « pointés » par cette expression.

11.2.2 Description d'un type adresse (pointeur)

Les adresses manipulées par un programme C ont un type. L'expression d'un type adresse peut avoir l'une des trois formes suivantes :

- *« adresse d'une variable de type* `T` *» où* `T` *est un type de base ou la description naturelle d'un type adresse ;*
- *« adresse d'un* `T` *» où* `T` *est la description naturelle d'un type tableau ;*
- *« adresse d'une fonction de type $T_1 \times \ldots \times T_n \rightarrow T$ » où $T_1 \times \ldots \times T_n$ et T sont soit des types de base, soit des descriptions naturelles de type.*

Rappelons :

- *qu'un type de base est soit un type numérique, soit un type structure, soit un type union, soit un nom de type introduit par une déclaration* `typedef` *;*
- *que la description naturelle d'un type tableau a la forme suivante : « tableau de n éléments de type* `T` *» où T est soit un type de base, soit la description naturelle d'un type tableau, soit la description naturelle d'un type adresse.*

Par exemple :

- *« adresse d'une variable de type* `int` *» ;*
- *« adresse d'un tableau de 3 éléments de type adresse d'une variable de type* `int` *» ;*
- *« adresse d'une fonction de type* `float` $\rightarrow$ `int` *».*

sont des descriptions naturelles d'un type adresse.

Dans la suite de ce chapitre, nous étudierons :

- *la déclaration de variables de type adresse et les opérateurs de manipulation d'adresses ;*
- *le passage d'arguments par adresse ;*
- *les rôles spécifiques joués par l'adresse nulle et les adresses de type générique ;*
- *l'allocation dynamique de variables ou de tableaux.*

11.3 Adresses de variables, de tableaux et de fonctions

11.3.1 Adresses de variables

La déclaration d'une variable var de type « adresse d'une variable de type `Type` *», où* `Type` *est un type de base, a la forme suivante :*

```
Type *nom;
```

où type `Type` *est le type de l'"objet" pointé et* `nom` *est le nom du pointeur. Cette déclaration déclare un identificateur,* `nom`*, dont la valeur est l'adresse de l'objet pointé qui est de type* `Type`*. L'identificateur* `nom` *est une Lvalue, sa valeur est modifiable.*

Par exemple :

```
int *pi;
char *pc;
struct personne *pp1;
Personne *pp2;
```

déclarent successivement un pointeur `pi` *sur une variable de type* `int`*, un pointeur* `pc` *sur une variable de type* `char`*, un pointeur* `pp1` *sur une variable de type* `struct personne` *et un pointeur* `pp2` *sur une variable de type* `Personne`*.*
On peut déclarer simultanément des variables de type T et des variables de type « adresse d'une variable de type T ». Par exemple :

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

est la déclaration d'une variable `i` *de type* `int` *et d'un pointeur* `pi` *sur une variable de type* `int`*.*

Un élément d'un tableau ou un champ d'une structure peut être de type adresse.

11.3.2 Adresses de structures : pointeurs et structures

On peut manipuler une structure en utilisant son adresse. Pour cela, on déclare un pointeur sur cette structure. Par exemple :

```
struct personne *personnes[50];
```

déclare un tableau personnes dont les éléments pointent sur une variable de type `struct personne` *et :*

```
struct livre
  {
    char titre[30];
    struct personne *auteur;
  }
```

déclare un type `struct livre` *dont le champ* `auteur` *pointe sur une variable de type* `struct personne`.

⚠ **Remarque 11.3.1.** *Un type structure Type est récursif s'il possède un champ qui pointe directement ou indirectement sur une variable de type Type. Par exemple :*

```
struct personne
    {
     char nom[30];
     struct personne *mere;
     struct personne *pere;
    }
```

est la déclaration d'un type structure récursif.

Une telle déclaration est possible car un nom de structure peut être utilisé avant ou dans sa propre déclaration.

Cette déclaration peut être simplifiée en nommant le type `struct personne` *:*

```
typedef struct personne Personne;

struct personne
     {
      char nom[30];
      Personne *mere;
      Personne *pere;
     }
```

Si le père ou la mère d'une personne est inconnu le champ `pere` *ou le champ* `mere` *auront pour valeur l'adresse nulle (*`NULL`*) qui est une instance de tout type adresse.*

11.3.3 Adresse et valeur d'une variable

L'adresse d'une variable est obtenue par l'opérateur `&`*. Si* `expr` *est une valeur gauche de type* `Type` *alors :*

`&exp`

est une expression qui a les propriétés suivantes :

- `type(&expr)` *= adresse d'une variable de type* `Type`
- `val(&expr)` *= adresse de la variable désignée par* `expr`.

⚠ **Remarque 11.3.2.** *Attention !* `&exp` *n'est pas une valeur gauche. Par exemple, si* `i` *est une variable de type* `int`*, on ne pourra pas écrire* `&i = 12` *en pensant affecter la valeur* 12 *à* `i`*. Il suffit, évidemment, d'écrire i = 12.*

L'opérateur ⋆ permet d'obtenir la valeur d'une variable à partir de son adresse. On dit que l'on « déréférence » cette adresse. Si `expr` *est une expression de type « adresse d'une variable de type* `Type` *» alors :*

```
*exp
```

est une expression qui a les propriétés suivantes :
- `type(*expr)` = `Type`
- `val(*expr)` = *valeur de la variable d'adresse* `val(expr)`
- *c'est une valeur gauche.*

▷ **Exemple 11.3.1.** *Suite aux instructions suivantes,*

```
{
  int i, *pi;
    i = 0;
    pi = &i;
    *pi = 5;
}
```

`i` *aura la valeur 5.*

11.3.3.1 Accès aux champs d'une structure ou d'une union pointée

L'opérateur -> permet d'obtenir la valeur d'un champ d'une structure ou d'une union dont l'adresse est donnée. L'expression :

```
expr->c
```

est équivalente à :

```
(*expr).c
```

▷ **Exemple 11.3.2.** *Soit* `p` *et* `pp` *deux variables définies par :*

```
struct personne
{
char nom[30];
int age;
} p = {"Dupont", 20}, *pp;
```

Après de l'exécution de l'instruction :

```
pp = &p;
```

on aura : valeur(`pp->age`*) = valeur(*`p.age`*) = 20*

11.3.4 Adresses de tableaux : pointeurs et tableaux

Un tableau est identifié par l'adresse de son premier élément (élément de rang 0). Une expression de type tableau a donc pour valeur l'adresse de l'élément de rang 0 du tableau qu'elle désigne. C'est le cas, notamment, du nom d'un tableau qui de plus est une constante.

Soit les deux tableaux définis par :

```
int tab[12];
int m[2][3];
```

Dans une expression :

- `tab` *est une constante de type « adresse d'une variable de type int » qui a pour valeur l'adresse de la variable* `tab[0]` *;*
- `m` *est une constante de type « adresse d'un tableau de 3 éléments de type* `int` *» qui a pour valeur l'adresse du sous-tableau* `m[0]`.

La déclaration d'une variable `p` *de type « adresse d'un tableau de n_1 éléments de type tableau de ... de type tableau de n_k éléments de type* `Type` *» où* `Type` *est un type de base, a la forme suivante :*

```
Type (*p)[n1]...[nk];
```

Par exemple :

```
float (*ptf)[3];
```

déclare une variable `ptf` *de type « adresse d'un tableau de 3 éléments de type* `float` *».*

L'opérateur de déréférencement $\star$ s'applique aussi aux adresses de tableaux. Si `expr` *est une expression de type « adresse d'un tableau d'éléments de type* `Type` *», alors :*

```
*expr
```

est une expression qui a les propriétés suivantes :

- `type(*exp)` = *« adresse d'un élément (tableau ou variable) de type* `Type` *»*
- `val(*exp)` = *« adresse de l'élément de rang 0 du tableau pointé par* `expr` *»*
- *ce n'est pas une valeur gauche.*

11.3.5 Arithmétique des pointeurs

C a la particularité d'offrir trois opérations arithmétiques pour manipuler les adresses d'éléments de tableaux : l'addition et la soustraction d'un nombre à une adresse et la soustraction de deux adresses.

Soit tab un tableau. Si $expr_1$ est une expression dont la valeur a est l'adresse d'un élément du tableau `tab` *et si $expr_2$ est une expression dont la valeur i est un nombre entier, alors :*

- *val*$(expr_1 + expr_2)$ = *adresse du ie élément suivant l'élément d'adresse a*
- *val*$(expr_1 - expr_2)$ = *adresse du ie élément précédant l'élément d'adresse a*
  Si $expr_1$ et $expr_2$ sont des expressions dont les valeurs a_1 et a_2 sont les adresses d'un élément du tableau tab, alors :
- *val*$(expr_1 - expr_2) = i$, *si* $a_1 + i = a_2$

L'opérateur `[]` *d'accès à un élément de tableau est défini en fonction de l'addition d'une adresse et d'un nombre. On a l'équivalence :*

$$expr_1[expr_2] \equiv *(expr_1 + expr_2)$$

Si tab est le nom d'un tableau et `i` *un indice, on a :*

`tab[i]` $\equiv$ `*(tab+i)`

Notons de plus que si tab est le nom d'un tableau, on a :

`tab + i` ≡ `& tab[i]`
et
`tab` ≡ `& tab[0]`
car la valeur du nom d'un tableau est égale à l'adresse de son élément de rang 0.

▷ **Exemple 11.3.3.** *L'instruction :*

```
{
  int tab[12] = {68, 25, 20, 22, 21, 0, 1, 4, 130, 57, 68, 97};
  int *a, *s;
  a = tab + 3;
  s = tab + 8;
printf("val(*a) = %d, val(*s) = %d, val(s - a) = %d", *a, *s, s-a);
}
```

affichera :

```
val(*a) = 22, val(*s)= 130, val(s - a)= 5
```

▷ **Exemple 11.3.4.** *L'instruction :*

```
{
  int m[2][3] = {{5, 4, 3}, {6, 2, 1}};

  int (*r)[3];
/* adresse du sous tableau {5,4,3} : 1er elmnt de la matrice m */

  int *e;

  r = m + 1;
/* adresse du sous tableau {6, 2, 1} : 2eme elmnt de la matrice m */

  e = (*r) + 2;
/* adresse du 3eme elment du sous tableau {6, 2, 1} */

  printf("val(*e) = %d, val((*r)[2]) = %d, val(m - r) = %d", /
   *e, (*r)[2], m - r);
}
```

où la variable `r` *est de type adresse d'un tableau de 3 éléments de type* `int` *à laquelle est affecté l'adresse du sous-tableau de rang* 1 *du tableau* `m`*, affichera :*

```
val(*e) = 1, val((*r)[2]) = 1, val(m - r) = -1
```

11.3.6 Passage d'arguments par adresse

En C, les arguments d'une fonction sont « passés par valeur » : les arguments effectifs sont affectés aux l'argument formels correspondants. Dans le cas où un argument effectif est la valeur d'une variable de la fonction appelante, la valeur de cette variable est recopiée dans l'espace mémoire de la fonction appelée.

Le passage d'un argument par valeur a deux inconvénients :

- *une fonction ne peut pas modifier la valeur d'une variable de la fonction appelante car elle n'a accès qu'à une copie de la valeur de cette variable ;*
- *si l'argument effectif est une structure volumineuse, sa copie peut être coûteuse en temps et en mémoire.*

La solution, dans ces deux cas, est de passer en argument l'adresse de cette variable au lieu de sa valeur. On dit alors que cet argument est « passé par adresse ».

11.3.6.1 Passage en argument de l'adresse d'une variable

L'exemple suivant illustre le passage en argument de l'adresse d'une variable et son intérêt lorsqu'une fonction doit retourner plusieurs valeurs.

▷ **Exemple 11.3.5.** *Le programme suivant appelle une fonction* `somme_et_produit` *pour calculer la somme et le produit de deux nombres.*

```
void somme_et_produit(int x, int y, int *s, int *p)
{
  *s = x + y;
  *p = x * y;
}


int main(void)
{
 int somme, produit;
 somme_et_produit(2, 3, &somme, &produit);
 return 0;
}
```

11.3.6.2 Passage en argument de l'adresse d'une structure

L'exemple suivant montre comment définir une fonction qui modifie les champs d'une structure dont l'adresse est passée en argument.

▷ **Exemple 11.3.6.**

```
struct employe
{
char nom[30];
float salaire;
}
```

La fonction :

```
void augmenter_salaire(struct employe *pe, float taux)
{
pe->salaire *= (1 + taux);
}
```

augmente de taux% le salaire d'un employé `pe` *où* `pe` *est un pointeur sur la structure qui décrit cet employé.*

11.3.6.3 Passage en argument de l'adresse d'un tableau

En C, passer un tableau en argument c'est passer l'adresse de ce tableau en argument. Le tableau lui-même ne peut pas l'être puisqu'un tableau n'étant pas une valeur, il ne peut pas être affecté à un argument formel qui est une variable. De toute façon ce serait une solution coûteuse, puisqu'elle entraînerait une copie du tableau à l'appel de la fonction et une au retour dans la fonction appelante, dans le cas où la fonction appelée a modifié le tableau.

Il y a deux façons de déclarer un argument de type adresse d'un tableau d'éléments de type T : soit comme un nom de tableau d'éléments de type T, soit comme une variable de type adresse d'un élément de type T. Le nombre d'éléments du tableau passé en argument peut être omis mais pas les nombres d'éléments de ses sous-tableaux qui sont nécessaires pour calculer leurs adresses.
Par exemple

- *si le tableau dont l'adresse doit être passée en argument est de type tableau de 10 éléments de type* `int` *et que le nom de cet argument est* `tab`*, il pourra être déclaré :* `int tab[10]`, `int tab[]` *ou* `int *tab;`
- *si le tableau dont l'adresse doit être passée en argument est de type tableau de 2 tableaux de 3 éléments de type* `int` *et que le nom de cet argument est* `tab`*, il pourra être déclaré :* `int t[2][3]`, `int t[][3]` *ou* `int (*t)[3]`.

▷ **Exemple 11.3.7.**

```
int somme_tableau(int tab[], int taille)
{
int i, s = 0;
for (i = 0; i < taille; i++)
s += tab[i];
return s;
}
```

⚠ **Remarque 11.3.3.** *L'argument* `tab` *aurait pu être déclaré* `int tab[10]` *ou* `int *t`. *La déclaration* `int tab[]` *a l'avantage de mettre en évidence que cette fonction peut être utilisée pour un tableau de nombre d'éléments quelconque à condition bien sûr que ce nombre d'éléments soit passé en argument.*

La fonction suivante calcule et retourne le déterminant d'une matrice de 2x2 entiers dont l'adresse `m` *est passée en argument :*

```
int determinant_matrice_2_2(int m[2][2])
{
return m[0][0] * m[1][1] - m[0][1] * m[1][0];
}
```

L'argument `m` *aurait pu être déclaré* `int m[][2]` *ou* `int (*m)[2]` *mais ni* `m[][]` *ni* `(*m)[]`

11.4 Allocation dynamique de variables ou de tableaux

Dans les programmes que nous avons étudiés jusqu'à présent, les variables manipulées sont définies dans le texte du programme. Ceci est suffisant lorsque les variables sont connues lors de l'écriture du programme, mais ce n'est pas toujours le

cas. Considérons, par exemple, un programme qui gère des objets dont le nombre varie au cours de l'exécution du programme. Une première solution est de fixer dans le texte du programme un nombre maximum N d'objets et de réserver en mémoire une place pour ces N objets. Mais, si N a été a été sous-évalué, il faudra le modifier et recompiler le programme pour pouvoir gérer un plus grand nombre d'objets et si N a été surévalué, de la place mémoire aura été réservée pour rien.

Une meilleure solution serait, lors de la création d'un nouvel objet, de pouvoir allouer la place mémoire nécessaire à sa description et lors de la suppression d'un objet, de pouvoir libérer la place mémoire occupée par sa description. Ceci est possible en C, grâce à la fonction `malloc` *qui permet d'allouer dynamiquement une variable en mémoire et à la fonction* `free` *qui permet de libérer la place occupée par cette variable lorsqu'elle n'est plus utilisée.*

11.4.1 Taille des instances d'un type

La taille mémoire des instances d'un type peut être obtenue en utilisant l'opérateur `sizeof` *qui peut être utilisé de deux façons différentes :*

- *l'appel* `sizeof(T)` *retourne la taille en nombre d'octets d'une valeur de type T ;*
- *l'appel* `sizeof exp` *a pour valeur la taille en nombre d'octets d'une valeur de type* `type(exp)`.

⚠ **Remarque 11.4.1.** *Attention ! si* `tab` *est un nom de tableau, l'expression :*

```
sizeof tab
```

a pour valeur la taille de ce tableau (c.-à-d. le nombre d'octets occupés par la suite de ses éléments) et l'expression :

```
sizeof tab / sizeof tab[0]
```

a pour valeur le nombre d'éléments de ce tableau.

▷ **Exemple 11.4.1.** *Par exemple, si* `tab` *est un tableau défini par :* `int tab[10];` *et si la taille d'une valeur de type int est 4 octets, on a :*

```
val(sizeof tab) = 40
val(sizeof tab / sizeof tab[0]) = 10
```

11.4.2 Allocation dynamique d'une variable

L'allocation dynamique d'une variable est réalisée en utilisant la fonction `malloc` *de la bibliothèque standard. , déclarée dans le fichier stdlib.h. Elle a pour en-tête :* `void *malloc (size_t t)` *Elle alloue dans le tas un emplacement de* `t` *octets pour stocker la variable et retourne l'adresse de cet emplacement.*

L'allocation dynamique d'une variable de type T sera alors réalisée en évaluant l'expression :

```
(Type *)  malloc(taille d'une valeur ou d'un tableau de type Type)
```

qui aura pour valeur l'adresse de cette variable.

⚠ **Remarque 11.4.2.** *Notons que* `malloc` *retournant une adresse de type générique* `(void *)`, *il est nécessaire de la convertir en une adresse d'une variable de type T en lui appliquant l'opérateur de changement de type* `(T *)`.

▷ **Exemple 11.4.2.** *On suppose qu'un point du plan est représenté par une instance du type Point dont la déclaration est la suivante :*

```
typedef struct
{
char nom;
float x, y;
} Point;
```

La fonction suivante :

```
Point *creer_point(char nom, float x, float y)
{
 Point *pp;
 pp = (Point *) malloc(sizeof(Point));
 if (pp == NULL)
    {
     printf("Allocation impossible !");
     exit(1);
    }
 pp->nom = nom;
 pp->x = x;
 pp->y = y;
 return pp;
}
```

crée un point de nom `nom` *et de coordonnées* `x` *et* `y`*. Plus précisément, elle alloue dynamiquement une variable de type* `Point` *et affecte son adresse à la variable* `pp` *de type « adresse d'une variable de type* `Point` *». Elle affecte ensuite aux champs* `nom`*,* `x` *et* `y` *de la structure pointée par* `pp` *les valeurs des variables* `nom`*,* `x` *et* `y`*. Enfin, elle retourne l'adresse* `pp`*.*

11.4.3 Allocation dynamique d'un tableau

Rappelons qu'un tableau, en C, est une suite contiguë d'éléments qui peuvent être des variables ou des tableaux. Pour allouer dynamiquement un tableau de n éléments de type Type, il faut allouer dans le tas un emplacement de (n x taille de Type) octets dans lequel seront stockés les n éléments de ce tableau et récupérer l'adresse du début de cette zone convertie en une adresse d'une variable de type Type : celle de l'élément de rang 0 de ce tableau. Ceci peut être fait en utilisant la fonction `malloc` *et en évaluant l'expression :*

```
(Type *) malloc(n * taille de Type)
```

▷ **Exemple 11.4.3.** *La fonction suivante alloue dynamiquement un tableau de n entiers, remplit ce tableau de telle façon que chaque élément a pour valeur son rang et retourne l'adresse de l'élément de rang 0 de ce tableau*

```
int *allouer_tableau(int n)
 {
   int *tab, i;
   tab = (int *) malloc(n * sizeof(int));
```

```
    if (tab == NULL)
        {
         printf("Allocation impossible !");
         exit(1);
        }
  for (i = 0; i < n; i++)
     tab[i] = i;
  return tab;
}
```

⇒ *On peut aussi utiliser la fonction* `calloc` *de la bibliothèque standard Cette fonction est déclarée dans le fichier* `stdlib.h`*. Son en-tête est :*

```
void *calloc (size_t n, size_t t)
```

Elle alloue dans le tas n variables consécutives de taille t, les initialise à 0 (ce que ne fait pas la fonction `malloc`*) et retourne l'adresse de la première de ces variables. L'allocation dynamique d'un tableau de* n *éléments de type* `Type` *sera alors réalisée en évaluant l'expression :*

```
(Type *) calloc(n, taille de Type)
```

qui aura pour valeur l'adresse de l'élément de rang 0 de ce tableau. En cas d'échec de l'allocation, du à une place mémoire insuffisante, la fonction `calloc` *retourne l'adresse nulle.*

11.4.4 Libération d'un emplacement mémoire alloué dynamiquement

Lorsqu'une variable ou un tableau alloué dynamiquement par l'appel de la fonction `malloc` *ou* `calloc`*, n'est plus utilisé par le programme en cours, on pourra demander sa libération en faisant appel à la fonction* `free` *de la bibliothèque standard. Cette fonction est déclarée dans le fichier* `stdlib.h`*. Son en-tête est :*

```
void free(void *a)
```

Elle signale que l'emplacement d'adresse a, qui devra avoir été alloué par un appel à `malloc` *ou* `calloc`*, peut être récupéré.*

⚠ **Remarque 11.4.3.** *Attention ! Avant de faire appel à* `free`*, il faut être sûr que la variable ou le tableau stocké dans l'emplacement à libérer ne sera plus utilisé dans la suite de l'exécution du programme.*