École Centrale Marseille

Programmation Structurée en Langage C

Stéphane Derrode

Mathématique et Informatique
Révision 2.5, 2006.

Table des matières

Chapitre 1

En guise d'introduction ...

Sommaire

1.1 Quelques repères historiques

Le langage C est lié à la conception du système UNIX par les Bell-Labs [JR78]. Les langages ayant influencés son développement sont :

- Le langage BCPL de M. Richards - 1967,
- Le langage B développé aux Bell-Labs - 1970.

Ces deux langages partagent avec le C :

- Les structures de contrôle,
- L'usage de pointeurs,
- La récursivité.

Les deux langages prédécesseurs du C avaient la particularité d'être sans type. Ils ne connaissent que le mot machine, ce qui leur donne un degré de portabilité nul. Le langage C comble ces lacunes en introduisant des types de données tels que l'entier ou le caractère.

Les dates marquantes de l'histoire du C sont les suivantes :

1972 La première version du C est écrite en assembleur par B. Kerninghan et D. Ritchie,

1973 Alan Snyder écrit un compilateur C portable,

1989 Sortie du premier document normalisé appelé norme ANSI[1] X3-159,

1990 Réalisation du document final normalisé auprès de l'ISO[2] : ISO/IEC 9899 [98990].

Jusqu'en 1987, il n'y avait pas de norme. Le livre « The C programming language » [RK78] contient une définition précise du langage C appelée « C reference manual ». C'est principalement de ce livre (et de sa traduction française [KR94]) que s'inpire ce support de cours.

1. — « American National Standard Institute ».
2. — « International Standard Organization ».

1.2 Présentation du langage C

Le langage C est un langage de bas niveau dans la mesure où il permet l'accès à des données que manipulent les ordinateurs (bits, octets, adresses) et qui ne sont pas toujours disponibles dans les langages évolués tels que le Fortran, le Pascal ou ADA.

Le langage C a été conçu pour l'écriture de systèmes d'exploitation. Plus de 90% du noyau du système UNIX est écrit en C. Le compilateur C lui-même est écrit en grande partie en langage C ou à partir d'outils générant du langage C. Il en est de même pour les autres outils de la chaîne de compilation : assembleurs, éditeurs de liens, pré-processeurs. De plus, tous les utilitaires du système UNIX sont écrits en C (« shell », outils).

Il est cependant suffisamment général pour permettre de développer des applications variées de type scientifique, ou encore pour l'accès aux bases de données (application de gestion)[3]. Le langage C est disponible sur pratiquement toutes les plate-formes, de l'ordinateur personnel jusqu'aux gros calculateurs scientifiques, en passant par les stations de travail. De nombreux logiciels du domaine des ordinateurs personnels, tels que Microsoft Word ou Excel sous le système Windows, sont eux-aussi écrits à partir du langage C, ou de son successeur orienté objet C++ [Str86].

Le C est un langage impératif classique qui comporte
- ➪ des types standards de base (entiers, réels, caractères),
- ➪ des structures de contrôle (Si ...alors, séquences, boucles),
- ➪ des constructions de types (tableaux, unions, enregistrements),
- ➪ des sous-programmes (appelées fonctions).

Il reflète bien le savoir faire des années 70, et se situe dans la famille du langage Pascal. Son avantage vis-à-vis du Pascal est son plus grand pragmatisme. Il autorise clairement deux styles de programmation, le style « bidouille » pour produire du code efficace et le style « génie logiciel » pour produire des programmes plus lisibles, plus sûrs et plus facilement modifiables.

Bien que pouvant être considéré de bas niveau, le langage C supporte les structures de base nécessaires à la conception des applications structurées. Cette caractéristique le range dans la catégorie des langages de haut niveau. Il est aussi un des premiers langages offrant des possibilités de programmation modulaire. Un programme en C peut être constitué de plusieurs modules. Chaque module est un *fichier source* qui peut être compilé de manière autonome pour obtenir un *fichier objet*. L'ensemble des fichiers objets participant à un *programme* doivent être associés pour constituer un *fichier exécutable* (cf. figure 1.1).

Lorsque nous parlons du langage C, nous faisons référence à ce que sait faire le compilateur lui-même. Plusieurs outils interviennent dans la transformation d'un ensemble de fichiers sources, constituant un programme, en un fichier binaire exécutable, résultat de ce que l'on appelle communément, la *compilation d'un programme*.

Le langage C se limite aux fonctionnalités qui peuvent être traduites efficacement en instructions machine. Cette règle de fonctionnement doit permettre de détecter ce qui est fait directement par le compilateur lui-même et ce qui ne peut pas être fait. Illustrons cette règle par quelques exemples :
- ➪ Le compilateur est capable de générer des instructions machines qui permettent de manipuler des élément binaires (« bit ») ou des groupes d'éléments binaires (« octet », ou « byte » en anglais).
- ➪ Il permet les manipulations algébriques (addition, multiplication, ...) de groupes d'octets qui représentent des valeurs entières ou des valeurs réelles.
- ➪ Il permet de manipuler des caractères en considérant que ceux-ci sont représentés par un octet

3. — Il faut noter que les langages Fortran et Cobol gardent une place prépondérante, respectivement, dans les domaines du calcul scientifique et de la gestion.

à partir du code ASCII[4].

- ➥ Il ne permet pas de manipuler directement les tableaux. En particulier, il ne permet pas de manipuler directement les groupes de caractères utilisés pour stocker les chaînes de caractères. Pour cela, il faut faire appel à des fonctions de bibliothèques (cf. section 12.1).
- ➥ De manière contradictoire à la règle précédente, le compilateur accepte l'affectation d'une collection de données groupées (structure) par une collection de données de type identique. Ceci s'explique par le fait qu'une structure est considérée dans le langage comme une donnée simple, elle doit donc pouvoir être affectée à une autre donnée de même type.

Pour réaliser des fonctions plus compliquées, le programmeur doit écrire ses propres fonctions ou faire appel aux fonctions pré-définies de la bibliothèque du langage C (cf. chapitres 11 et 12). Ces fonctions sont elles-aussi standardisées.

1.3 Premier programme en C

Le programme 1.1 affiche « Hello World ! » sur une console à l'écran.

Prog. 1.1 – Hello World !

```
/*
   Premier Programme : Affiche Hello World!
*/
 #include <stdio.h>

 int main() {

    // Affiche le message
    printf("\nHello World!");

    // Valeur de retour de la fonction
    return 0;
 }
```

Notez en premier lieu que toutes les instructions se terminent par un point-virgule « ; ». Omettre un point virgule déclenche un message d'erreur lors de la compilation.

Des *commentaires* peuvent être insérés n'importe où dans le programme, dès lors qu'ils sont placés entre les délimiteurs de début « /* » et de fin « */ » (cf. lignes 1 à 3). Un commentaire peut également être introduit par « // »[5] (cf. lignes 8 et 11). On peut ainsi détailler de façon particulièrement utile les fonctionnalités du programme et sa logique sous-jacente.

La ligne 6 définit l'en-tête de la *fonction principale* de notre programme. La fonction principale d'un programme s'appellera toujours `main`. Dans notre cas, cette fonction ne prend pas de paramètre en entrée (parenthèses ouvrante et fermante), mais retourne une valeur de type `int` (c'est à dire une variable de type entier). Le retour est réalisé grâce à l'instruction `return` en ligne 12 (soit juste avant la fin du programme), la valeur de retour de la fonction étant `0`. En général, `0` signifie une terminaison sans erreur.

4. — ASCII : « American Standard Code for Information Interchange », cf. annexe A. Ensemble de caractères codés sur 7 bits. En conséquence, les accents et les caractères spéciaux comme le « ç » ne sont pas permis !

5. — Ce type de commentaire a été introduit avec le langage C++ et n'est pas purement C-ANSI. Il est cependant supporté par tous les compilateurs actuels. Précisons que dans ce cas, le commentaire doit se limiter à une unique ligne.

Voyons maintenant plus précisément le contenu de la ligne 9 qui réalise l'affichage de la phrase « Hello World ! » sur une console. La commande utilisée pour afficher la *chaîne de caractères* est l'instruction `printf(...)`, la chaîne à afficher étant entre guillemets. Le caractère `\n` placé en début de chaîne indique qu'un saut de ligne doit être réalisé avant d'afficher la phrase [6].

À sa base, le langage C n'est qu'un ensemble de bibliothèques à partir desquelles le compilateur trouve les fonctions et les applications qui lui permettent de créer un programme exécutable. Exactement ce que l'on fait lorsqu'on cherche dans une encyclopédie pour faire un exposé. Certaines bibliothèques (les plus courantes) sont incluses dans le compilateur, ce qui permet à notre programme de compiler. Ainsi, l'instruction `printf` est définie dans la bibliothèque `stdio.h` (bibliothèque standard d'entrées/sorties), que l'on inclus dans le programme grâce à la directive `#include` (ligne 4).

Les librairies standards du C seront présentées au fur et à mesure de leur utilisation dans ce cours. Néanmoins, nous pouvons déjà en dire quelques mots. À l'instar de l'étudiant qui recherche dans des livres, on peut dire que le fichier « .h » représente l'index du livre et le fichier « .cpp » correspondant le contenu du chapitre concerné. Ainsi, lorsque le compilateur rencontre le mot `printf`, il regarde dans chacun des fichiers « .h » déclaré par l'instruction `#include` si ce mot est défini. Il trouve celui-ci dans la bibliothèque `stdio.h`. À l'inverse, s'il ne le trouve pas, le compilateur émet un message d'erreur.

1.4 Langage C et programmation structurée

La programmation structurée est un nom générique qui couvre un courant de pensée. Ce courant de pensée s'est développé entre les années 1965 et 1975. La programmation structurée a pour but de faciliter le travail de relecture des programmes et de minimiser le travail de maintenance (corrections, ajouts de fonctionnalités, ...).

Le langage C est apparu en 1972, c'est à dire en pleine période de réflexion sur les langages structurés. Il supporte donc un ensemble de fonctionnalités qui sont directement issues de ce courant de pensée. Le langage C a été conçu et réalisé pour écrire un système d'exploitation et le logiciel de base de ce système. Il doit être capable de faire les mêmes choses que l'assembleur. Il est assez permissif, ce qui va à l'encontre de la programmation structurée telle que Wirth [Wir74] l'a décrite. En effet, en C, le programmeur peut écrire des choses explicites qui sont liées à la structure de la machine.

Le langage C est assez peu contraignant. Il offre des structures de programme mais il n'oblige pas à les utiliser. En particulier, il autorise les entrées multiples et les sorties multiples dans les tâches. La mise en page est libre, ce qui permet d'écrire des programmes dont la mise en page reflète la structure. Les programmes sans mise en page sont rapidement illisibles du fait de la richesse de la syntaxe du C.

Comme le montre la figure 1.1, un programme en C est constitué d'un ensemble de fichiers sources destinés à être compilés séparément et à subir une édition de liens commune [7]. Ces fichiers sources sont également appelés « modules », et ce type de programmation est appelée « programmation modulaire ». La visibilité des modules entre-eux est expliquée plus loin dans ce support de cours (cf. chapitre 8).

Le fait de pouvoir compiler chaque fichier source de manière autonome amène à concevoir des programmes de manière modulaires en regroupant, dans chaque fichier source, des fonctions qui manipulent les mêmes *variables* ou qui participent aux mêmes *algorithmes*.

6. — La commande `printf` ne se limite pas à l'affichage d'une chaîne de caractères, mais est beaucoup plus puissante comme nous le verrons dans le chapitre 3.

7. — Les différentes étapes de la compilation sont plus complexes que cela. Toutes ces opérations sont relativement transparentes avec le logiciel Visual C++ de Microsoft, puisqu'il suffit de cliquer sur un bouton pour réaliser l'ensemble des étapes aboutissant à l'exécutable !

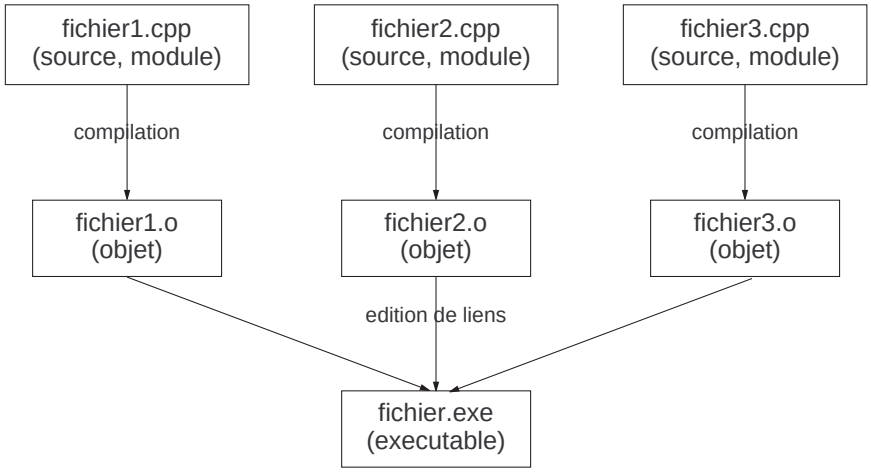

Figure 1.1 – Les étapes de compilation d'un programme.

Maîtriser la programmation en langage C nécessite beaucoup de savoir faire et donc de pratique. Elle s'apprend essentiellement à partir de ses erreurs, alors n'hésitez pas à mettre les mains dans le « cambouis » ...

Chapitre 2

Types et variables

Sommaire

Ce chapitre traite des définitions de variables. Dans tous les langages, une définition de variable a les rôles suivants :

1. définir le domaine de valeur de cette variable (taille en mémoire et représentation machine).
2. définir les opérations possibles sur cette variable.
3. définir le domaine de validité de cette variable.
4. permettre à l'environnement d'exécution du programme d'associer le nom de la variable à une adresse mémoire.
5. initialiser la variable avec une valeur compatible avec le domaine de valeur.

En langage C, une variable se caractérise à partir de son type et de sa classe mémoire. Les points 1 et 2 sont associés au type de la variable, les points 3 et 4 sont associés à la classe mémoire de la variable.

2.1 Types de base

Ce sont les types pré-définis du compilateur. Ils sont au nombre de cinq[1] et on peut les classer en trois catégories :

- Les types entiers,
- Les type réels,
- Le type vide.

2.1.1 Les type entiers : `char` et `int`

On distingue principalement deux types entiers :

char : C'est le type caractère. Il représente un nombre entier codé sur un octet (huit bits). Sa valeur peut donc évoluer entre -128 et $+127$ ($2^8 = 256$ possibilités). Le nom `char` n'est pas anodin car il est le support des caractères au sens commun du terme. La correspondance entre un nombre et le caractère qu'il représente est transcrite dans la table ASCII (cf. annexe A).

int : C'est le type entier. Il est généralement codé sur 4 octets (32 bits) et permet de représenter $2^{32} = 4294967295$ nombres entiers. La plage des valeurs acceptables pour ce type est donc $[-2147483648; 2147483647]$!

Les types entiers peuvent être qualifiés à l'aide du mot `unsigned` qui force les variables de ce type à être considérées comme uniquement positives[2]. Par exemple, la valeur d'une variable du type `unsigned char` ne peut évoluer qu'entre 0 et 255.

Le type `int` se décline avec les qualificateurs `short` ou `long` pour préciser sa taille. Cela donne la possibilité d'avoir des définitions du type : `short int` ou `long int`[3]. Le langage C considère les types `char`, `short int`, `int` et `long int` comme des types entiers et permet de les mélanger lors des calculs.

Les plages des valeurs acceptables pour tous les types entiers sont données dans le fichier `limits.h`.

2.1.2 Les types réels : `float`, `double` et `long double`

Ces types servent à représenter les nombres réels. On distingue trois types réels qui se distinguent par :

- la précision sur les parties décimales et,
- les plages des valeurs acceptables.

Les plages des valeurs acceptables pour tous les types réels sont données dans le fichier `float.h`.

float : C'est le type de base des nombres réels.

double : Ce type permet de représenter des valeurs ayant une partie décimale avec une plus grande précision que le type `float`.

long double : Ce type est récent. Il permet de représenter des nombres avec parties décimales qui nécessitent une très grande précision, si la machine le permet.

1. — Il n'y a pas de type booléen en C. Celui-ci a été introduit plus tard avec le langage C++, l'extension orientée objet du langage C.

2. — Le qualificateur `signed` est appliqué par défaut. Ainsi, il n'y a pas de différence entre une variable de type `int` et une variable de type `signed int`.

3. — Ces définitions peuvent aussi s'écrire de manière abrégée : `short` ou `long`, le type de base `int` étant inclus implicitement.

2.1.3 Le type vide : `void`

C'est le type vide. Il est surtout utilisé pour préciser les fonctions sans argument ou sans retour. Il joue également un rôle particulier dans l'utilisation des pointeurs (voir chapitre 9).

2.1.4 Taille des types de base

L'espace qu'occupent les différents types en mémoire dépend de la machine sur laquelle est implanté le compilateur. Le choix est laissé aux concepteurs des compilateurs. Les seules contraintes sont des inégalités non strictes, à savoir :

- `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
- `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`

`sizeof` est un opérateur qui donne la taille en nombre d'octets du type dont le nom est entre parenthèses. Le tableau 2.1 donne les tailles des types dans le cas d'un PC. Les machines supportant un type `long double` différent du type `double` sont assez rares actuellement.

Tableau 2.1 – Longueur des types de base sur un processeur Intel i686.

Type	Taille (octets)
`char`	1
`short`	2
`int`	4
`long`	4
`float`	4
`double`	8
`long double`	8

2.2 Constantes associées aux types de base

Les constantes sont reconnues par le compilateur grâce à l'utilisation de caractères qui ne participent pas à la construction d'un identificateur.

2.2.1 Constantes de type entier

Les constantes de type entier sont construites à partir de chiffres. Elles sont naturellement exprimées en base dix mais peuvent être exprimées en base huit (octale) ou seize (hexadécimale).

Une constante est a priori du type `int` si le nombre qu'elle représente est plus petit que le plus grand entier représentable. Si la valeur de la constante est supérieure au plus grand entier représentable, la constante devient de type `long`.

Les constantes peuvent être suffixées par un « l » ou un « L » pour préciser que leur type associé est `long int`. Les constantes peuvent être précédées par un signe « - » ou « + ». Elles peuvent être suffixées par un « u » ou « U » pour préciser qu'elles expriment une valeur positive (qualifiées `unsigned`).

Voici quelques exemples de constantes de type entier :

- constante sans précision de type : 0377 octal, 0X0FF hexadécimal, 10 décimal, -20 décimal
- constante longue entière : 120L, 0364L, 0x1faL, 120l, 0364l, 0x1fal

- constante entière non signée : 120U, 0364U, 0x1faU, 120u, 0364u, 0x1fau
- constante longue entière non signée : 120UL, 0364UL, 0x1faUL, 120uL, 0364uL, 0x1fauL, 120Ul, 0364Ul, 0x1faUl, 120ul, 0364ul, 0x1faul

2.2.2 Constantes avec partie décimale

Les constantes avec partie décimale ont le type `double` par défaut. Elles peuvent être exprimées à partir d'une notation utilisant le point décimal ou à partir d'une notation exponentielle. Ces constantes peuvent être suffixées par un `f` ou `F` pour préciser qu'elles représentent une valeur de type `float`. Elles peuvent de même être suffixées par un `l` ou un `L` pour exprimer des valeurs de type `long double`.

Voici quelques constantes avec partie décimale :

- **121.34** constante exprimée en utilisant le point décimal, son type implicite est `double`.
- **12134e-2** constante exprimée en notation exponentielle.
- **121.34f** constante de valeur identique mais de type `float` car suffixée par `f`.
- **121.34l** constante de valeur identique mais de type `long double` car suffixée par `l`.

2.2.3 Constantes caractère

Les constantes du type caractère simple sont toujours entourées d'apostrophes (simples quotes aiguës). En général, lorsque le caractère est disponible au clavier, le caractère correspondant est donné directement, par exemple `'a'`. Certains caractères du clavier ne sont pas disponibles directement et il convient d'utiliser les notations suivantes[4] :

`'\\'` : Barre de fraction inversée ;

`'\"'` : Apostrophe ;

`'\t'` : Tabulation horizontale (HT) ;

`'\b'` : Backspace (BS) ;

Un certain nombre d'abréviations est également disponible :

`'\a'` : Alerte ou sonnerie (BEL) ;

`'\f'` : Saut de page (FF) ;

`'\n'` : Fin de ligne (LF) ;

`'\0'` : Fin de chaîne de caractères ;

`'\r'` : Retour chariot (CR) ;

`'\v'` : Tabulation verticale (VT) ;

2.3 Variables de base : déclaration et initialisation

En langage C, une variable de type entier notée `i` se déclare par `int i ;` et s'initialise à la valeur 3 grâce à l'instruction `i = 3 ;`. De même, une variable de type caractère se déclare par `char c ;` et s'initialise grâce à l'instruction `c = 'd' ;` ou bien `c = 100 ;`[5]. Nous pourrons déclarer et initialiser une variable en une seule instruction : `int i = 5 ;` ou `char c = '\t' ;`.

Une définition de variable a les rôles suivants :

Définir : le domaine de valeur de cette variable et les opérations légales sur cette variable (grâce au type) ;

4. — Attention, même si la notation de ces caractères particuliers semble contenir deux caractères (une barre inversée et un caractère), il faut bien considérer qu'il n'y en a qu'un seul !

5. — Le nombre 100 correspond au code ASCII décimale du caractère `'d'`.

Réserver : l'espace mémoire nécessaire lors de l'exécution au support de la variable (grâce au type et à la classe mémoire) ;

Initialiser : la variable à l'aide d'une constante dont le type correspond à celui de la variable ;

Déterminer : la durée de vie de la variable et permettre l'utilisation dans certaines parties du programme (grâce à la classe mémoire).

Une définition de variable est l'association d'un identificateur à un type et la spécification d'une classe mémoire.

2.3.1 Classe mémoire

La classe mémoire sert à expliciter la visibilité d'une variable et son implantation en machine. Nous approfondirons les différentes possibilités associées à ces classes mémoire dans le chapitre 8 sur la visibilité. Les classes mémoire sont :

global cette classe est celle des variables définies en dehors d'une fonction. Ces variables sont accessibles à toutes les fonctions. La durée de vie des variables de type `global` est la même que celle du programme en cours d'exécution.

local ou auto cette classe comprend l'ensemble des variables déclarées dans un bloc d'instructions. C'est le cas de toute variable déclarée à l'intérieur d'une fonction. L'espace mémoire réservé pour ce type de variable est alloué dans la pile d'exécution. C'est pourquoi elles sont appelées aussi `auto` c'est-à-dire automatique car l'espace mémoire associé est créé lors de l'entrée dans la fonction et il est détruit lors de la sortie de la fonction. La durée de vie des variables de type `local` est celle de la fonction dans laquelle elles sont définies.

static ce prédicat modifie la visibilité de la variable, ou son implantation :

- dans le cas d'une variable locale il modifie son implantation en attribuant une partie de l'espace de mémoire globale pour cette variable. Une variable locale de type `static` a un nom local mais a une durée de vie égale à celle du programme en cours d'exécution.
- dans le cas d'une variable globale, ce prédicat restreint la visibilité du nom de la variable à l'unité de compilation. Une variable globale de type `static` ne peut pas être utilisée par un autre fichier source participant au même programme par une référence avec le mot réservé `extern` (voir point suivant).

extern ce prédicat permet de spécifier que la ligne correspondante n'est pas une tentative de définition mais une déclaration. Il précise les variables globales (noms et types) qui sont définies dans un autre fichier source et qui sont utilisées dans ce fichier source.

register ce prédicat permet d'informer le compilateur que les variables locales définies dans le reste de la ligne sont utilisées souvent. Le prédicat demande de les mettre si possible dans des registres disponibles du processeur de manière à optimiser le temps d'exécution. Le nombre de registres disponibles pour de telles demandes est variable selon les machines. Seules les variables locales peuvent être déclarées `register`.

2.3.2 Qualificatifs des variables

Nous avons déjà parlé des qualificatifs `unsigned` et `signed` qui s'appliquent aux variables de type entier. Il existe deux autres qualificatifs qui ont été spécifiés par la norme. Il s'agit de `const` et `volatile`.

Une définition de variable qualifiée du mot `const` informe le compilateur que cette variable est considérée comme constante et ne doit pas être utilisée dans la partie gauche d'une affectation. Ce type de définition autorise le compilateur à placer la variable dans une zone mémoire accessible en lecture seulement à l'exécution. Exemple : `const double PI = 3.14 ;`. Notez qu'une variable de type

`const` doit être initialisée au moment de sa déclaration. Il ne sera plus possible ensuite de modifier cette valeur.

Le qualificatif `volatile` informe le compilateur que la variable correspondante est placée dans une zone de mémoire qui peut être modifiée par d'autres parties du système que le programme lui-même. Ceci supprime les optimisations faites par le compilateur lors de l'accès en lecture de la variable. Ce type de variable sert à décrire des zones de mémoire partagées entre plusieurs programmes ou encore des espaces mémoires correspondant à des zones d'entrée-sorties de la machine. Exemple : `volatile char c ;`.

Les deux qualificatifs peuvent être utilisés sur la même variable, spécifiant que la variable n'est pas modifiée par la partie correspondante du programme mais par l'extérieur.

2.4 Types dérivés

Un type dérivé est créé à partir de types standards pour l'usage propre à un programme. Les types dérivés sont principalement :

- les pointeurs,
- les chaînes de caractères,
- les tableaux et
- les structures, les unions et les énumérations.

Nous allons examiner en détails les deux premiers types dérivées. Les tableaux et les structures seront examinés dans les chapitres 9 et 10.

2.4.1 Pointeurs

Le pointeur est une variable destinée à contenir une adresse mémoire. Un pointeur est associé à un type d'objet et est reconnu par l'emploi d'un `*` lors de sa définition. Ce type est utilisé en particulier lors des calculs d'adresse qui permettent de manipuler des tableaux à partir de pointeurs (voir chapitre 9).

Prenons les définitions suivantes : `int* ptint ; char* ptchar ;`. Dans cet exemple, `ptint` est une variable du type pointeur sur un entier. Cette variable peut donc contenir des[6] valeurs qui sont des adresses de variables du type entier (`int`). De même, `ptchar` est une variable du type pointeur sur un caractère. Elle peut donc contenir des valeurs qui sont des adresses de variables de type caractère (`char`).

Le compilateur C vérifie le type des adresses mises dans un pointeur. Le type du pointeur conditionne les opérations arithmétiques sur ce pointeur. Les opérations les plus simples sur un pointeur sont les suivantes :

- affectation d'une adresse au pointeur ;
- utilisation du pointeur pour accéder à l'objet dont il contient l'adresse.

Considérons les variables suivantes `int in ; char car ; int* ptint ; char* ptchar ;`. Un pointeur peut être affecté avec l'adresse d'une variable ayant un type qui correspond à celui du pointeur : `ptint = &in ; ptc = &car ;`.

Une fois un pointeur affecté avec l'adresse d'une variable, ce pointeur peut être utilisé pour accéder aux cases mémoires correspondant à la variable (valeur de la variable) : `*ptint = 12 ; *ptc = 'a' ;`. La première instruction met la valeur entière `12` dans l'entier `in`, la deuxième instruction met le caractère `'a'` dans l'entier `car`.

6. — Une valeur à la fois mais comme le pointeur est une variable cette valeur peut changer au cours de l'exécution du programme.

2.4.2 Les chaînes de caractères

Les constantes du type chaîne de caractères doivent être mises entre guillemets (double quote). Le compilateur génère une suite d'octets terminée par un caractère nul ('\0') à partir des caractères contenus dans la chaîne. Une chaîne de caractères est en fait un tableau de `char`. La chaîne est référencée par l'adresse du tableau (i.e. l'adresse du premier élément du tableau, cf. chapitre 9). Par exemple, la chaîne de caractères « message » est générée par le compilateur selon le schéma du tableau 2.2.

Tableau 2.2 – Chaîne de caractères constante

m	e	s	s	a	g	e	\0

La déclaration d'une variable `ch` désignant une chaîne de 10 caractères s'écrit : `char ch[10] ;`. Nous pouvons écrire au plus 9 caractères, sachant que le 10^{e} emplacement est réservé pour le caractère de terminaison de chaîne `'\0'`.

Il est impossible d'utiliser les opérateurs sur les chaînes de caractères. Notamment, il est interdit de concaténer deux chaînes de caractères en utilisant l'opérateur `+`. De même, il n'est pas possible de copier une chaîne dans une seconde en utilisant le signe `=`. Pour obtenir ce résultat, il faut travailler non pas au niveau de la chaîne de caractères, mais au niveau des caractères qui la compose (en programmant des boucles, comme nous le verrons par la suite). Néanmoins, la norme du langage C a prévu un certain nombre d'outils (c'est à dire de « fonctions », notion étudiée au chapitre 6) permettant de manipuler avec aisance les chaînes de caractères. Ces outils font partie de la bibliothèque standard du C (cf. chapitre 12).

Il est néanmoins possible d'initialiser une chaîne de caractères constante de la manière suivante : `char ch[50] = "Bonjour" ;`. Dans cet exemple, nous avons réservé un espace mémoire de 50 caractères. Les 7 premiers espaces sont utilisés pour les 7 caractères du mot, le 8^{e} espace est utilisé pour le caractère de terminaison de chaîne. Les 42 espaces restant sont non utilisés (et donc perdus !). D'autres exemples d'initialisation sont présentés dans le tableau 2.3.

Tableau 2.3 – Exemples d'initialisation de chaînes.

`char t1[10] = "Coucou" ;`	Tableaux de 10 caractères initialisé avec `'c'`, `'o'`, `'u'`, `'c'`, `'o'`, `'u'`, `'\0'`. Les trois derniers caractères sont eux aussi initialisés avec `'\0'`.
`char t2[] = "bonjour" ;`	Tableau de caractères initialisé avec la chaîne `"bonjour"`. La taille du tableau est calculée selon le nombre de caractères +1.
`char t3[10] = {'a', 'b', 'c'} ;`	Tableau de 10 caractères dont les 3 premiers éléments sont initialisés avec les lettres `'a'`, `'b'` et `'c'`.

2.5 Conversion de types

La conversion de type est un outil très puissant, elle doit donc être utilisée avec prudence. Le compilateur fait de lui-même des conversions lors de l'évaluation des expressions. Pour cela, il applique

des règles de conversion implicite. Ces règles ont pour but la perte du minimum d'information dans l'évaluation de l'expression.

Règle de conversion implicite Convertir les éléments de la partie droite d'une expression d'affectation dans le type de la variable ou de la constante le plus riche. Faire les opérations de calcul dans ce type. Puis convertir le résultat dans le type de la variable affectée (partie gauche de l'affectation).

La notion de richesse d'un type est précisée dans la norme. Le type dans lequel le calcul d'une expression à deux opérandes doit se faire est donné par les règles suivantes :

1. si l'un des deux opérandes est du type `long double` alors le calcul doit être fait dans le type `long double` ;
2. sinon, si l'un des deux opérandes est du type `double` alors le calcul doit être fait dans le type `double` ;
3. sinon, si l'un des deux opérandes est du type `float` alors le calcul doit être fait dans le type `float` ;
4. sinon, appliquer la règle de promotion en entier, puis :
   (a) si l'un des deux opérandes est du type `unsigned long int` alors le calcul doit être fait dans ce type ;
   (b) si l'un des deux opérandes est du type `long int` alors le calcul doit être fait dans le type `long int` ;
   (c) si l'un des deux opérandes est du type `unsigned int` alors le calcul doit être fait dans le type `unsigned int` ;
   (d) si l'un des deux opérandes est du type `int` alors le calcul doit être fait dans le type `int`.

La règle de promotion en entier précise que lorsque des variables ou des constantes des types suivants sont utilisées dans une expression, alors les valeurs de ces variables ou constantes sont transformées en leur équivalent en entier avant de faire les calculs. Ceci permet d'utiliser des caractères et des entiers courts de la même façon que des entiers.

Des exemples de conversions implicites sont donnés dans le tableau 2.4.

Tableau 2.4 – Exemples de conversion implicite.

`float f ; double d ; int i ; long li ;`	
`li = f + i ;`	`i` est transformé en `float` puis additionné à `f`, le résultat est transformé en `long` et rangé dans `li`.
`d = li + i ;`	`i` est transformé en `long` puis additionné à `li`, le résultat est transformé en `double` et rangé dans `d`.
`i = f + d ;`	`f` est transformé en `double` puis additionné à `d`, le résultat est transformé en `int` et rangé dans `i`.

Il est possible de forcer la conversion d'une variable (ou d'une expression) dans un autre type avant de l'utiliser par une conversion implicite. Cette opération est appelée « cast ». Elle se réalise de la manière suivante : `(type) expression`.

Prenons pour exemple l'expression : `i = (int) f + (int) d ;`. `f` et `d` sont convertis en `int`, puis additionnés. Le résultat entier est rangé dans `i`. Il peut y avoir une différence avec l'expression : `i = f + d ;`, du fait de la perte des parties fractionnaires des nombres.

Chapitre 3

Lire et écrire

Sommaire

Le langage C est utilisé dans un contexte interactif. Ce qui veut dire que la plupart des programmes écrits en langage C font des échanges d'information avec un utilisateur du programme.

Bien sûr, le langage C est un langage des années 70 et l'idée de l'interaction avec l'utilisateur est celle des systèmes centralisés à temps partagé. Un utilisateur de ce type de système est connecté via une voie d'entrée-sortie qui permet d'échanger des caractères. Ces voies sont la plupart du temps reliées à un télétype (écran, clavier, avec sortie optionnelle sur papier). Les caractères sont écrits sur l'écran du terminal et lus à partir du clavier.

Les entrée-sorties en langage C ne sont pas prises en charge directement par le compilateur mais elles sont réalisées à travers de *fonctions* de la bibliothèque `"stdio.h"` (librairie d'entrée/sortie standard). Le compilateur ne peut pas faire de contrôle de cohérence dans les arguments passés à ces fonctions car ils sont de type variable. Ceci explique l'attention toute particulière avec laquelle ces opérations doivent être programmées.

3.1 Exemple de lecture et écriture

LA fonction `scanf()` fait le pendant à la fonction `printf()`. `scanf()` permet de lire des valeurs à partir du clavier (entrée), alors que `printf()` permet d'afficher une valeur (sortie). Le programme 3.1 est un exemple de lecture et d'écriture d'une chaîne de caractères.

Prog. 3.1 – Lecture et écriture de chaîne par `scanf()` et `printf()`

```
#include <stdio.h>
char tt[80];    // Tableau de 80 caractères
int main() {
    printf("Ecrivez une chaine de caracteres : ");
    scanf("%s", tt);
    printf("\nLa chaine entree est : %s\n", tt);
    return 0;
}
```

Ce programme contient la définition d'une variable globale [1]. Cette variable est un tableau de quatre-vingt caractères destiné à recevoir les caractères lus au clavier. Les seules instructions sont les appels aux fonctions de lecture et d'écriture (`scanf()` et `printf()`). Remarquons que l'affichage de la chaîne dans le second `printf()` se fait grâce `%s` (`s` désigne le mot « string », soit « chaîne » en français).

3.2 Les fonctions `printf()` et `scanf()` en détail

Les fonctions `printf()` et `scanf()` transforment des objets d'une représentation à partir d'une chaîne de caractères (vision humaine) en une représentation manipulable par la machine (vision machine), et vice et versa. Pour réaliser ces transformations, ces fonctions sont guidées par des formats qui décrivent le type des objets manipulés (vision interne) et la représentation en chaîne de caractères cible (vision externe). Par exemple, un format du type `%x` signifie d'une part que la variable est du type entier et, d'autre part, que la chaîne de caractères qui la représente est exprimée en base `16` (notation hexadécimale). Autre exemple : `%d` signifie d'une part que la variable est du type entier et, d'autre part, que la chaîne de caractères qui la représente est exprimée en base `10` (notation décimale).

Pour `printf()` un format est une chaîne de caractères dans laquelle sont insérés les caractères représentant la ou les variables à écrire. Pour `scanf()`, un format est une chaîne de caractères qui décrit la ou les variables à lire. Pour chaque variable, un type de conversion est spécifié. Ce type de conversion est décrit par les caractères qui suivent le caractère `%`.

Dans une première approche de `scanf()`, nous considérerons qu'il ne faut mettre que des types de conversions dans le format de lecture. Le lecteur curieux peut se reporter à la section 11.5.

Le tableau 3.1 donne un résumé des déclarations de variables et des formats nécessaires à leurs manipulations pour `printf()` et `scanf()`. Pour `scanf()`, il faut le mettre devant le nom de la variable, sauf pour les variables du type tableau de caractères.

L'exemple 3.2 montre qu'il est possible de réaliser l'écriture ou la lecture de plusieurs variables en utilisant une seule chaîne de caractères contenant plusieurs descriptions de formats.

Prog. 3.2 – Lectures multiples avec `scanf()`

```
#include <stdio.h>
int main() {
  int i=10;
  float l=3.14159;
  char p[50]="Bonjour";

  printf("Apres␣lecture␣au␣clavier␣:␣%d␣%f␣%s\n",i,l,p);
  scanf("%d%f%s",&i,&l,p);
  printf("Apres␣lecture␣au␣clavier␣:␣%d␣%f␣%s\n",i,l,p);

  return 0;
}
```

En ce qui concerne `printf()`, un certain nombre de caractères optionnels peuvent être insérés entre le symbol `%` et le caractères spécifiant la conversion (`d`, `x`, `e`, ...). Par exemple :

- le signe `-` pour demander un cadrage à gauche, au lieu du cadrage à droite (par défaut).

1. — ... dans le sens où la variable `tt` est définie en dehors de toute fonction. La plupart du temps, nous utiliserons des variables locales, c'est-à-dire des variables définies à l'intérieur d'une fonction.

Tableau 3.1 – Exemples de `printf()` et `scanf()`.

Déclaration	Lecture	Écriture	Format externe
`int i ;`	`scanf("%d",&i) ;`	`printf("%d",i) ;`	décimal
`int i ;`	`scanf("%o",&i) ;`	`printf("%o",i) ;`	octal
`int i ;`	`scanf("%x",&i) ;`	`printf("%x",i) ;`	hexadécimal
`unsigned int i ;`	`scanf("%u",&i) ;`	`printf("%u",i) ;`	décimal
`short j ;`	`scanf("%hd",&j) ;`	`printf("%d",j) ;`	décimal
`short j ;`	`scanf("%ho",&j) ;`	`printf("%o",j) ;`	octal
`short j ;`	`scanf("%hx",&j) ;`	`printf("%x",j) ;`	hexadécimal
`unsigned short j ;`	`scanf("%hu",&j) ;`	`printf("%u",j) ;`	décimal
`long k ;`	`scanf("%ld",&k) ;`	`printf("%ld",k) ;`	décimal
`long k ;`	`scanf("%lo",&k) ;`	`printf("%lo",k) ;`	octal
`long k ;`	`scanf("%lx",&k) ;`	`printf("%lx",k) ;`	hexadécimal
`unsigned long k ;`	`scanf("%lu",&k) ;`	`printf("%lu",k) ;`	décimal
`float l ;`	`scanf("%f",&l) ;`	`printf("%f",l) ;`	point décimal
`float l ;`	`scanf("%e",&l) ;`	`printf("%e",l) ;`	exponentielle
`float l ;`		`printf("%g",l) ;`	la + courte
`double m ;`	`scanf("%lf",&m) ;`	`printf("%lf",m) ;`	point décimal
`double m ;`	`scanf("%le",&m) ;`	`printf("%le",m) ;`	exponentielle
`double m ;`		`printf("%lg",m) ;`	la + courte
`long double n ;`	`scanf("%Lf",&n) ;`	`printf("%Lf",n) ;`	point décimal
`long double n ;`	`scanf("%Le",&n) ;`	`printf("%Le",n) ;`	exponentielle
`long double n ;`		`printf("%Lg",n) ;`	la plus courte
`char o ;`	`scanf("%c",&o) ;`	`printf("%c",o) ;`	caractère
`char p[10] ;`	`scanf("%s",p) ;`	`printf("%s",p) ;`	chaîne de caractères

- un nombre indiquant la taille minimale en caractères du champs à imprimer. Les « espaces » jouant le rôle de caractère de remplissage.
- un point décimal (virgule flottante), suivi d'un nombre donnant la précision de la partie fractionnaire, c'est à dire le nombre de chiffre significatifs après le point. Si la donnée n'est pas du type flottant, ce nombre représente la taille maximale du champs à imprimer.

Voici quelques exemples :

`%8d` : Imprime un nombre en décimal cadré à droite dont la longueur du champ imprimable est de huit caractères. Des espaces de remplissage précédent le nombre.

`%-25s` : Imprime une chaîne de caractères cadrée à gauche assurant une longueur minimum de 25 caractères.

`%.6f` : Imprime un nombre flottant avec un maximum de six chiffres significatifs.

En ce qui concerne `scanf()`, une option intéressante est la suivante. Le caractère `*` précédé de `%` spécifie que la valeur lue sera sautée, donc non affectée à la variable suivante. Exemple : `scanf("%d %*s %d %*s %d %*s", &heure, &minutes, &secondes) ;` L'entrée des données au clavier pourrait se présenter de la forme : `17 H 35 min 30 secondes`. Les chaînes de caractères « H », « min » et « secondes » seront alors ignorées.

Chapitre 4

Opérateurs et expressions

Sommaire

Le langage C est connu pour la richesse de ses *opérateurs* et il apporte quelques notions innovantes. En particulier, le langage C considère l'affectation comme un opérateur normal alors que la plupart des langages la considèrent comme une opération privilégiée. Cette richesse permet d'écrire des expressions (combinaisons d'opérateurs et d'opérandes) parfois complexes. Les opérateurs sont les éléments du langage qui permettent de faire du calcul ou de définir des relations. Ils servent à combiner des variables et des constantes pour réaliser des *expressions*.

L'organisation de ce chapitre est guidée par le nombre d'opérandes mis en cause par l'opérateur et non par l'utilisation des opérateurs.

4.1 Opérateurs unaires

Un opérateur unaire agit sur un opérande qui peut-être une constante, une variable, ou une expression. Ainsi, l'opérateur unaire `-` permet d'inverser le signe et on peut écrire :

`-2` où `2` est une constante ;

`-i` où `i` est une variable ;

`-(i+2)` où `i+2` est une expression.

Le tableau 4.1 donne la liste des opérateurs unaires. Nous allons prendre quelques exemples pour expliquer l'utilisation de base de ces opérateurs sur les variables : `int var=10, *pint=&var, nvar=0 ;` et `long f=20L ;`.

Tableau 4.1 – Liste des opérateurs unaires.

Opérateur	Utilisation
`&`	opérateur d'adresse
`*`	opérateur d'indirection sur une adresse
`--`	opérateur de décrémentation
`++`	opérateur d'incrémentation
`sizeof`	opérateur donnant la taille en octet
`!`	non logique, il sert à inverser une condition
`-`	moins unaire : inverse le signe
`+`	plus unaire : sert à confirmer

4.1.1 Opérateur d'adresse et d'indirection

Ces deux opérateurs, noté `*` et `&` ont déjà été étudiés dans la section 2.4 du chapitre 2.

Le `&` est l'opérateur d'adresse, il retourne l'adresse de la variable suivant le `&`. `&var` donne l'adresse de la variable `var`. Cette adresse peut être utilisée pour affecter un pointeur (à la condition que le pointeur soit d'un type compatible avec l'adresse de la variable) : `int* pint = &var ;`

Le `*` est l'opérateur d'indirection. Il permet d'accéder à une variable à partir d'une adresse (souvent contenue dans un pointeur). `*&var` donne la valeur `10`, de même que `*pint`, puisque `pint` a été initialisé à partir de l'adresse de `var`.

4.1.2 Opérateur d'incrémentation et de décrémentation

Les opérateurs `--` et `++` permettent de décrémenter et d'incrémenter des variables de type entier. D'une manière simpliste, nos pouvons considérer que :

- `var--` est équivalent à `var = var - 1 ;`
- `var++` est équivalent à `var = var + 1`.

Il est possible d'utiliser les opérateurs unaires d'incrémentation et de décrémentation après ou avant la variable. Ce qui permet de post-incrémenter, de pré-incrémenter, de post-décrémenter ou de pré-décrémenter. Lorsque l'opérateur est préfixé, l'opération est appliquée avant que la valeur correspondant à l'opération ne soit calculée. Dans le cas où l'opération est post-fixée, la valeur de la variable avant l'opération est utilisée pour les autres calculs et ensuite l'opération est appliquée.

Prenons comme exemple, deux entiers `i` et `j`, et initialisons ces deux variables avec la valeur `0` : `int i=0, j=0 ;`. Si nous écrivons `j = ++i`, c'est une pré-incrémentation de la variable `i`. Cela signifie incrémenter `i` de `1` puis mettre la valeur de `i` dans `j`. À la fin de cette opération `i` vaut `1` et `j` vaut `1`. En anticipant sur l'opérateur de succession 4.2.4, nous pouvons considérer cette opération comme équivalente à `i = i+1 ; j = i ;`. Si au contraire, nous écrivons `j = i++ ;`, cela signifie mettre la valeur de `i` dans `j` puis incrémenter `i` de `1`. En partant des mêmes valeurs, `i` valant `0` et `j` valant `0`, à la fin de cette opération `i` vaut `1` et `j` vaut `0`. Cette opération est équivalente à `j = i ; i = i+1 ;`.

4.1.3 Opérateur de dimension

`sizeof` donne la taille en octets de la variable. Exemple : `long int f ; sizeof(f) ;` donne la valeur `4` [1]. L'opérateur `sizeof` peut aussi donner la taille d'un type, le type doit être entre parenthèses. Dans notre exemple, `sizeof(f)` est équivalent à `sizeof(long)`.

Les calculs associés à l'opérateur `sizeof()` sont réalisés par le compilateur lors de la traduction du langage en assembleur, et non lors de l'exécution du programme. L'expression `sizeof(f)` est donc une valeur constante et peut entrer dans la construction d'une expression constante (i.e. calculable lors de la compilation).

4.1.4 Non logique, plus et moins unaires

Le non logique sert à inverser une condition de vrai à faux et réciproquement. En langage C, une expression est fausse si la valeur qu'elle retourne est égale à `0`, elle est vraie sinon. De plus, la norme spécifie que `!0` vaut `1`. Dans notre exemple, `!var` vaut `0` car `var` est vraie puisque `var` contient `10`.

Le moins unaire inverse le signe de l'expression qui le suit. Le plus unaire sert à confirmer le signe de l'expression.

4.2 Opérateurs binaires

Le tableau 4.2 donne la liste des opérateurs binaires.

Tableau 4.2 – Liste des opérateurs binaire.

Type d'opérateurs	Opérateurs	Usage

Arithmétique	`+ -`
	`* /`
	`%`
		addition, soustraction,
		multiplication, division,
		reste de la division entière.

Masquage	`& | ∧ ~`	et, ou, ou exclusif, complément à 1.
Décalage	`» «`	vers la droite ou vers la gauche.

Relation	`< <=`
	`> >=`
	`== !=`
		inférieur, inférieur ou égal,
		supérieur, supérieur ou égal,
		égal, non égal.

Logique	`&& ||`	et logique, ou logique.
Affectation	cf. tableau 4.3	affectation.
Succession	`,`	succession.

4.2.1 Opérateurs arithmétiques

Le langage C permet l'utilisation des opérateurs de calcul que l'on trouve habituellement dans les autres langages, à savoir : l'addition, la soustraction, la multiplication et la division. Il utilise pour cela les symboles respectifs : `+ - * /`.

Comme nous avons déjà vu les opérateurs unaires, vous remarquerez l'utilisation contextuelle dans le cas des trois symboles : `+ - *`. Le compilateur détermine la signification de l'opérateur à son nombre d'opérandes.

1. — Un entier `long` sur les machines les plus courantes est représenté sur `4` octets, soit `32` bits.

Comme nous le verrons plus loin, le type (au sens type des données) de l'opération est déterminé par le type des valeurs sur lesquelles portent l'opération. Les opérations arithmétiques classiques peuvent s'appliquer aux types entiers et dans ce cas elles ont un comportement d'opération entière (en particulier la division). Ces opérations s'appliquent aussi aux types avec partie décimale et dans ce cas, elles donnent un résultat avec partie décimale.

Le langage C introduit l'opérateur modulo, noté `%`, qui permet d'obtenir le reste de la division entière déterminée par les deux opérandes associés au `%`. Par exemple, l'expression `14 % 3` donne la valeur `2`.

4.2.2 Opérateurs de manipulation de bits

Ces opérateurs servent à manipuler des mots bit à bit. Les opérandes doivent être de type discret. On distingue les opérateurs de masquage des opérateurs de décalage.

Opérateur de masquage :

- `&` : `a & b`, ET logique bit à bit (AND),
- `|` : `a | b`, OU logique inclusif bit à bit (OR),
- ∧ : `a ∧ b`, OU logique exclusif bit à bit (XOR),
- `~` : `a ~ b`, complément à 1, les bits sont inversés 1–>0 et 0–>1.

Opérateur de décalage :

- `«` : `a « b`, Décalage vers la gauche (en nombre de bits),les bits de poids fort disparaissent tandis que des 0 par la droite.
- `»` : `a » b`, Décalage vers la droite (en nombre de bits),les bits de poids faible disparaissent tandis que des 0 par la gauche (quand le nombre est négatif, sur certaines machines, c'est un 1 qui apparaît).

4.2.3 Opérateurs booléens

Les opérateurs de relation servent à réaliser des tests entre les valeurs de deux expressions. Comme nous le verrons dans le chapitre 5, ces opérateurs sont surtout utilisés à l'intérieur des instructions de contrôle (tests).

Les opérateurs de relation algébrique sont au nombre de six : `<`, `<=`, `>`, `>=`, `==` et `!=`. Ces opérateurs peuvent être utilisés avec des variables de type entier ou des variables ayant une partie décimale. Notez le double égal pour le test d'égalité qui est souvent source de confusion avec le simple égal qui décrit l'affectation.

Les deux autres opérateurs de tests (`&& et ||`) sont appelés respectivement le « et logique » et le « ou logique ». Le « et logique » permet de décrire qu'une condition constituée de deux parties est satisfaite si et seulement si les deux parties sont satisfaites. Le « ou logique » permet de décrire qu'une condition constituée de deux parties est satisfaite dés lors qu'une des deux parties au moins est satisfaite. Ces opérateurs s'appliquent à des expressions que l'on peut considérer comme de type entier et dont la valeur est testée comme pour le non logique à savoir une expression est considérée comme fausse si la valeur correspondante à cette expression est égale à `0`.

Ainsi en langage C, pour tester si une variable de type entier `j` contient une valeur comprise entre deux bornes non strictes `12` et `143` on écrit : `(j >= 12) && (j <= 143)`. De même, un test pour savoir si un caractère `car` contient un `'a'` en minuscule ou en majuscule s'écrit : `(car == 'a') || (car == 'A')`.

4.2.4 Opérateur d'affectation et de succession

En langage C, l'affectation est un opérateur comme les autres. Ceci permet d'écrire des expressions comme : `i = j = k = 1` qui détermine une affectation multiple.

Le langage C permet de construire des opérateurs binaires d'affectation à partir des opérateurs binaires arithmétiques, des opérateurs de masquage et des opérateurs de décalage, en les faisant suivre d'un égal (=). Ceci donne les opérateurs décrits dans le tableau 4.3.

Tableau 4.3 – Liste des opérateurs binaires d'affectation.

Arithmétique	`+= -= *= /= %=`
Masquage	`&= |= ∧=`
Décalage	`»= «=`

Le tableau 4.4 donne un exemple d'utilisation de chacun de ces opérateurs et la façon de lire ces différentes expressions. Ce tableau est construit en supposant le type entier sur `32` bits, et les deux variables `i` et `j` définies de la manière suivante : `int i = 100, j = 5;`.

Tableau 4.4 – Exemple d'opérateurs binaires d'affectation.

Arithmétique	**Résultat**	**Équivalence**	**Lecture**
Opérateurs arithmétiques			
`i += 10`	`110`	`i = i + 10`	ajoute `10` à `i`
`i += j`	`115`	`i = i + j`	ajoute `j` à `i`
`i -= 5`	`110`	`i = i - 5`	retranche `10` à `i`
`i -= j`	`105`	`i = i - j`	retranche `10` à `i`
`i *= 10`	`1050`	`i = i * 10`	multiplie `i` par `10`
`i *= j`	`5250`	`i = i * j`	multiplie `i` par `j`
`i /= 10`	`525`	`i = i / 10`	divise `i` par `10`
`i /= j`	`105`	`i = i / j`	divise `i` par `j`
`i %= 10`	`5`	`i = i % 10`	`i` reçoit le reste de la division entière de `i` par `10`
Opérateurs de masquage			
`i &= 8`	`0`	`i = i & 8`	ET de `i` avec `8`
`i |= 8`	`8`	`i = i | 8`	OU de `i` avec `8`
`i ∧= 8`	`0x0C`	`i = i ∧ 4`	OU exclusif de `i` avec `4`
Opérateurs de décalage			
`i «= 4`	`0xC0`	`i = i « 4`	décale `i` à gauche de `4` positions
`i »= 4`	`0x0C`	`i = i » 4`	décale `i` à droite de `4` positions

La partie droite de l'opérateur peut être une expression : `i += ( j * 25 + 342 )`. Ceci permet d'écrire des expressions plus complexes : `i += ( j += j * 25 + 342 ) - 12`. La dernière expression arithmétique fait plusieurs affectations :

- celle de `j` avec `j + j * 25 + 342;`
- celle de `i` avec `i + j - 12`

Soit, si `i` et `j` ont pour valeur `1` avant cette expression, `j` vaudra `368` après et `i` vaudra `357`. Ce type d'expression est un peu compliqué pour favoriser une bonne lisibilité du programme. Les possibilités offertes en matière d'expressions sont de ce fait peu utilisées.

La virgule, quant à elle, sert à séparer deux expressions qui sont évaluées successivement. La valeur associée sera la dernière valeur calculée. Une expression comme : `i = (j=2, k=3)` associe à `i` la valeur `3`.

4.3 Opérateur ternaire

L'opérateur `? :` est un opérateur ternaire dont la syntaxe est résumé par :

`expression1 ? expression2 : expression3 ;`

Si `expression1` rend la valeur vraie (l'entier 0) alors `expression2` est évaluée et le résultat est celui d'`expression2`, sinon c'est `expression3` qui est évaluée et le résultat est celui d'`expression3`. L'exemple `c = a<b ? a : b ;` calcule le minimum des nombres `a` et `b`, et place le résultat dans `c`.

4.4 Précédence des opérateurs

Les opérateurs sont liés par des relations de précédence qu'il faut connaître. Cette précédence détermine l'ordre d'évaluation de l'expression par le compilateur.

Règle de précédence : La priorité des opérateurs du langage C est décroissante de haut en bas selon le tableau 4.5. Lorsque deux opérateurs se trouvent dans la même ligne du tableau 4.5, la priorité d'évaluation d'une ligne de C dans laquelle se trouvent ces opérateurs, est donnée par la colonne de droite (associativité).

Tableau 4.5 – Précédence des opérateurs (priorité décroissante de haut en bas).

Précédence des opérateurs		
Classe d'opérateur	*Opérateurs*	*Associativité*
Parenthèsage	`()`	de gauche à droite
Suffixes	`[] -> . ++ -`	de gauche à droite
Unaires	`& * + - ! sizeof ~`	de droite à gauche
Changement de type	`(type)`	de droite à gauche
Multiplicatifs	`* / %`	de gauche à droite
Additifs	`+ -`	de gauche à droite
Décalage	`« »`	de gauche à droite
Comparaisons	`< <= > >=`	de gauche à droite
Égalités	`== !=`	de gauche à droite
ET bit à bit	`&`	de gauche à droite
OU exclusif bit à bit	`∧`	de gauche à droite
OU bit à bit	`|`	de gauche à droite
ET logique	`&&`	de gauche à droite
OU logique	`||`	de gauche à droite
Condition	`? :`	de droite à gauche
Affectations	`= += -= *= /= &= |= ∧= «= »=`	de droite à gauche
Succession	`,`	de gauche à droite

Il faut consulter ce tableau pour être sûr de l'ordre d'évaluation d'une expression.

L'associativité n'est pas la précédence entre opérateurs d'une même ligne de ce tableau. C'est la façon dont le compilateur analyse la ligne source en C. C'est l'ordre dans la ligne source qui est important

et non l'ordre sur la ligne du tableau.

Pour résumer, cette règle de précédence suit l'ordre :

- parenthèsage ;
- opérateurs d'accès, appel de fonction et post incrémentation ou décrémentation ;
- opérateurs unaires (associativité de droite à gauche) ;
- opérateurs binaires (méfiez vous des relations opérateurs de test et opérateurs bit à bit) ;
- opérateur ternaire (associativité de droite à gauche) ;
- opérateurs binaires d'affectation (associativité de droite à gauche) ;
- succession.

Chapitre 5

Instructions de contrôle

Sommaire

On distingue les instructions conditionnelles ou de sélection (instructions de test et d'aiguillage), les instructions de répétition ou d'itération, et les instructions de rupture de séquence.

5.1 Instructions conditionnelles ou de sélection

Les instructions conditionnelles permettent de réaliser des tests, et suivant le résultat de ces tests, d'exécuter des parties de code différentes.

5.1.1 Test

L'opérateur de test se présente sous les deux formes présentées dans le tableau 5.1.

Tableau 5.1 – Syntaxes du `if`.

1	`if` (expression) instruction
2	`if` (expression) instruction1 `else` instruction2

Comme nous l'avons déjà vu : une expression est « vraie » si le résultat est non nul, elle est « fausse » si le résultat est nul.

L'expression est évaluée. Si le résultat est différent de 0, alors l'instruction qui suit le `if` est exécutée, et on n'exécute pas la partie du `else`. Si l'expression rend un résultat égal à 0, c'est l'instruction qui suit le `else` (si elle existe) qui est exécutée.

Voici quelques exemples de tests en langage C :

`if (a == b) ...`	usuel comme dans tous les autres langages
`int b = 1;` `if (a = b) ...`	est vrai puisque `b` est égal à `1` (donc non nul), attention car cela met la valeur de `b` dans `a`.

`int c;`
`if (c = getchar()) ...`
`if (c) ...`
	vrai si la fonction `getchar` ne ramène pas `'\0'`.
	vrai si `c` n'est pas égal à `0` sinon faux.

Les tests sont souvent imbriqués :

```
if (n>0)
  printf("n=\%d est positif", n);
else
  if (n<0)
    printf("n=\%d est negatif", n);
  else
    printf("n=\%d est nul", n);
```

Le programme 5.1 propose un autre exemple très simple qui compare deux nombres `a` et `b`, et regarde si deux autres nombres `c` et `d` sont donnée dans le même ordre.

Prog. 5.1 – Premier exemple de test

```
int a, b, c, d, result;
a=1; b=4; c=5; d=6;
result = (a<b) == (c<d);
if (a<b)
  printf("%d est inférieur à %d\n", a,b);
if (result)
  printf("\n%d et %d sont dans le meme ordre\n", c,d);
```

Le second exemple d'utilisation de `if` est présenté dans le programme 5.2.

Prog. 5.2 – Second exemple de test

```
printf("Voulez vous le mode d'emploi ?\n");
char c = getchar();
if (c=='o')
  printf("\nDesole c'est une farce!");
else
  if (c=='n')
    printf("\nTant mieux car il n'y en a pas !");
  else
    printf("Vous auriez du repondre par o ou n");
```

5.1.2 Table de branchement ou d'aiguillage

Pour éviter les imbrications d'instructions `if`, le C possède une instruction qui crée une table de branchement : c'est l'instruction `switch`. Le `switch` est une instruction de choix multiple qui effectue un aiguillage direct vers des instructions en fonction d'une valeur discrète résultat de l'évaluation de l'expression. La syntaxe du `switch` est résumée dans le tableau 5.2.

Tableau 5.2 – Syntaxe classique du `switch`.

```
switch  (expression)     {
        case value1 :    inst 10
                         inst 11
                         break ;
        case value2 :    inst 20
                         inst 21
                         break ;
        case valueN :    inst N0
                         inst N1
                         break ;
        default :        inst d0
                         inst d1
                         break ;
}
```

L'exécution du `switch` est réalisée selon les règles et les étapes suivantes :

1. l'expression est évaluée comme une valeur entière ;
2. les valeurs des `case` sont évaluées comme des constantes entières ;
3. l'exécution se fait à partir du `case` dont la valeur correspond à l'expression. Elle s'exécute en séquence jusqu'à la rencontre d'une instruction `break ;` ;
4. les instructions qui suivent la condition `default` sont exécutées lorsqu'aucune constante des `case` n'est égale à la valeur retournée par l'expression ;
5. l'ordre des `case` et du `default` n'est pas pré-défini par le langage mais par les besoins du programme ;
6. l'exécution à partir d'un `case` continue sur les instructions des autres `case` tant qu'un `break` n'est pas rencontré ;
7. plusieurs valeurs de `case` peuvent aboutir sur les mêmes instructions ;
8. le dernier `break` est facultatif. Il vaut mieux le laisser pour la cohérence de l'écriture, et pour ne pas avoir de surprise lorsqu'un `case` est ajouté.

Dans l'exemple 5.3, si vous omettez les instructions `break`, vous imprimerez l'ensemble de tous les messages dans le cas où un caractère `'a'` a été frappé.

Prog. 5.3 – Premier exemple de `switch`.

```
char c;
printf("Entrez une voyelle : ");
c=getchar();
switch(c) {
    case 'a' : case 'A' :
        printf("C'est un a\n");
        break;
    case 'e' : case 'E' :
        printf("C'est un e\n");
        break;
    case 'i' : case 'I' :
        printf("C'est un i\n");
        break;
    case 'o' : case 'O' :
        printf("C'est un o\n");
        break;
    case 'u' : case 'U' :
        printf("C'est un u\n");
        break;
    case 'y' : case 'Y' :
        printf("C'est un y\n");
        break;
    default :
        printf("Ce n'est pas une voyelle\n");
        break;
}
```

Prog. 5.4 – Second exemple de `switch`.

```
switch(i) {
  case 6:
  case 8:  printf("le nombre est superieur a 5\n");
  case 0:
  case 2:
  case 4:  printf("le nombre est pair\n");
           break;
  case 9:
  case 7:  printf("le nombre est superieur a 5\n");
  case 5:
  case 1:
  case 3:  printf("le nombre est impair\n");
           break;
  default: printf("ceci n'est pas un nombre\n");
           break;
}
```

Dans l'exemple 5.4 :

1. lorsque `i` est égal à `6` ou `8`, le programme écrit « le nombre est supérieur a `5` », puis écrit « le nombre est pair » ;
2. lorsque `i` est égal à `0`, `2` ou `4`, le programme écrit « le nombre est pair » ;
3. lorsque `i` est égal à `9` ou `7`, le programme écrit « le nombre est supérieur a `5` », puis écrit « le nombre est impair » ;

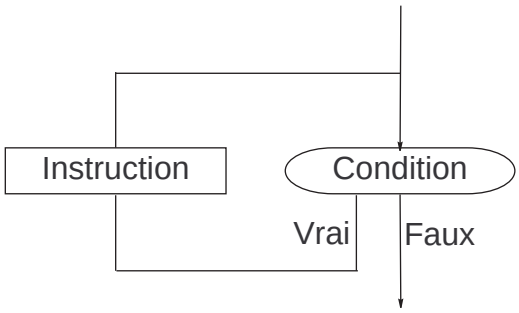

Figure 5.1 – Organigramme du `while`.

4. lorsque `i` est égal à `1`, `3` ou `5`, le programme écrit le nombre est impair ;
5. dans les autres cas, le programme écrit « ceci n'est pas un nombre ».

5.2 Instructions de répétition ou d'itération

Les instructions répétitives sont commandées par trois types de boucles :
- le `while`
- le `for`
- le `do while`

5.2.1 Le `while`

La syntaxe du `while` est la suivante :

`while` (expression) instruction

Le `while` répète l'instruction tant que la valeur de l'expression s'interprète comme vraie (différente de zéro). Il correspond à l'organigramme de la figure 5.1.

L'exemple 5.5 correspond à la recopie de la chaîne de caractères contenue dans `tab` dans `tab2`. Le test de fin de chaîne correspond à `tab[i] == '\0'` puisque le test de passage dans la boucle correspond à `tab[i] != '\0'`. Ce test marche car le compilateur a mis un octet nul (`'\0'`) à la fin de la chaîne `tab` (qui sans cela n'en serait pas une).

Prog. 5.5 – Recopie d'une chaîne avec une boucle `while()`.

```
char tab[]="Coucou c'est moi";
char tab2[50];
int i=0;
while( tab[i] != '\0'){
   tab2[i] = tab[i];
   i++;
}
tab2[i] = '\0';
```

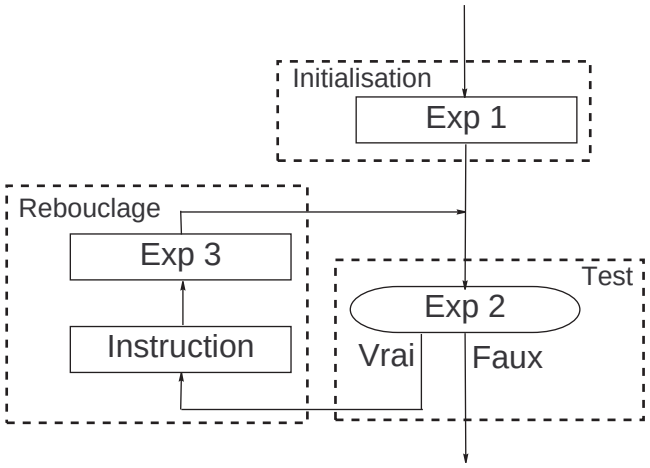

Figure 5.2 – Organigramme du `for`.

5.2.2 Le `for`

La syntaxe du `for` est la suivante :

`for` (exp1 ; exp2 ; exp3) instruction

Le `for` s'utilise avec trois expressions, séparées par des points virgules, qui peuvent être vides :

1. l'expression `expr1` est une instruction d'initialisation. Elle est exécutée avant l'entrée dans la boucle ;
2. l'expression `expr2` est la condition de passage. Elle est testée à chaque passage, y compris le premier. Si elle calcule une valeur vraie l'instruction est exécutée, sinon la boucle se termine ;
3. l'expression `expr3` est une instruction de rebouclage. Elle fait avancer la boucle, elle est exécutée en fin de boucle avant le nouveau test de passage.

Le `for` correspond à l'organigramme 5.2. Elle est équivalente à la construction suivante :

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

Dans le `for` comme dans le `while`, il est à remarquer, que le test est placé en tête et donc que l'instruction n'est pas forcément exécutée. Voici quelques exemples de boucle `for` :

1. `for (i=0 ; i<n ; i++)` ; Cet exemple correspond à une boucle de parcours classique d'un tableau. Dans ce cas, l'indice de début est `0` ; la condition de passage se fait en testant si l'indice courant est strictement inférieur à la taille du tableau, et la progression d'indice se fait par pas de un (`i++`).
2. `for (i=0, j=n ; i<j ; i++,j--)` ; Le deuxième exemple montre comment avoir plusieurs initialisations et plusieurs expressions dans l'instruction de rebouclage, en utilisant l'opérateur de succession « , ».
3. `for ( ; ; ) instruction`. Le dernier exemple est une convention pour écrire une boucle infinie. Ce type de boucle infinie est utilisé lorsque l'instruction n'est pas une instruction simple mais

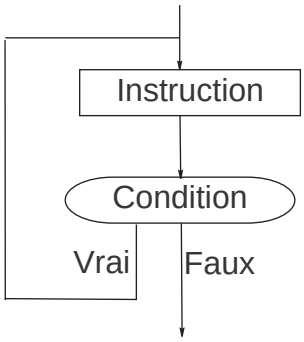

Figure 5.3 – Organigramme du `do while`.

plutôt un bloc d'instructions dans lequel se trouvent des conditions de sortie de la boucle (voir section 5.3 sur les ruptures de séquence).

Le programme 5.6 est un programme qui récupère une ligne contenant au maximum 80 caractères. La fonction `getchar()` est une fonction qui lit un seul caractère au clavier. Nous supposerons que la valeur `EOF` est connue [1].

Prog. 5.6 – Lecture d'une ligne avec `for`.

```
char tab[80];
int rang, c;
for( rang=0; rang<80 && (c=getchar()) != EOF; rang++ )
   tab[rang] = c;
```

5.2.3 Le `do while`

La syntaxe du `do while` est la suivante :

`do` instruction `while` (expression) ;

À l'inverse du `while`, le `do while` place son test en fin d'exécution, d'où au moins une exécution. L'organigramme correspond à celui du `while` mais inversé.

L'exemple donné dans le programme 5.7 teste la divisibilité par trois d'un nombre.

Prog. 5.7 – Divisibilité par trois.

```
int nombre;
do {
    printf("\nEntrez␣un␣nombre␣divisible␣par␣trois␣:␣");
    scanf("%d", &nombre);
    if (nombre%3 == 0)
        printf("\nLe␣nombre␣est␣divisible␣par␣trois␣!");
    else
        printf("\nLe␣nombre␣n'est␣pas␣divisible␣par␣trois␣!␣
            Recommencez␣...");
} while (nombre%3 != 0);
```

1. — `EOF` est le caractère qui marque la fin d'un fichier : « End Of File ». Il est obtenu par la frappe du caractère Contrôle D sur un système de type UNIX, et par la frappe du caractère Contrôle Z sur un système Windows.

5.3 Ruptures de séquence

Dans le cas où une boucle commande l'exécution d'un bloc d'instructions, il peut être intéressant de vouloir sortir de cette boucle alors que la condition de passage est encore valide. Ce type d'opération est appelé une rupture de séquence. Les ruptures de séquence sont utilisées lorsque des conditions multiples peuvent conditionner l'exécution d'un ensemble d'instructions.

Les ruptures de séquence peuvent être classées en quatre genres qui correspondent à leur niveau de travail :

1. `continue`,
2. `break`,
3. `goto`,
4. `return`.

Notons également que l'appel à la fonction`void exit(int status)` termine brutalement l'exécution d'un programme (cf. chapitre 12, librairie `stdlib.h`).

5.3.1 `continue`

Le `continue` est utilisé en relation avec les boucles. Il provoque le passage à l'itération suivante de la boucle en sautant à la fin du bloc. Ce faisant, il provoque la non exécution des instructions qui le suivent à l'intérieur du bloc.

Prenons l'exemple 5.8, qui compte le nombre de caractères non blancs rentrés au clavier, et le nombre total de caractères. Les caractères sont considérés comme blancs s'ils sont égaux soit à l'espace, la tabulation horizontale, le saut de ligne ou le retour à la colonne de numéro zéro. À la fin de l'exécution :

1. `i` contient une valeur qui correspond au nombre total de caractères qui ont été tapés au clavier ;
2. `j` contient une valeur qui correspond au nombre de caractères non blancs ;
3. `i-j` contient une valeur qui correspond au nombre de caractères blancs.

Prog. 5.8 – Utilisation du `continue` dans une boucle `for()`

```
int i, j, c;
for (i=0, j=0; (c=getchar()) != EOF; i++){
    if (c == ' ') continue;
    if (c == '\t') continue;
    if (c == '\r') continue;
    if (c == '\n') continue;
    j++;
}
```

5.3.2 `break`

Nous avons déjà vu une utilisation du `break` dans le `switch`. Plus généralement, il permet de sortir de la boucle d'itération. Il ne peut sortir que d'un niveau d'accolade.

Dans l'exemple 5.9, nous reprenons l'exemple 5.8, de manière à créer une boucle qui compte le nombre de caractères jusqu'à l'obtention de `EOF` (cf. note de bas de page 1, page 39) en utilisant des instructions `break`. Le `break` provoque la sortie de la boucle `for`. Il en serait de même avec un `while` ou un `do while`.

Prog. 5.9 – Utilisation des ruptures de séquence dans une boucle `for()`.

```
int i, j, c;
for( i=j=0; (c=getchar()) != EOF; i++ ) {
    if( c == '\r' ) break;
    if( c == '␣' ) continue;
    j++ ;
}
```

Dans l'exemple 5.10, nous reprenons l'exemple 5.6, de manière à créer une boucle qui remplit le tableau en vérifiant qu'il n'y a pas de débordement de taille. Dans ce cas, nous utilisons le `break` pour sortir de la boucle lorsque la fin de fichier est rencontrée.

Prog. 5.10 – Lecture d'une ligne avec `for` et `break`

```
char tab[80];
int c, rang;
for( rang=0; rang<80; rang++ ) {
    if( (c=getchar()) != EOF )
        tab[rang] = c ;
    else
        break ;
}
```

Les figures 5.4(a), 5.4(b) et 5.4(c) sont des organigrammes qui montrent les effets de `break` et de `continue` sur les boucles `for`, `while` et `do while`.

5.3.3 `goto`

Le `goto` permet d'aller n'importe où à l'intérieur d'une fonction. Son utilisation systématique nuit à la lisibilité des programmes. Il est cependant très utilisé après des détections d'erreur, car il permet de sortir de plusieurs blocs imbriqués. Il est associé à une étiquette appelée `label`. Un `label` est une chaîne de caractères suivie de « : » (cf. programme 5.11).

Prog. 5.11 – Utilisation de l'infâme `goto`.

```
while(exp1) {
    while(exp2) {
        while(exp3) {
            if (probleme) goto erreur;
        }
    }
}

erreur : printf("Erreur␣au␣niveau␣untel");
```

5.3.4 `return`

L'instruction `return` provoque la terminaison de l'exécution de la fonction dans laquelle elle se trouve et le retour à la fonction appellante. Cette instruction peut être mise à tout moment dans le corps d'une fonction ; son exécution provoque la fin de celle-ci. Cette instruction est appelée de manière implicite à la fin d'une fonction. Le `return` est associé à l'expression dont il est suivi.

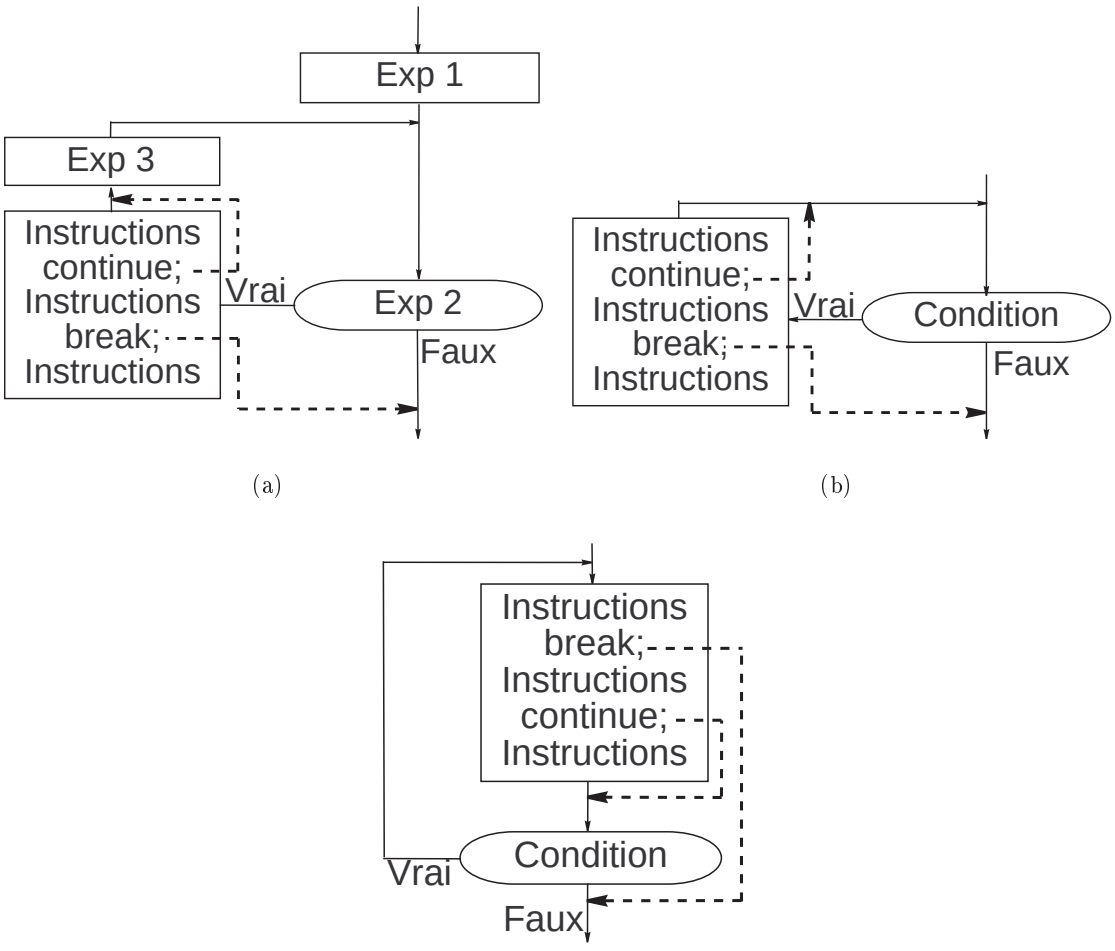

Figure 5.4 – `Break` et `continue` dans (a) un `for`, (b) un `while` et (c) un `do while`.

Cette expression est évaluée et la valeur calculée est retournée à la fonction appellante. C'est la valeur que retourne la fonction contenant le `return`. Les formes possibles du `return` sont :

- `return` ;
- `return` expression ;

Nous allons créer une fonction (voir prog. 5.12) qui retourne :

- `0` si elle lit une majuscule,
- `1` si elle lit une minuscule,
- `2` si elle lit un chiffre,
- `-1` si elle lit `EOF` (cf. note de bas de page 1, page 39),
- `-2` dans tous les autres cas.

Prog. 5.12 – Utilisation de plusieurs `return`.

```
int lec() {
    int c;
    switch( c=getchar() ) {
        case EOF :                  return -1;
        case 'A' : ... case 'Z' : return 0;
        case 'a' : ... case 'z' : return 1;
        case '0' : ... case '9' : return 2;
        default  :                  break;
    }
    return -2;
}
```

Chapitre 6

Fonctions

Sommaire

Les fonctions sont des parties de code source qui permettent de réaliser le même type de traitement plusieurs fois ou sur des objets différents. Les mots *procédure* et *fonction* sont employés dans le reste de ce chapitre de manière indifférente.

Une fonction en langage C peut :

- modifier des données globales. Ces données sont dans une zone de mémoire qui peut être modifiée par le reste du programme. Une fonction peut dans ces conditions réaliser plusieurs fois le même traitement sur un ensemble de variables défini statiquement à la compilation ;
- communiquer avec le reste du programme par une interface. Cette interface est spécifiée à la compilation. L'appel de la fonction correspond à un échange de données à travers cette interface, au traitement de ces données (dans le corps de fonction), et à un retour de résultat via cette interface. Ainsi, une fonction permet de réaliser le même traitement sur des ensembles différents de variables.

6.1 Définition d'une fonction

Lors de leur définition ou de leur utilisation les fonctions sont distinguées des variables par la présence des parenthèses ouvrantes et fermantes. Une définition de fonction, cf. figure 6.1, contient :

- une interface ;
- un corps de fonction.

L'interface complète d'une fonction contient :

- la déclaration du type et du nom de la fonction ;
- une parenthèse ouvrante ;
- la déclaration des types et des noms des paramètres ;

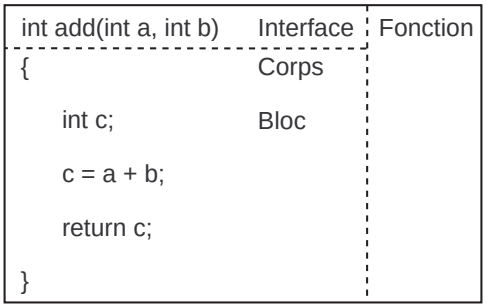

Figure 6.1 – Structure d'une fonction.

- une parenthèse fermante.

Le corps de fonction est un bloc, c'est à dire :

- une accolade ouvrante ;
- des déclarations de variables locales au bloc ;
- des instructions ;
- une accolade fermante.

Le tableau 6.1 donne des exemples de définition de fonctions en C.

Tableau 6.1 – Exemples de définition de fonctions.

Code C	**Explications**

`int plus(int a, int b){`
`  a += b ;`
`  return a ;`
`}`
	fonction `plus` qui retourne un résultat de type entier et dont les deux arguments `a` et `b` sont de type entier.

`void add(int a, int b, int* c)`
`  {`
`  *c = a+b ;`
`  }`
	fonction `add` qui ne retourne pas de résultat (`void`) et qui a trois arguments : `a` et `b` sont deux entiers et `c` est un pointeur d'entier. Cette fonction modifie l'entier pointé par `c`.

`long push(double X, int Y)`
`  {`
`  ...`
`  }`
	fonction `push` qui retourne un résultat de type `long` et qui a deux arguments : `X` de type `double` et `Y` de type `int`.

6.2 Passage des paramètres

En langage C, *les passages de paramètres se font par valeur*, c'est à dire que la fonction appellante fait une copie de la valeur passée en paramètre et passe cette copie à la fonction appelée à l'intérieur d'une variable créée dans l'espace mémoire géré par la pile d'exécution. Cette variable est accessible de manière interne par la fonction à partir de l'argument formel correspondant.

Voici deux exemples d'appels de la fonction `plus()` définie dans le tableau 6.1 :

```
int x = 4, y = 6, z ;
z = plus(1,23) ;
z = plus(x,y) ;
```

Le premier appel à la fonction constitue un **passage de constantes**. Le second appel à la fonction

constitue un **passage de variables**. Ces deux exemples montrent que la fonction appelée peut modifier les paramètres (sauf s'ils sont qualifiés par `const`), mais ces paramètres sont dans son univers local. Les modifications des paramètres formels par une fonction n'ont aucune influence sur la valeur des paramètres utilisés lors de l'appel.

6.3 Utilisation de pointeurs en paramètres

Il est possible, à partir d'une fonction, de modifier des objets de la fonction appellante. Pour cela, il faut que la fonction appellante passe les adresses de ces objets [1].

Les adresses sont considérées comme des pointeurs dans la fonction appelée. Comme pour les autres constantes, il y a promotion de constante à variable lors du passage des paramètres par valeur. Il est en effet possible d'appeler la fonction `plus()` avec des constantes et d'utiliser en interne de cette même fonction des variables. Dans le cas d'une adresse, la promotion en variable rend un pointeur.

Voici deux exemples d'appels de la fonction `add(...)` définie dans le tableau 6.1. Cette fonction utilise le troisième argument pour communiquer le résultat de ses calculs.

```
int x = 5, y = 7, z ;
add(x, y, &z) ;
add(43, 4, &x) ;
```

Lors du premier appel `add(x,y,&z) ;`, la variable `z` est modifiée et elle prend la valeur `12` (`5`+`7`). Lors du second appel `add(43,4,&x) ;` la variable `x` est modifiée et elle prend la valeur `47` (`43`+`4`).

6.4 Conversions de type des paramètres

Lors de l'appel d'une fonction, les paramètres subissent les conversions de type unaire telles que décrites dans la section 2.5. Les plus importantes sont les suivantes :

- les paramètres du type `char` et `short` sont transformés en `int` ;
- les paramètres du type `float` sont transformés en `double`.

Les conversions de type unaire ayant lieu lors de l'appel d'une fonction sont décrites dans le tableau 6.2.

Tableau 6.2 – Conversions de type unaire.

Type du paramètre	Type après conversion
`char`	`int`
`short`	`int`
`int`	`int`
`long`	`long`
`float`	`double`
`double`	`double`

1. — Vous trouvez ici l'explication rationnelle de la formule magique qui consiste à mettre un « et commercial » devant les noms de variables lors de l'appel de la fonction `scanf()`. Il faut en effet passer l'adresse de la variable à modifier à cette fonction.

6.5 Retour de fonction

Toute fonction qui n'est pas de type `void` retourne un résultat. Le type de ce résultat est celui de la fonction. La génération du retour de fonction est provoquée par l'appel de l'instruction « `return` expression ». L'expression qui suit le `return` est évaluée, et la valeur obtenue est retournée. Au niveau de la fonction appellante, le retour de fonction peut être utilisé comme la valeur d'une expression. Si nous prenons le cas de la fonction `plus()` vue précédemment, la valeur du retour de cette fonction peut être utilisée pour affecter une variable (dans cet exemple la variable `z`).

```
int x=12, y=5, z;
z = plus(x, y);
```

Nous pouvons utiliser ce retour de fonction dans toute expression telle que définie dans le chapitre 4. Ce qui nous permet d'écrire par exemple : `z = z * plus(x, y);` ou `z *= plus(x, y);`.

6.6 Récursivité

En C, toute fonction peut appeler toute fonction dont elle connaît le nom (nous reviendrons sur ces problèmes dans le chapitre 8 sur la visibilité). En particulier, elle peut s'appeler elle-même. Il est donc possible d'écrire des *fonctions récursives*.

Prenons l'exemple le plus connu en matière de récursivité : la factorielle. Cette fonction factorielle peut s'écrire de la manière suivante :

Prog. 6.1 – Fonction factorielle.

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

Les limites de cette récursivité sont imposées par la taille de la pile d'exécution, au chargement du programme en mémoire si le système utilise une gestion de pile statique. Si vous testez cette fonction, vous serez sûrement limité, non par la taille de la pile, mais par l'espace de valeur d'un entier.

Pour montrer la concision du langage, voici une factorielle écrite en une seule ligne :

```
int fac(int n) { return n? n*fac(n-1) : 1; }
```

6.7 Paramètres de la fonction principale

La structure des arguments de la fonction `main()` reflète la liaison entre le langage C et le système d'exploitation (shell sous unix). Les paramètres de la fonction `main()` sont passés par le shell dans la majorité des cas. Ces paramètres ont une structure pré-définie. Ils sont décrits de la manière suivante :

```
int main(int argc, char** argv, char** envp)
```

Les noms `argc`, `argv` et `envp` sont des noms mnémoniques. Ils signifient « argument count », « argument values » et « environment pointeur ».

La fonction `main()` dispose donc toujours de trois paramètres passés par l'environnement système. Ces paramètres sont un entier et deux tableaux de pointeurs sur des caractères. La signification de ces arguments est la suivante :

- l'entier `argc` contient le nombre d'arguments qui ont été passés lors de l'appel de l'exécutable (nombre de mots dans la ligne de commande) ;

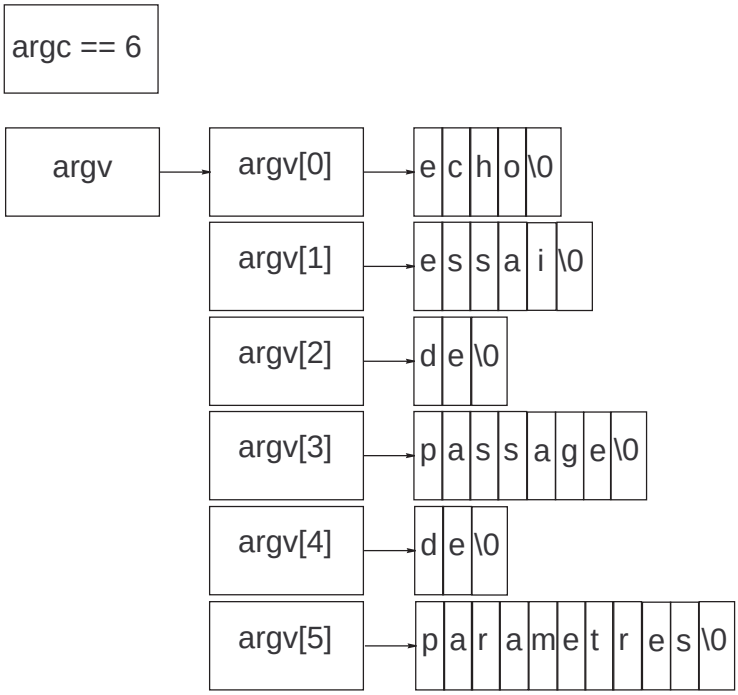

Figure 6.2 – Illustration d'un passage d'arguments à la fonction principale.

- le premier tableau contient les arguments de la ligne de commande au niveau du shell. Ces arguments sont découpés en mots par le shell et chaque mot est référencé par un pointeur dans le tableau. Il y a toujours au moins un argument qui correspond au nom de l'exécutable appelé ;
- le nombre de pointeurs valides dans le premier tableau est donné par `argc` ;
- le deuxième tableau contient les variables d'environnement du shell au moment de l'appel de l'exécutable. Contrairement au premier tableau, la taille de ce deuxième tableau n'est pas donnée par un nombre de mots valides. La fin de ce deuxième tableau est déterminée par un marqueur. Ce marqueur est un pointeur `NULL`, c'est-à-dire, un pointeur qui contient l'adresse `0` ;
- la chaîne pointée par la première entrée du tableau `argv` contient le nom de la commande elle-même.

La figure 6.2 donne la vision interne des paramètres `argc` et `argv` issus de la commande :

`echo essai de passage de parametres`.

6.8 Étapes d'un appel de fonction

Pour résumer voyons les étapes lors d'un appel de fonction. Ces étapes sont au nombre de six et sont réalisées soit par la fonction appellante soit par la fonction appelée :

Mise en pile des paramètres la fonction appellante empile les copies des paramètres. Si le paramètre est une constante (entière, flottante ou adresse), le programme exécutable empile une copie de cette constante et l'on obtient une variable de type compatible. Ceci explique pourquoi vous pouvez passer une constante et manipuler cette constante comme une variable dans la fonction appelée. En C ANSI, vous avez cependant la possibilité de dire que la fonction considère ses arguments comme constants (qualificatif `const`). Si les paramètres sont des variables, la fonction appellante empile une copie constituée à partir de la valeur courante de la variable. Ceci explique pourquoi la valeur d'un argument formel peut être différente de celle de son argument réel.

Saut à la fonction appelée La fonction appellante provoque un saut à l'adresse de début de la fonction appelée en empilant l'adresse de retour.

Prologue dans la fonction appelée La fonction appelée prépare son environnement en particulier, elle positionne son pointeur de contexte dans la pile en sauvegardant l'ancien pointeur de contexte. Ce pointeur de contexte servira pendant toute l'exécution de la fonction à retrouver d'un coté les arguments de la fonction, de l'autre les variables locales à la fonction. Elle fait ensuite grandir la pile de manière à pouvoir ranger une copie des registres qu'elle va utiliser et les variables locales qui ne sont pas en registre. Cette étape est appelée le prologue de la fonction. Ce prologue dépend du type du processeur et des choix de réalisation faits par les concepteurs du compilateur.

La fonction appelée s'exécute jusqu'à rencontrer un `return` en langage C. Ce `return` provoque le passage à l'épilogue dans la fonction appelée. Lorsque le `return` est associé à une valeur cette valeur est conservée dans un registre de calcul (qui sert aussi pour évaluer les expressions).

Epilogue dans la fonction appelée L'épilogue fait le travail inverse du prologue à savoir, il restitue le contexte de la fonction appellante au niveau des registres du processeur (sauf les registres scratch qui contiennent la valeur de retour de la fonction). Pour cela, l'épilogue restaure les registres qu'il avait sauvegardés (correspondants aux registres demandés par les définitions de variables de type `register`), puis il restaure le contexte de pile en reprenant l'ancienne valeur dans la pile. Enfin, il fait diminuer la pile conformément à la réservation qu'il avait fait dans le prologue. Finalement il retourne à la fonction appellante.

Récupération du résultat et effacement des paramètres la fonction appellante dépile les paramètres qu'elle avait empilés au début de l'appel et utilise la(es) valeur(s) de retour de la fonction pour calculer l'expression courante dans laquelle la fonction a été appelée.

Chapitre 7

Préprocesseur

Sommaire

Le préprocesseur traite le fichier source avant le compilateur. Il ne manipule que des chaînes de caractères. Il retire les parties commentaires (entre `/*` et `*/`). Il prend en compte les lignes commençant par un `#` pour créer le code que le compilateur analysera.

Ses possibilités sont de 4 ordres :

- inclusion de fichier : `#include "nom de fichier"` ou `#include <nom de fichier>`
- définition de variables de « preprocessing » :
  - `#define NOM valeur`
  - `#undef NOM`
- définition de macro-fonctions ou macro-expressions : `#define m(x) (342*(x)*(x))`
- sélection du code en fonction des variables du préprocesseur :
  - `#if`
  - `#ifdef` (`if defined ...`)
  - `#ifndef` (`if not defined ...`)
  - `#else`
  - `#endif`

7.1 Commentaires

Les commentaires sont destinés à faciliter la compréhension du source lors de la relecture. Ils ne sont d'aucune utilité au compilateur, et il est naturel qu'ils n'apparaissent pas dans le source qui lui est destiné. Le préprocesseur retire les caractères compris entre `/*` et `*/`. Il ne gère pas les imbrications de commentaires. La mise en commentaire d'une section source peut alors créer des problèmes.

7.2 Inclusion de fichiers

L'inclusion de fichiers par le préprocesseur est déclenchée par la rencontre de la directive : `#include nom_de_fichier`.

Par convention, les fichiers à inclure ont des noms terminés par `.h` pour signifier « header ». Il existe trois façons de nommer un fichier à inclure. Ces façons déterminent l'endroit où le préprocesseur va chercher le fichier.

1. à partir du catalogue courant, si le nom de fichier est entouré par des guillemets :
   - `#include "header.h"`
   - `#include "include\mem.h"`
   - `#include "..\include\uucp.h"`
   - `#include "d:\temp\essai\header.h"`
2. à partir d'un catalogue prédéfini correspondant à l'installation du compilateur, si le nom de fichier est entouré par un inférieur et un supérieur :
   - `#include <stdio.h>`

Les fichiers inclus sont traités par le préprocesseur. Ils peuvent contenir d'autres inclusions de fichiers. Ce processus récursif est parfois limité à quelques niveaux d'inclusion.

7.3 Variables de précompilation

Les variables de précompilation ont deux utilisations :

- La définition de constantes de compilation.
- La définition de variables de précompilation dont la présence permet la sélection du code (cf section 7.5).

7.3.1 Définition de constantes de compilation

Comme le montre le tableau 7.1, l'usage le plus courant des constantes de compilation est associé à la manipulation de tableaux. Il est plus simple et plus sûr d'avoir une constante qui soit utilisée lors de la définition et lors de la manipulation du tableau. La définition d'une constante de précompilation se fait par : `#define nom_de_la_variable valeur`.

Tableau 7.1 – Utilisation d'une constante de compilation.

Sans constante de précompilation	Avec constante de précompilation

`int tab[20] ;`
`for( i=0 ; i<20 ; i++ )`
	`#define LG 20`
	`int tab[LG]`
	`for( i=0 ; i<LG ; i++ )`

Ceci évite de rechercher dans le source les instructions qui font référence à la taille du tableau lorsque cette taille change. Lorsqu'une constante de compilation a été définie, le préprocesseur change toutes les occurrences du nom de la constante par sa valeur, sauf lorsque le nom se trouve dans une chaîne de caractères. Le changement ne se fait que lorsque le nom de la variable est isolé. Le tableau 7.2 est un exemple d'utilisation des variables de précompilation.

Tableau 7.2 – Interprétation des variables par le préprocesseur.

Avant précompilation	Après précompilation
`#define PI 3.14159` `#define LG 20`	

`int i,t[LG] ;`
`for( i=0 ; i<LG ; i++)`
`t[i] = PI ;`
	`int i, t[20] ;`
	`for( i=0 ; i<20 ; i++)`
	`t[i] = 3.14159 ;`

`for( i=0 ; i<LG ; i++ )`	`for( i=0 ; i<20 ; i++ )`
`printf("Valeur de PI %f", PI) ;`	`printf("Valeur de PI %f", 3.14159) ;`

`(PI)`
`LG PI`
`LGPI`
	`(3.14159)`
	`20 3.14159`
	`LGPI`

7.3.2 Définition destinée à la sélection

Le préprocesseur permet de définir des variables de précompilation qui permettent de réaliser les tests de sélection de parties de fichier source. Ces variables n'ont pas besoin d'être associées à une valeur. Elles jouent en quelque sorte le rôle de booléens puisque le préprocesseur teste si elles sont définies (`#ifdef`) ou non (`#ifndef`). Elles servent à déterminer les parties de codes à compiler.

La définition d'une variable de précompilation se fait par : `#define nom_de_la_variable`

7.3.3 Effacement d'une définition

Il est possible d'effacer une définition de variable par : `#undef nom_de_la_variable`. La variable est alors considérée comme non définie. Dans le cas où la variable est associée à une valeur, les occurrences de son nom ne sont plus remplacées par sa valeur.

7.4 Définition de macro-expressions

Le précompilateur permet la définition de macro-expressions encore appelées macro-fonctions. La définition d'une macro-expression se fait par la directive `#define`. Le nom de la macro est suivi d'une parenthèse ouvrante, de la liste des arguments, d'une parenthèse fermante et de la définition du corps de la macro :

`#define add(x1,x2) ((x1) += (x2))`.

Lorsque le préprocesseur rencontre une expression du type `add(a,b)`, il génère `((a) += (b))`.

Il faut faire particulièrement attention à l'ordre d'évaluation pour des macro-expressions complexes, et une technique simple et systématique consiste à encadrer le nom des pseudo-variables dans l'expression de la macro. Comme le montre l'exemple 7.3, le parenthèsage garantit que l'expression résultante est correctement interprétée.

7.5 Sélection de code

La sélection de code permet de générer à partir d'un même fichier source des fichiers exécutables pour des machines et des systèmes différents (Linux, Windows, ...). Le principe de base consiste à

Tableau 7.3 – Évaluation de macros par le préprocesseur.

Avant précompilation	**Après précompilation**	**Bon/Faux**

`#define add(x1,x2) x1 += x2`
`#define m(x) 128*x+342*x*x`
`#define y(x) 128*(x)+342*(x)*(x)`		

`int a,b,c,d ;`
`int e,f,g ;`
`int a = b = c = 1 ;`
`int d = e = f = 2 ;`
	`int a,b,c ;`
	`int e,f,g ;`
	`a = b = c = 1 ;`
	`d = e = f = 2 ;`	

`add(a,b) ;` `add(a,b+1) ;`	`a += b ;` `a += b + 1 ;`	bon bon
`d = m(a) ;` `e = y(a) ;`	`d = 128*a+342*a*a ;` `e = 128*(a)+342*(a)*(a) ;`	bon bon

`d = m(a+b) ;`
`d = y(a+b) ;`
	`d = 128*a+b+342*a+b*a+b ;`
	`d = 128*(a+b)+342*(a+b)`
	`*(a+b) ;`
		faux
		bon

`f = m(add(a,b)) ;`
 	
`f = y(add(a,b)) ;`
	`f = 128*a += b+342*`
	`a += b* a+= b ;`
	`f = 128*(a+=b)+342*`
	`(a += b)*(a+= b) ;`
		faux
		
		bon

passer ou à supprimer des parties de code suivant des conditions fixées à partir des variables de précompilation.

Ceci permet d'avoir un même fichier source destiné à être compilé sur plusieurs machines, ou plusieurs systèmes, en tenant compte des différences entre ces machines.

La sélection est aussi utilisée pour avoir un source avec des instructions qui donnent des informations sur les variables (traces), et pouvoir générer le fichier exécutable avec ou sans ces traces.

La sélection se fait à partir des lignes de directives suivantes :

- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#endif`

Toute condition de sélection de code commence par un `#if[n[def]]`, et se termine par `#endif`, avec éventuellement une partie `#else`.

Lorsque la condition est vraie, le code qui se trouve entre le `#if[n[def]]` et le `#else` est passé au compilateur. Si elle est fausse, le code passé est celui entre le `#else` et le `#endif`. Si il n'y a pas de partie `#else` aucun code n'est passé.

La sélection avec `#if` se fait par test d'une expression valide du langage C. Cette expression ne peut mettre en jeu que des constantes et des variables de précompilation.

Chapitre 8

Compilations séparées

Sommaire

La compilation séparée permet de fragmenter un grand programme en des parties qui peuvent être compilées indépendamment les unes des autres. Dans ce chapitre, nous allons voir comment cette possibilité est exploitable en langage C.

À partir de maintenant, nous appellerons :

déclaration : une association de type avec un nom de variable ou de fonction (dans ce cas la déclaration contient aussi le type des arguments de la fonction),

définition : une déclaration et si c'est une variable, une demande d'allocation d'espace pour cette variable, si c'est une fonction la définition du corps de fonction contenant les instructions associées à cette fonction.

8.1 Programme et fichiers sources

Comme nous l'avons déjà écrit dans le chapitre 1, un programme en langage C est constitué par un ensemble de fichiers destinés à être compilés séparément. La structure d'un programme, écrit en langage C, est résumée dans la figure 8.1.

Comme le montre schématiquement la figure 8.2, chaque fichier source contient les éléments suivants dans un ordre quelconque :

- des déclarations de variables et de fonctions externes,
- des définitions de types synonymes ou de modèles de structures,

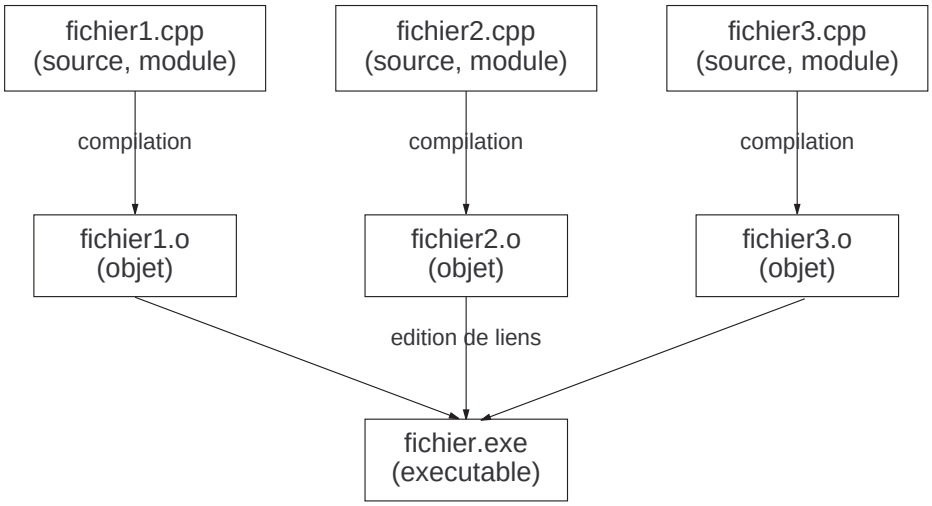

Figure 8.1 – Du source à l'exécutable.

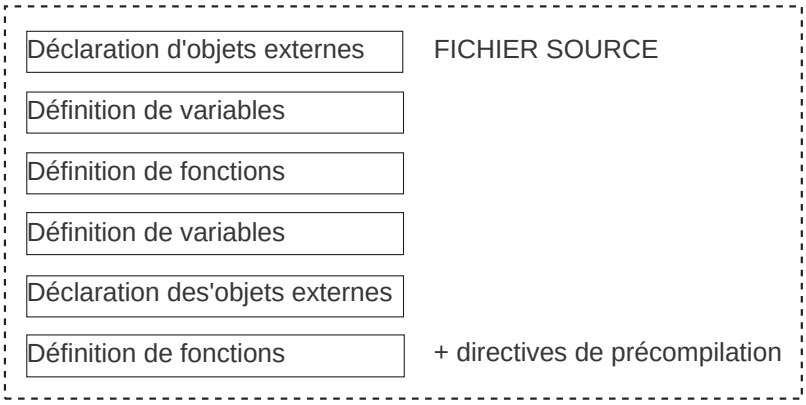

Figure 8.2 – Survol d'un fichier source.

- des définitions de variables[1],
- des définitions de fonctions,
- des directives de précompilation et des commentaires.

Les directives de précompilation et les commentaires sont traités par le préprocesseur. Le compilateur ne voit que les quatre premiers types d'objets.

Les fichiers inclus par le préprocesseur ne doivent contenir que des déclarations externes ou des définitions de types et de modèles de structures.

Règle fondamentale Toute variable ou fonction doit être déclarée avant d'être utilisée.

8.2 Visibilité

La compilation séparée des différentes parties d'un programme implique le respect de certaines règles que nous appellerons *Règles de visibilité*. Ces règles s'appliquent aux noms (de variables et de fonctions).

1. — Ces définitions de variables globales sont en fait des demandes de réservation mémoire destinées à l'éditeur de liens.

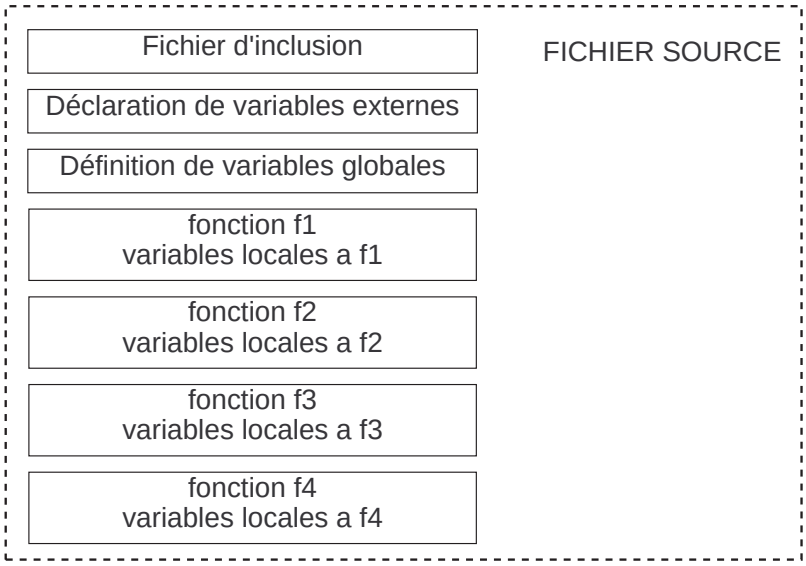

Figure 8.3 – Exemple de visibilité.

8.2.1 Règle de visibilité

Le compilateur ne lit le fichier source qu'une seule fois, du début à la fin. Lorsqu'il rencontre l'utilisation d'une variable ou d'une fonction, il doit connaître son type et sa classe d'adresse.

Par convention, dans le cas où une fonction n'est pas connue, le compilateur considère qu'elle retourne une valeur de type `int` et il essaye de deviner le type des paramètres à partir de l'appel (les appels postérieurs devront se conformer à ce premier appel).

La fonction peut être définie plus loin dans le fichier. Si l'interface est conforme à ce que le compilateur a deviné, le compilateur n'émet pas de message d'erreur et utilise l'adresse de la fonction pour les appels suivants, mais il ne peut pas modifier ce qu'il a déjà généré. La liaison du premier appel avec la fonction est laissée à l'éditeur de liens.

Nous allons nous servir de l'exemple de la figure 8.3 pour continuer à expliquer la visibilité des noms. Pour cet exemple, les règles de visibilité de noms que nous venons d'énoncer nous permettent de dire :

1. la fonction `f4` peut appeler les fonctions `f3`, `f2` et `f1` ;
2. la fonction `f3` peut appeler les fonctions `f2` et `f1` ;
3. la fonction `f2` peut appeler la fonction `f1` ;
4. la fonction `f1` ne peut appeler aucune autre fonction ;
5. les fonctions `f1`, `f2`, `f3` et `f4` peuvent appeler des fonctions inconnues. Dans ce cas le compilateur utilise la règle par défaut et suppose que le résultat de la fonction appelée est un entier.

Une fonction peut utiliser les variables internes qu'elle a définies et les variables globales qui ont été définies avant la fonction. Les noms de variables internes masquent les noms de variables globales. Ainsi, dans la figure 8.4 :

- la définition de la variable locale `a` de type `long` à l'intérieur de la fonction `f1` masque dans cette fonction la variable globale de même nom et de type `int`. L'affectation `a=50` réalisée dans la fonction `f1` modifie la variable locale de type `long` et non la variable globale ;
- par contre, la modification `a=10` réalisée dans la fonction `f2` affecte la variable globale de type entier ;

```
int a;                  // variable globale
void f1(void) {
    long a;             // variable locale a f1
    a = 50;             // modification locale a f1
}

void f2(void) {
    a = 10;             // modification globale
}

void f3(float a) {
    a = 10;             // modification du parametre local a f3
}
```

Figure 8.4 – Masquage de nom.

- de même, l'argument `a` de type `float` de la fonction `f3` masque dans cette fonction la variable globale de même nom et de type `int`. L'affectation `a=10` réalisée dans la fonction `f3` modifie l'argument local de type `float` et non la variable globale.

8.2.2 Extension de la visibilité

Pour le moment, nous n'avons pris en compte que les variables définies dans le module correspondant à une compilation. Il est possible de demander au compilateur de manipuler un objet défini dans un autre module, en lui précisant que l'objet est de classe `extern`. Ceci revient donc à faire une déclaration et non une définition de l'objet.

Si l'objet est une variable, le compilateur accepte son utilisation et fait les contrôles de cohérence sur son type. Il ne réserve pas de place mémoire pour elle. Il passe des informations dans le fichier généré pour demander à l'éditeur de liens de retrouver la variable dans les autres modules. Ces déclarations se placent aussi bien au niveau global qu'à l'intérieur des blocs.

Bien entendu, une telle variable doit être définie dans un autre fichier du programme et tous les fichiers doivent être associés par une édition de liens, sinon celle-ci se terminera avec des références non résolues.

La figure 8.5 donne un exemple de définition et de déclaration de variables entre deux modules d'un même programme. Dans cet exemple :

- les variables `a` et `b` sont définies dans le fichier `prg1.cpp`. Ces variables peuvent être utilisées par les fonctions contenues dans ce fichier. Ces variables sont également accessibles à partir des autres modules du programme qui les auront déclarées puisqu'elles ne sont pas de type `static` ;
- la variable `a` est déclarée en tête du fichier `prg2.cpp` ; elle peut donc être utilisée par toutes les fonctions définies dans `prg2.cpp`. Le compilateur fait la liaison avec cette déclaration lors de l'utilisation de la variable dans les fonctions `f2` et `f3`. Il demande à l'éditeur de liens de trouver la variable `a` ;
- de même, la variable `b` est déclarée localement dans la fonction `f3` et peut être utilisée par cette fonction. Le compilateur fait les vérifications grâce aux informations fournies par la déclaration et demande à l'éditeur de liens de trouver la variable.

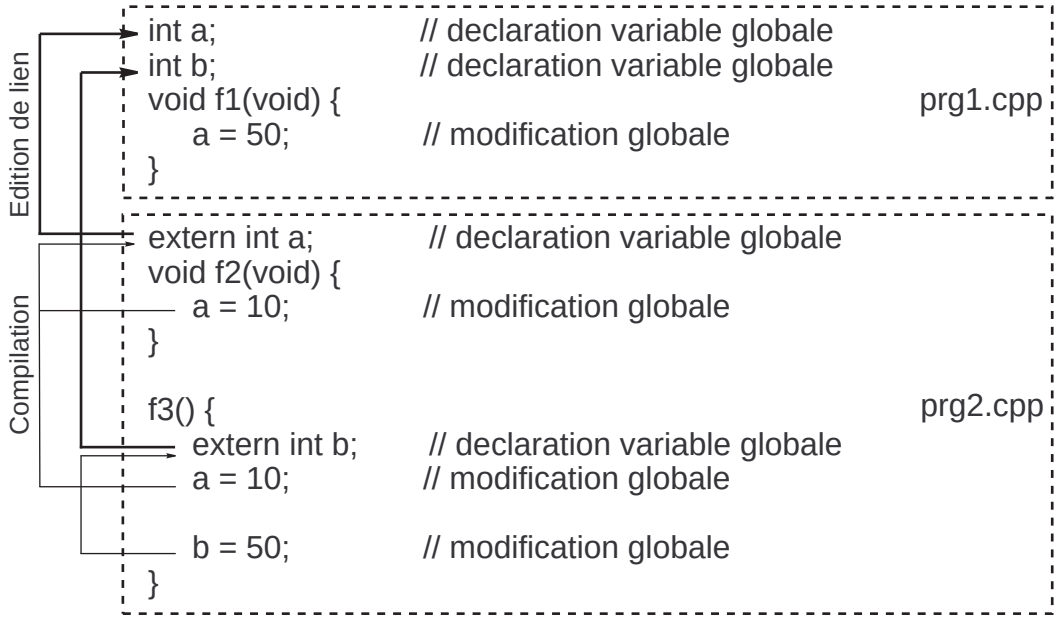

Figure 8.5 – Visibilité des variables entre modules.

8.3 Prototypes des fonctions

Les règles de visibilité s'appliquent aux fonctions de la même manière qu'elles s'appliquent aux variables globales. Il est possible de déclarer une fonction `extern` afin de l'utiliser dans un module différent de celui où elle est définie [2].

La norme ANSI a ajouté au langage C la possibilité de déclarer les prototypes des fonctions, appelés parfois signatures des fonctions, en incluant les types des arguments dans la ligne de déclaration. Un prototype permet de vérifier que les arguments passés à une fonction sont corrects en nombre et en type. Ainsi, les interfaces des fonctions que nous avons vues dans le chapitre 6 sont décrites par les prototypes du programme 8.1.

Prog. 8.1 – Exemples de prototypes de fonctions.

```
1. extern int add(int , int);
2. extern void add2(int, int, int*);
3. extern long push(int, double);
4. extern int fac(int);
5. extern main(int, char**, char**);
6. extern int scanf(char*, ...);
7. extern int printf(char*, ...);
8. extern int getchar(void);
```

Pour représenter des fonctions ayant des arguments variables en nombre et en type (comme `printf()` et `scanf()`), le langage C utilise trois points dans la liste des arguments (cf. lignes `6` et `7` du programme 8.1). Pour décrire le prototype d'une fonction qui n'accepte pas d'argument, on utilise le type `void` dans la liste des arguments, comme le montre la dernière ligne du programme 8.1.

2. — Lorsque le compilateur rencontre un appel à une fonction qui n'est pas déclarée ou définie, il considère que la fonction retourne une valeur entière.

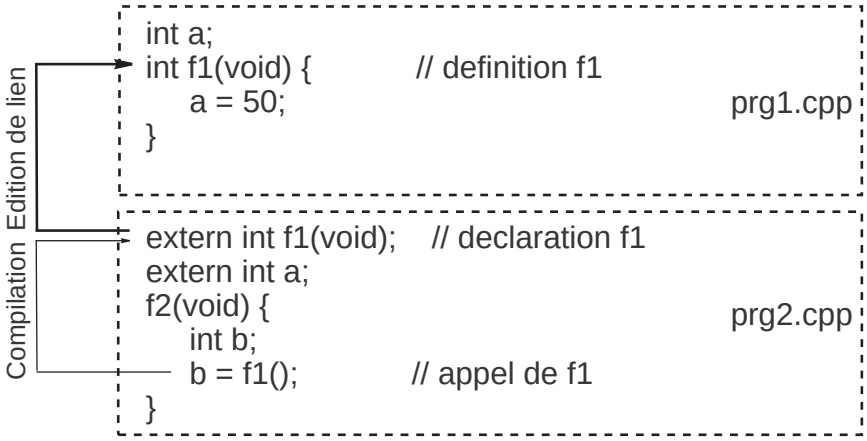

Figure 8.6 – Visibilité des fonctions entre modules.

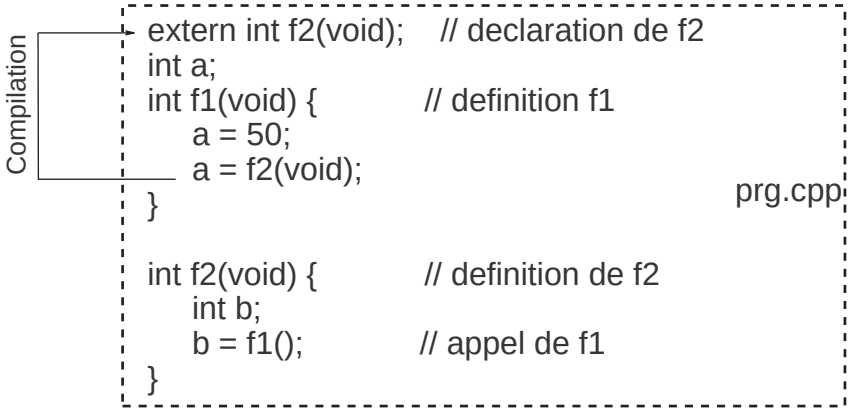

Figure 8.7 – Visibilité des fonctions dans un module.

8.4 Fonctions externes et fonctions définies ultérieurement

Une fois connues, les fonctions des autres modules peuvent être utilisées. Comme le montre la figure 8.6, si la fonction `f1` est définie dans le fichier `prg1.cpp` et que l'on souhaite utiliser dans le fichier `prg2.cpp`, il faut mettre le prototype `extern int f1(void) ;` au début du fichier `prg2.cpp`.

De même, pour utiliser une fonction définie plus loin dans le module, il faut faire une déclaration de cette fonction. Le compilateur fait la mise-à-jour de la référence lorsqu'il rencontre la définition. Comme le montre la figure 8.7, la déclaration `extern int f2(void)` permet l'utilisation d'une fonction sans en connaître le corps.

8.5 Déclarations et définitions multiples

Une déclaration de variable globale qui n'est pas associée avec une initialisation est candidate à être une définition. Ainsi, il est possible de trouver plusieurs déclarations de variables sans le mot `extern` dans plusieurs modules et même dans un seul module. Le compilateur demande à l'éditeur de liens de résoudre les conflits potentiels. Il ne peut bien sur n'y avoir qu'une seule initialisation qui transforme le déclaration candidate en définition.

Pour des questions de lisibilité, les compilateurs acceptent donc les définitions candidates de variables,

ils acceptent ces définitions de manière multiple si le type est identique. Le compilateur ne fait la demande de réservation d'espace qu'une seule fois. Il considère les définitions ultérieures comme des déclarations. Cette facilité permet au programmeur de mettre des déclarations de variables près des endroits où elles sont manipulées.

Nous recommandons :
- d'éviter les déclarations candidates à la définition,
- de toujours associer la définition avec une initialisation,
- d'utiliser le mot `extern` devant les déclarations.

L'exemple suivant montre la déclaration multiple d'une variable. `f1` et `f4` manipulent la même variable `a`.

```
int a;
void f1(void){ a=a+1; }
int a;
void f4(void){ a--;} }
```

Il est cependant plus académique d'écrire une seule définition et des déclarations. `f1` et `f4` manipulent la même variable `a`.

```
int a;
void f1(void){ a=a+1; }
extern int a;
void f4(void){ a— ;} }
```

8.5.1 Fichiers d'inclusion

Les fichiers d'inclusion sont destinés à contenir des déclarations [3] d'objets des types suivants :
- types non pré-définis,
- modèles et noms de structures,
- types et noms de variables,
- prototypes de fonctions.

Les fichiers d'inclusion contiennent les déclarations des variables et fonctions utilisées par plusieurs modules.

La figure 8.8 est un exemple dans lequel :
- le fichier `<defs.h>` contient les déclarations des variables globales et des fonctions qui peuvent être utilisées dans plusieurs fichiers sources participant au programme ;
- les variables `a` et `b`, qui sont définies dans le module `prg1.cpp`, peuvent être utilisées par plusieurs autres modules ;
- les fonctions utilisables sont `f1` qui est définie dans le module `prg1.cpp` et `f3` qui est définie dans le module `prg2.cpp` ;
- le module `prg2.cpp`, demande l'inclusion du fichier `<defs.h>` qui fournit la déclaration de la fonction `f3` lors de la compilation. Ceci permet l'utilisation de cette fonction dans la fonction `f2` sans ambiguïté.

8.5.2 Réduction de la visibilité (fonctions privées)

Dans un environnement qui permet à plusieurs programmeurs de constituer un seul programme, il est intéressant d'avoir une notion de variables privées accessibles seulement à l'intérieur d'un

3. — Ce sont des déclarations et non des définitions. En effet, les fichiers d'inclusion sont associés à plusieurs fichiers source et si ces fichiers contiennent des définitions, les variables risquent d'être définies plusieurs fois.

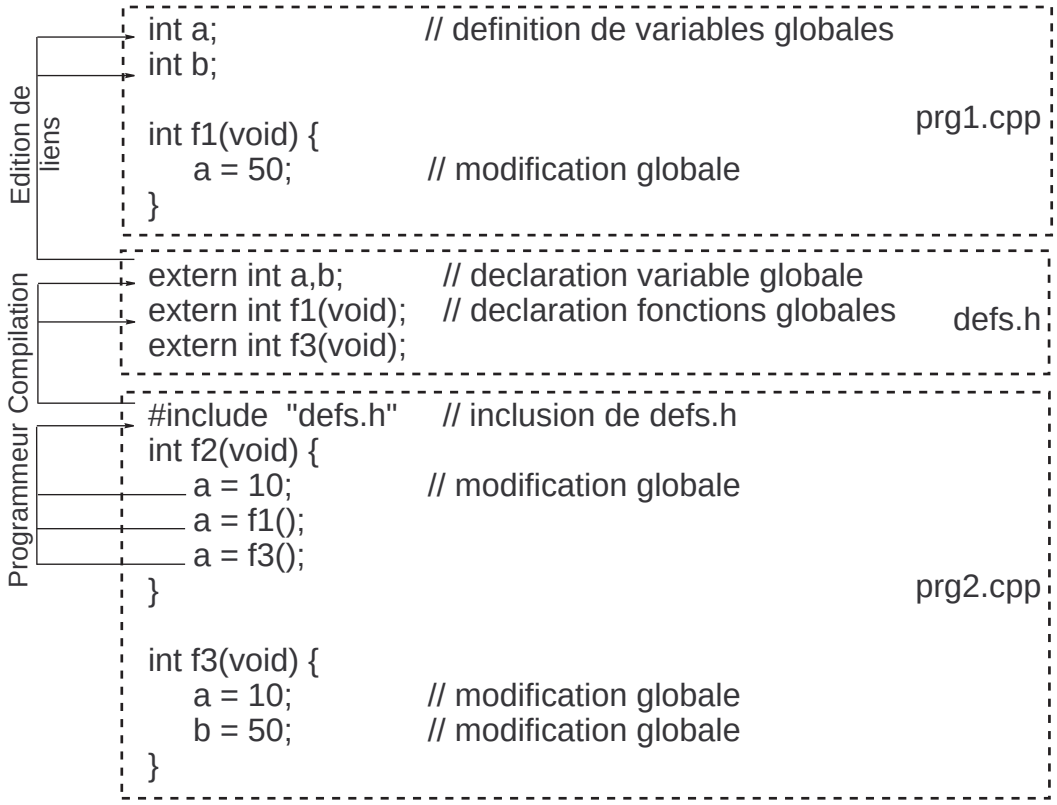

Figure 8.8 – Utilisation d'un fichier d'inclusion.

module. Cette notion de restriction de visibilité peut aussi s'appliquer aux fonctions pour donner un mécanisme d'encapsulation qui n'autorise pas directement l'accès à des variables mais permet cet accès à travers l'appel de fonctions.

En langage C, le prédicat `static` permet de masquer des noms de données ou de fonctions aux autres fichiers du programme. Une variable de type global `static` n'est visible que depuis la partie du programme qui la déclare. Elle n'est accessible qu'aux fonctions définies après elle dans le même fichier. Ceci permet, entre autre, de manipuler des données qui ont le même nom dans plusieurs fichiers avec des définitions différentes dans chacun d'entre-eux.

Comme le montre la figure 8.9, une fonction peut aussi être déclarée `static`, elle sera alors inaccessible aux fonctions des autres fichiers constituant le programme. Une tentative de déclaration `extern` d'une variable ou d'une fonction statique est inefficace.

8.5.3 Variables locales rémanentes

Pendant l'exécution, les variables locales sont normalement créées à l'entrée du bloc dans lequel elles sont définies. Les variables locales à un bloc, appelées également variables automatiques, sont naturellement invisibles de l'extérieur de la fonction.

Le langage C donne la possibilité d'avoir des variables internes à un bloc dont la durée de vie est la même que celle des variables globales. Le prédicat `static` appliqué à une variable locale, modifie le lieu où cette variable est implantée ; elle est, alors, mise avec les variables globales.

Son nom reste invisible à l'extérieur de la fonction. Le prédicat `static` peut être utilisé pour des structures ou des tableaux de grande taille internes à une fonction. Ceci permet de minimiser l'« overhead » causé par la création ou la destruction de l'espace mémoire pour stocker ces variables de

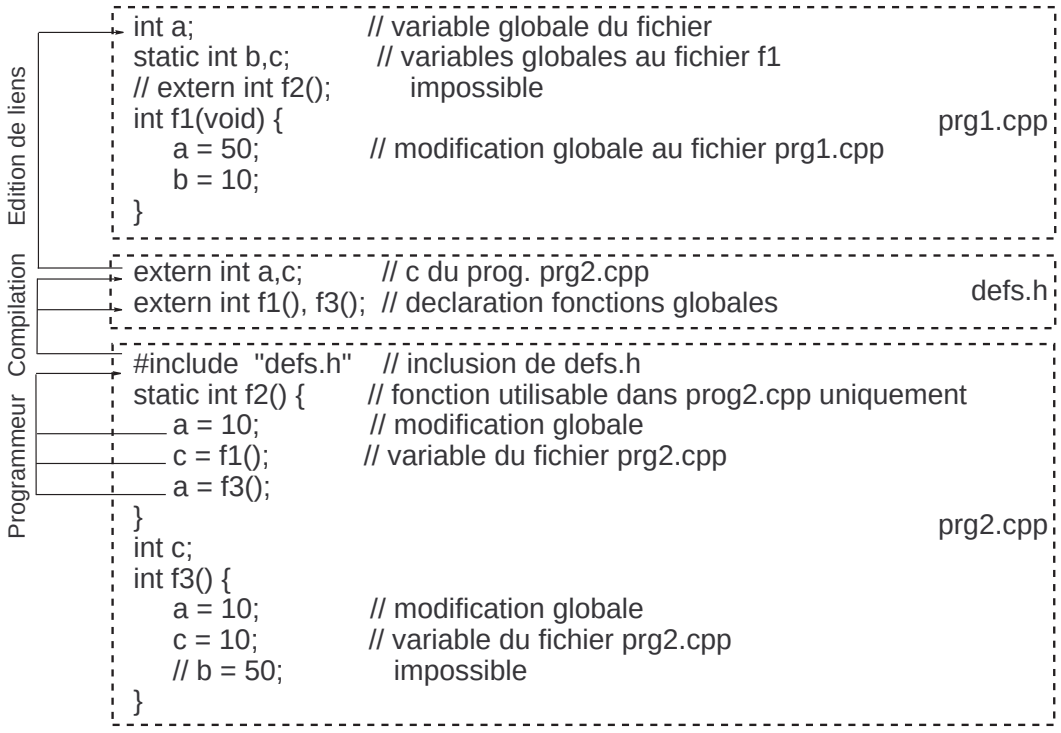

Figure 8.9 – Réduction de visibilité (`static`).

grande taille à l'entrée ou à la sortie de la fonction.

Trois points sont à remarquer :

- si une fonction retourne un pointeur sur une variable statique, celui-ci peut être utilisé par le reste du programme. Le contrôle d'accès à la variable est vérifié à la compilation mais non à l'exécution. Une variable locale statique peut aussi être modifiée par des effets de bord ;
- la durée de vie d'une variable locale statique est la même que celle des variables globales. À chaque appel, une fonction retrouve la valeur d'une variable locale statique qu'elle a modifiée lors des appels précédents ; Nous pouvons donc avoir une variable interne à une fonction qui compte le nombre d'appels à cette fonction ;
- l'initialisation d'une variable statique interne à une fonction est faite à la compilation, et non à l'entrée dans la fonction.

Lorsque le programme de la figure 8.10 s'exécute, la variable entière `a` de type global prend successivement les valeurs : `1,2,3,4,5,6,7,8,9` et la variable `a` locale à `f1` prend successivement les valeurs : `1,11,21,31,41,51,61,71,81,91`.

```
int a;                          // variable globale
void f1(void) {
    static long a=1;          // variable statique locale a f1
    a += 10;                  // modification locale a f1
}

void f2(void) {
    for (a=1; a<10; a++)
        f1();                 // modification globale
}

main() {
    f2();
}
```

Figure 8.10 – Variables locales statiques.

Chapitre 9

Pointeurs et tableaux

Sommaire

Ce chapitre est consacré aux tableaux et à leur manipulation à travers l'utilisation de pointeurs. Cette utilisation des pointeurs pour accéder aux contenus des tableaux est une des difficultés du langage pour les débutants. Elle s'avère cependant l'une des techniques les plus utilisées par les programmeurs expérimentés.

9.1 Tableaux à une dimension

La déclaration d'un tableau à une dimension réserve un espace de mémoire contigu dans lequel les éléments du tableau peuvent être rangés.

Le nom du tableau seul est une constante dont la valeur est l'adresse du début du tableau. Les éléments sont accessibles par : le nom du tableau, un crochet ouvrant, l'indice de l'élément et un crochet fermant.

L'initialisation d'un tableau se fait en mettant une accolade ouvrante, la liste des valeurs servant à initialiser le tableau, et un accolade fermante. La figure 9.1 montre l'espace mémoire correspondant à la définition d'un tableau de dix entiers avec une initialisation selon la ligne :

```
int tab[10] = { 9,8,7,6,5,4,3,2,1,0 };
```

Comme le montrent les exemples du programme 9.1, il est aussi possible de ne pas spécifier la taille du tableau ou de ne pas initialiser tous les éléments du tableau. Dans ce programme, `tb1` est défini comme un tableau de `6` entiers initialisés, et `tb2` est défini comme un tableau de `10` entiers dont les `6` premiers sont initialisés. Depuis la normalisation du langage, l'initialisation des premiers éléments d'un tableau provoque l'initialisation de l'ensemble des éléments du tableau. Les derniers éléments sont initialisés avec la valeur `0`.

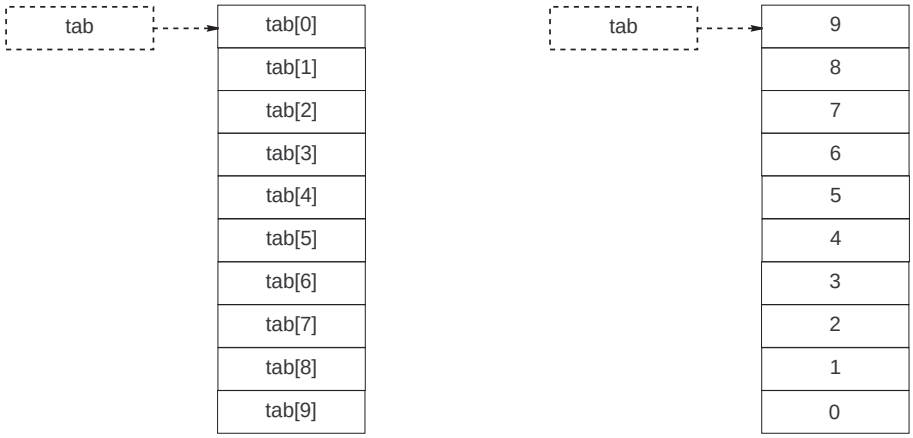

Figure 9.1 – Tableau de dix entiers.

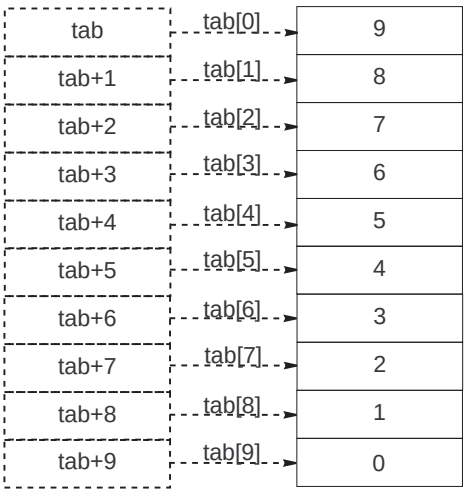

Figure 9.2 – Adresses dans un tableau de dix entiers.

Prog. 9.1 – Définition de tableaux et initialisations.

```
int tb1[] = {12,13,4,15,16,32000};
int tb2[10] = {112,413,49,5,16,3200};
```

Les noms de tableaux étant des constantes, il n'est pas possible de les affecter [1].

9.2 Arithmétique d'adresse et tableaux

Comme nous venons de l'écrire, le nom d'un tableau correspond à l'adresse du premier élément du tableau. Les éléments d'un tableau sont tous du même type. Ils ont donc tous la même taille et ils ont tous une adresse qui correspond au même type d'objet (par exemple une adresse d'entier pour chaque élément du tableau `tb1` du programme 9.1). La figure 9.2 reprend l'exemple de la figure 9.1 en le complétant par les adresses de chaque élément. Ces adresses sont exprimées à partir du début du tableau.

Ceci nous amène à considérer les opérations possibles à partir d'une adresse :

1. — L'instruction `tb1=...` est une hérésie qui mérite l'exclusion de la communauté des utilisateurs du langage C.

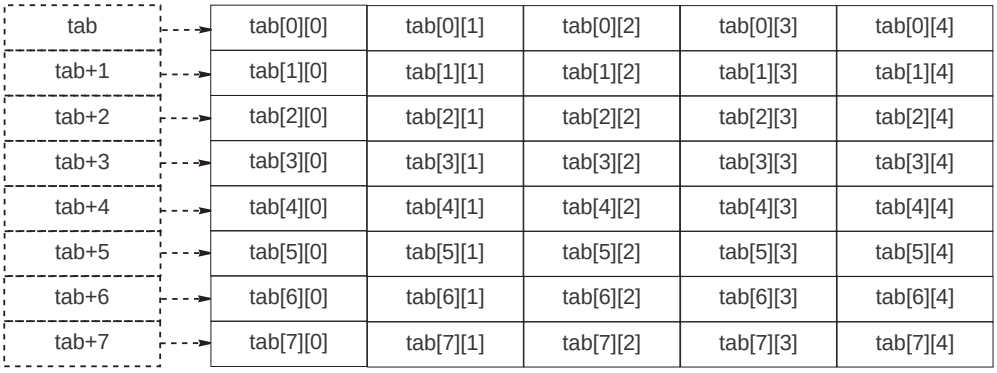

Figure 9.3 – Tableau à deux dimensions.

- il est possible d'additionner ou de soustraire un entier (`n`) à une adresse. Cette opération calcule une nouvelle adresse de la manière suivante :
  - L'opération suppose que l'adresse de départ et l'adresse résultante sont les adresses de deux variables contenues dans le même tableau.
  - L'opération suppose aussi que le tableau est d'une taille suffisamment grande, c'est-à-dire qu'elle suppose que le programmeur doit être conscient des risques de dépassement des bornes du tableau. Dans le cas du tableau `tab` pris en exemple dans la figure 9.2, les adresses doivent être comprises entre `&tab[0]` et `&tab[9]`. Pour une question de test de borne, l'adresse immédiatement supérieure est calculable. Il est donc autorisé d'écrire `&tab[10]`. La zone mémoire correspondante ne doit pas être accédée.
- Selon ces conditions, l'addition d'un entier à une adresse retourne une adresse qui est celle du `n`^{e} objet contenu dans le tableau à partir de l'adresse initiale. Dans ces conditions, `tab + n` est l'adresse du `n`^{e} entier à partir du début du tableau. Dans notre exemple de tableau de dix éléments, `n` doit être compris entre `0` et `10`. L'opérateur d'accès à la variable à partir de l'adresse (`*`) ne peut cependant s'appliquer que pour `n` valant entre `0` et `9`. Ainsi, `*(tab+n)` n'est valide que pour `n` compris entre `0` et `9`.
- L'addition ou la soustraction d'un entier est possible pour toute adresse dans un tableau. Ainsi, `&tab[3]+2` donne la même adresse que `&tab[5]`. De même, `&tab[3]-2` donne la même adresse que `&tab[1]`.
- il est aussi possible de réaliser une soustraction entre les adresses de deux variables appartenant à un même tableau. Cette opération retourne le nombre d'objets entre les deux adresses. Ainsi, `&tab[5]-&tab[3]` doit donner la valeur `2`, et `&tab[3]-&tab[5]` doit donner la valeur `-2`.

9.3 Tableaux multi-dimensions

Les tableaux à deux dimensions sont des tableaux de tableaux. Les indices de droite sont les plus internes. Les tableaux à `n` dimensions sont des tableaux de tableaux à `n-1` dimensions. La figure 9.3 donne les adresses des sous-tableaux et les noms des différents éléments constitués par la définition du tableau à deux dimensions suivant : `int tab[8][5];`.

9.4 Pointeurs et tableaux

Le pointeur est une variable destinée à contenir une adresse mémoire. Il est reconnu syntaxiquement par l'`*` lors de sa déclaration. Comme les adresses, le pointeur est associé à un type d'objet. Ce type est celui des objets qui sont manipulés grâce au pointeur. L'objet peut être une variable ou une

fonction.

La déclaration d'un pointeur n'implique pas la création de l'objet associé et l'affectation de l'adresse de l'objet au pointeur. Il faut donc déclarer un objet du type correspondant et initialiser le pointeur avec l'adresse de cet objet. Par convention, l'adresse `0` est invalide et si l'on cherche à l'accéder, on obtient une erreur d'exécution du type bus-error sur UNIX. Les pointeurs déclarés en variables globales sont initialisés à `0`. Les pointeurs déclarés en `local` ont des valeurs initiales dépendantes du contenu de la pile à cet instant [2].

Voici deux exemples de définition de pointeurs :

- `int* ptint ;` pointeur sur un entier,
- `char* ptchar ;` pointeur sur un caractère.

Le compilateur C vérifie le type des adresses qui sont affectées à un pointeur. Le type du pointeur conditionne les opérations arithmétiques sur ce pointeur. Les opérations possibles sur un pointeur sont les suivantes :

- affectation d'une adresse au pointeur ;
- utilisation du pointeur pour accéder à l'objet dont il contient l'adresse ;
- addition d'un entier (`n`) à un pointeur ; la nouvelle adresse est celle du i^e objet à partir de l'adresse initiale ;
- soustraction de deux pointeurs du même type ; ce qui calcule le nombre d'objets entre les adresses contenues dans les pointeurs (cette valeur est exprimée par un nombre entier).

Le tableau 9.1 donne des exemples de manipulations en relation avec les pointeurs et les tableaux, à partir des variables suivantes : `long x[10], *px, y ;`.

Tableau 9.1 – Addition d'un entier à un pointeur.

Instruction	**Interprétation**
`px = &x[0] ;`	`px` reçoit l'adresse du premier élément du tableau.
`y = *px ;`	`y` reçoit la valeur de la variable pointée par `px`.
`px++ ;`	`px` est incrémenté de la taille de l'objet pointé (4 octets). Il contient `&x[1]`.
`px = px+i ;`	`px` reçoit l'adresse du i^e objet à partir de l'objet courant.

L'addition décrite dans la dernière ligne du tableau 9.1 se traduit par les conversions suivantes : `px = (long*) ( (int) px + i * sizeof (long)) ;` La soustraction de deux pointeurs retourne le nombre d'objets contenus entre les deux pointeurs. Les deux pointeurs doivent être du même type, et ils doivent contenir des adresses d'objets qui sont dans un même tableau.

Le tableau 9.2 est un exemple de soustraction à partir des définitions de variables suivantes : `int i, tab[20], *pt1, *pt2 ;`

Par convention, le nom d'une variable utilisé dans une partie droite d'expression donne le contenu de cette variable dans le cas d'une variable simple. Mais un nom de tableau donne l'adresse du tableau qui est exactement l'adresse du premier élément du tableau.

Nous faisons les constatations suivantes :

- un tableau est une constante d'adressage ;
- un pointeur est une variable d'adressage.

Ceci nous amène à regarder l'utilisation de pointeurs pour manipuler des tableaux, en prenant les variables : `long i, tab[10], *pti ;`

- `tab` est l'adresse du tableau (adresse du premier élément du tableau `&tab[0]` ;

2. — Ce qui, en général, est égal à n'importe quoi.

Tableau 9.2 – Soustraction de deux pointeurs.

Instruction	Interprétation
`pt1 = &tab[0] ;`	`pt1` reçoit l'adresse du premier élément du tableau.
`pt2 = &tab[10] ;`	`pt2` reçoit l'adresse du dixième élément du tableau.
`i = pt2-pt1 ;`	`i` reçoit la différence des deux pointeurs `pt1` et `pt2`. c.a.d le nombre d'objets entre `pt2` et `pt1`. `i` contiendra `10` à la fin de cette instruction.

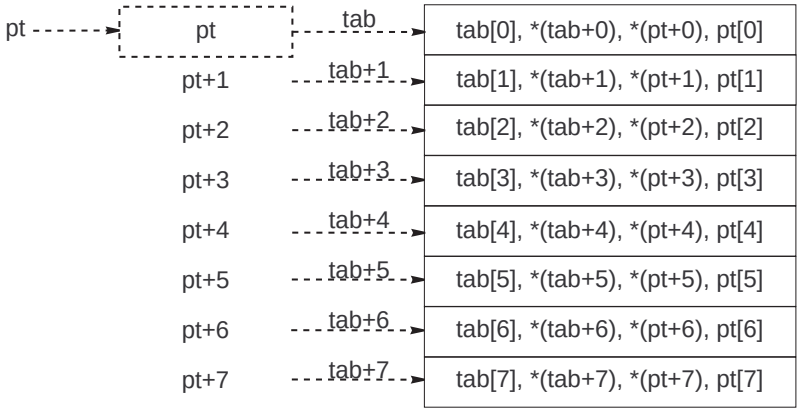

Figure 9.4 – Pointeur et tableau.

- `pti = tab ;` initialise le pointeur `pti` avec l'adresse du début de tableau. Le `&` ne sert à rien dans le cas d'un tableau. `pti = &tab` est inutile et d'ailleurs non reconnu ou ignoré par certains compilateurs ;
- `&tab[1]` est l'adresse du 2^{e} élément du tableau.
- `pti = &tab[1]` est équivalent à :
  - `pti = tab ;` où `pti` pointe sur le 1er élément du tableau.
  - `pti += 1 ;` fait avancer le pointeur d'une case ce qui fait qu'il contient l'adresse du 2^{e} élément du tableau.

Nous pouvons déduire de cette arithmétique de pointeur que : `tab[i]` est équivalent à `*(tab+i)`. La figure 9.4 est un exemple dans lequel sont décrites les différentes façons d'accéder au éléments d'un tableau après la définition suivante : `int tab[8], *pt=tab ;`.

9.5 Tableau de pointeurs

La définition d'un tableau de pointeurs se fait par : `type* nom[taille] ;`

Le premier cas d'utilisation d'un tel tableau est celui où les éléments du tableau de pointeurs contiennent les adresses des éléments du tableau de variables. La figure 9.5 est un exemple d'utilisation à partir des définitions de variables suivantes :

```
int tab[8], *pt[8] = {tab,tab+1,tab+2,tab+3,tab+4,tab+5,tab+6,tab+7} ;
```

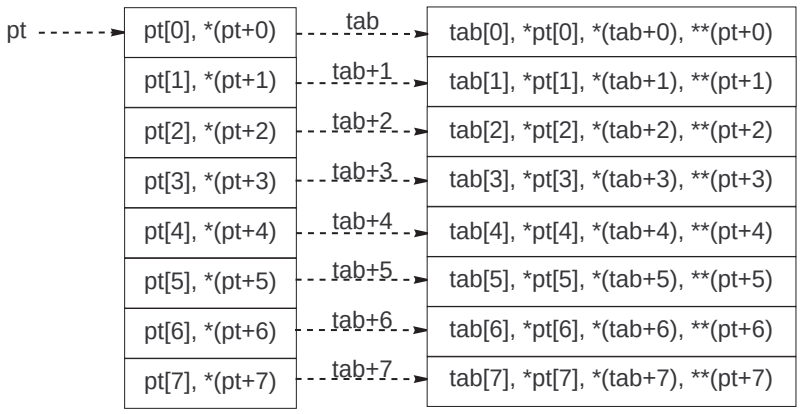

Figure 9.5 – Tableau de pointeurs sur des variables dans un tableau.

Chapitre 10

Structures, unions, énumérations et types synonymes

Sommaire

10.1 Structures

Une *structure* est un objet composé de plusieurs champs qui sert à représenter un objet réel ou un concept.

10.1.1 Définition d'une structure

Le langage C permet de définir des types de structures comme les autres langages évolués. Cela se fait selon la définition donnée par le programme 10.1.

Prog. 10.1 – Syntaxe de la définition d'une structure.

```
struct nom_de_structure {
    type1 nom_champ1;
    type2 nom_champ2;
    etc.
    typeN nom_champ_N;
} variables;
```

Une définition de ce type sert à définir un modèle de structure associé à un nom de modèle, qui est optionnel, et à définir des variables construites à partir de ce modèle (cette définition est optionnelle aussi). Comme le montre l'exemple suivant, il est possible de ne pas déclarer de variable à la définition de la structure, dans ce cas, le modèle de structure doit être associé à un nom de manière à pouvoir utiliser celui-ci pour définir des variables ultérieurement.

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

La définition d'une structure ne réserve pas d'espace mémoire. Il faut définir les variables correspondant à ce modèle de structure : `struct date obdate, *ptdate;`.

10.1.2 Utilisation d'une structure

Les structures peuvent être manipulées champ par champ ou dans leur ensemble.

10.1.2.1 Opérations sur les champs

L'accès aux éléments d'une structure, que nous appelons aussi *champs*, se fait selon la syntaxe : `nom_de_variable.nom_du_champ`. Lorsqu'on dispose d'un pointeur sur une structure, l'écriture diffère un peu en s'écrivant : `nom_de_variable->nom_du_champ`, où la flèche est construite avec le signe moins (`-`) et le signe supérieur (`>`).

En prenant les définitions de données suivantes, le tableau 10.1 donne les différentes possibilités d'affectation pour chaque champ.

```
struct date obdate, *ptdate;
ptdate = &obdate;
```

Une fois ces définitions réalisées, nous pouvons utiliser les variables `obdate` et `ptdate` comme le montre le tableau 10.1.

Tableau 10.1 – Adressage dans une structure.

Objet	Pointeur
`obdate.jour = 1;`	`ptdate->jour = 1;`
`obdate.mois = 1;`	`ptdate->mois = 1;`
`obdate.annee = 85;`	`ptdate->annee = 85;`

Nous remarquerons l'équivalence d'écriture entre : `(*ptdate).jour` et `ptdate->jour`.

Pour le compilateur, l'accès à un champ d'une structure revient à faire un calcul de déplacement par rapport au début de la structure.

10.1.2.2 Opérations sur la variable dans son ensemble

La normalisation du langage C a autorisé une pratique qui consiste à considérer une variable construite à partir d'un type structuré comme une variable simple.

Ainsi, comme le montre les définitions suivantes, il est possible de construire une fonction qui accepte en argument une variable structurée ou qui retourne une variable de type structuré.

```
struct date obdate, obdate2;
struct date newdate();
int checkdate(struct date uneDate);
```

Les opérations suivantes sont possibles sur les variables de type structuré :

1. l'affectation d'un variable structurée par une autre variable du même type, comme

   `obdate=obdate2;`

2. le test d'égalité ou d'inégalité entre deux variables structurées du même type, comme

   `obdate==obdate2;` ou `obdate!=obdate2;`

3. le retour d'une variable structurée par une fonction, comme

   `obdate=newdate();`

4. le passage en argument d'une variable structurée à une fonction, comme

   `result=checkdate(date1);`

10.1.3 Structures et listes chaînées

La syntaxe du langage C autorise les définitions de structure contenant des pointeurs sur ce type de structure. Ceci permet de faire des listes chaînées comme le montre la structure suivante :

```
struct noeud {
   int val;
   struct noeud* suiv;
   struct noeud* pred;
} node, *ptnoeud ;
```

Note sur les pointeurs de structures : nous avons vu que l'incrémentation d'un pointeur fait passer le pointeur sur l'élément suivant. Il en est de même pour un pointeur sur une structure. Un pointeur incrémenté permet l'accès à la structure suivante lorsque les structures sont dans un tableau.

Reprenons la structure noeud précédente et déclarons un tableau de structures et un pointeur sur la structure noeud : `struct noeud tab[100], *ptnoeud = tab;`. La variable `ptnoeud` pointe alors sur `tab[0]` et si nous écrivons `ptnoeud++;`, alors `ptnoeud` pointera sur `tab[1]`.

10.1.4 Champs de bits

Les structures donnent accès aux éléments binaires. Il est possible de découper logiquement un ensemble d'octets en des ensembles de bits. La précision de la longueur de chaque champ est faite par l'ajout de « : longueur » à chaque élément de la structure. Les structures de champs de bits sont déclarées selon le modèle du programme 10.2.

Prog. 10.2 – Définition d'un modèle de structure champs de bits.

```
struct nom_de_structure {
    unsigned nom_champ1 : longueur1;
    unsigned nom_champ2 : longueur2;
    etc.
    unsigned nom_champ_N : longueurN;
} objets ;
```

Il est recommandé de n'utiliser que des éléments de type `unsigned`. La norme X3J11 n'impose pas que d'autres types soient supportés. Un champ sans nom, avec simplement la taille, est un champ de remplissage pour cadrer sur des frontières de mot machine.

Voici quelques exemples de définitions de modèles de structures correspondant à des champs de bits :

```
struct mot {
   unsigned sign    : 1;
   unsigned val     : 15;
};

struct flottant {
   unsigned exposant    : 7;
   unsigned signe       : 1;
   unsigned mantisse    : 24;
};

struct mixte {
   unsigned exposant    : 7;
   unsigned signe       : 1;
   unsigned mantisse    : 24;
   unsigned comp        : 7;
                        : 9;
};
```

10.2 Unions

Les *unions* permettent l'utilisation d'un même espace mémoire par des données de types différents à des moments différents.

10.2.1 Définition d'union

La définition d'une union respecte une syntaxe proche de celle d'une structure (cf. programme 10.3).

Prog. 10.3 – Syntaxe de la définition d'une union.

```
union nom_de_union {
    type1 nom_champ1;
    type2 nom_champ2;
    etc.
    typeN nom_champ_N;
} variables;
```

L'exemple suivant définit deux variables `z1` et `z2` construites sur le modèle d'une zone qui peut contenir soit un entier, soit un entier `long`, soit un nombre avec point décimal, soit un nombre avec point décimal long.

```
union zone {
    int entier;
    long entlong;
    float flottant;
    double flotlong;
} z1,z2;
```

Lorsque l'on définit une variable correspondant à un type `union`, le compilateur réserve l'espace mémoire nécessaire pour stocker le plus grand des champs appartenant à l'union. Dans notre exemple, le compilateur réserve l'espace mémoire nécessaire pour stocker un double pour chacune des variables `z1` et `z2`.

10.2.2 Accès aux champs

La syntaxe d'accès aux champs d'une union est identique à celle pour accéder aux champs d'une structure (voir section 10.1.2.1).

Une union ne contient cependant qu'une donnée à la fois et l'accès à un champ de l'union pour obtenir une valeur, doit être fait dans le même type qui a été utilisé pour stocker la valeur. Si cette uniformité n'est pas respectée dans l'accès, l'opération devient dépendante de la machine.

Les unions ne sont pas destinées à faire des conversions. Elles ont été inventées pour utiliser un même espace mémoire avec des types de données différents dans des étapes différentes d'un même programme. Elles sont très utilisées dans les compilateurs.

Les différents « champs » d'une union sont à la même adresse physique. Ainsi, les égalités suivantes sont vraies :

```
&z1.entier == (int*)&z1.entlong
&z1.entier == (int*)&z1.flottant
&z1.entier == (int*)&z1.flotlong
```

Dans l'exemple 10.4, la zone mémoire correspondant à la variable `lmot` peut être vue sans signe ou avec signe. Ainsi si la valeur `-1` est affectée à `lmot.ent`, alors cette valeur peut être considérée comme plus grande que zéro (puisque égale à `USHRT_MAX` ou `65535U`) ou plus petite suivant qu'elle est vue à travers `lmot.ent` ou `lmot.ent1`. Ainsi le programme 10.4 donne le résultat suivant : Valeur de `lmot.ent : -1`, `Valeur de lmot.ent1 : 65535`.

Prog. 10.4 – Utilisation d'une union.

```
#include <stdio.h>
int main(int argc, char** argv) {
    union etiq {
        short int ent;
        unsigned short int ent1;
    } lmot= {-1,};
    printf("Valeur␣de␣lmot.ent␣:␣\%hd\n", lmot.ent);
    printf("Valeur␣de␣lmot.ent1␣:␣\%hd\n", lmot.ent1);
}
```

10.3 Énumérations

Les *énumérations* servent à offrir des possibilités de gestion de constantes énumérées dans le langage C. Ces énumérations sont un apport de la norme ANSI et permettent d'exprimer des valeurs constantes de type entier en associant ces valeurs à des noms.

Les énumérations offrent une alternative à l'utilisation du préprocesseur (cf. chapitre 7).

10.3.1 Définition

La définition d'une énumération respecte la syntaxe donnée dans le tableau 10.5.

Prog. 10.5 – Syntaxe de la définition d'une énumération.

```
enum nom_de_énumération {
    enumerateur1,
    enumerateur2,
    etc.
    enumerateurN
} variables;
```

Les valeurs associées aux différentes constantes symboliques sont, par défaut, définies de la manière suivante : la première constante est associée à la valeur `0` ; les constantes suivantes suivent une progression de `1`.

```
enum couleurs {
    rouge,
    vert,
    bleu
} rvb;
```

Ce premier exemple d'énumération définit les constantes symboliques en utilisant les possibilités standard des énumérations :

1. `rouge` qui correspond à la valeur `0` ;
2. `vert` qui correspond à la valeur `1` ;
3. `bleu` qui correspond à la valeur `2`.

Il est possible de fixer une valeur à chaque énumérateur en faisant suivre l'énumérateur du signe égal et de la valeur entière exprimée par une constante ; la progression reprend avec `+1` pour l'énumérateur suivant :

```
enum autrescouleurs {
    violet=4,
    orange,
    jaune=8
};
```

Dans ce deuxième exemple d'énumération, les constantes symboliques suivantes sont définies :

1. `violet` qui est associée à la valeur `4` ;
2. `orange` qui correspond à la valeur `5` (`violet + 1`) ;
3. `jaune` qui est associée à la valeur `8`.

10.3.2 Utilisation

Les énumérateurs peuvent être utilisés dans des expressions du langage à la même place que des constantes du type entier. Ainsi, ils peuvent se situer dans des calculs, pour affecter des variables et pour réaliser des tests.

10.3.3 Limites des énumérations

Les énumérations sont des types synonymes (cf. section 10.4) du type entier. Elles permettent d'utiliser des constantes symboliques. Les variables définies à partir de type énuméré sont cependant traitées comme des entiers. Ceci présente l'avantage de pouvoir réaliser des expressions combinant des variables de type énuméré avec des entiers. Le compilateur n'impose pas qu'une variable de type énuméré soit affectée avec une des valeurs correspondant aux constantes symboliques associées avec ce type. Ainsi nous pouvons écrire en reprenant les définitions précédentes :

```
rvb = 12;
arcenciel = violet + jaune;
if (rvb == arcenciel)
   tfic = repertoire;
else
   tfic = normal;
rvb = sock_unix;
```

Dans cet exemple nous mélangeons les constantes symboliques appartenant à des types d'énumération différents sans avoir de message d'avertissement du compilateur.

10.4 Types synonymes

Il est possible grâce au déclarateur `typedef` de définir un type nouveau qui est un type *synonyme*. L'introduction du mot réservé `typedef` comme premier mot d'une ligne de définitions provoque le fait que les noms qui seraient des noms de variables sur la même ligne sans le mot `typedef` deviennent des noms de types synonymes. Chaque nom de type synonyme correspond au type qu'aurait eu la variable sur la même ligne sans `typedef`.

Ainsi la définition suivante :

```
typedef int entier;
```

définit un type synonyme appelé `entier` ayant les mêmes caractéristiques que le type pré-défini `int`.

Une fois cette définition réalisée, nous pouvons utiliser ce nouveau type pour définir des variables et nous pouvons mélanger les variables de ce type avec des variables entières pour réaliser des expressions.

```
entier e1=23, e2=5, te[50]={1,2,3,4,5,6,7};
int i;
i = e1 + e2;
te[20] = i - 60;
```

Dans le cas de la déclaration de tableaux, la taille du nouveau type se trouve après le nouveau nom de type.

```
typedef int tab[10];
tab tt;
```

Dans cet exemple, `tab` devient un type synonyme correspondant au type tableau de dix entiers. La variable `tt` est donc un tableau de dix entiers. `typedef` est très semblable à une directive du préprocesseur (cf. chapitre 7) mais il s'adresse au compilateur.

Attention : `typedef` ne réserve pas d'espace mémoire. Le nom est un type ; il est donc inaccessible comme variable.

L'utilisation de `typedef` permet au compilateur de faire des vérifications sur les types d'objets. En particulier sur les pointeurs.

Il est aussi possible de définir un type équivalent à une structure :

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

et d'utiliser directement ce type. Ceci évite les répétitions fastidieuses du mot réservé `struct` :

`date obdate, *ptdate;`.

Ainsi `obdate` est un objet correspondant au type `date` qui est synonyme d'une structure anonyme contenant trois variables entières appelées `jour`, `mois` et `annee`. La variable `ptdate` est un pointeur qui peut contenir une adresse d'une variable du type `date` mais qui pour le moment n'est pas initialisée.

Chapitre 11

Entrées-sorties de la bibliothèque standard

Sommaire

Comme nous l'avons vu, le langage C est de bas niveau. La bibliothèque du langage C fournit des traitements assez courants de plus haut niveau et permet, entre autres, la réalisation des entrées-sorties. La bibliothèque standard contient :

1. les fonctions permettant la gestion des entrées-sorties (E/S) ;
2. des fonctions de manipulation de chaînes de caractères ;
3. des fonctions d'allocation dynamique de mémoire ;
4. des fonctions à caractère général qui permettent l'accès au système.

Ces fonctions et les structures associées sont décrites dans le fichier `<stdio.h>`. Tout programme désirant manipuler les E/S devra contenir la directive : `#include <stdio.h>`. Ce fichier contient :

➪ la définition de macro-expressions ;

- le prototype des fonctions ;
- la définition de constantes : `EOF` (cf. note de bas de page 1, page 39), etc ;
- la définition du tableau des fichiers ouverts.

Le langage C donne une vision « UNIXienne » des E/S. La même interface d'accès aux fichiers peut être utilisée sur les périphériques ; ainsi, même l'écran et le clavier qui permettent la communication avec l'utilisateur, seront appelés fichier ou flux.

En langage C, un fichier est une suite d'octets (un caractère = 1 octet). Le fichier n'a pas de structure propre qui lui est attachée. Il est cependant possible au niveau programme de considérer un fichier comme une suite d'enregistrements correspondant à une structure. Le principe de la manipulation d'un fichier est le suivant :

1. ouverture du fichier ;
2. lecture, écriture, et déplacement dans le fichier ;
3. fermeture du fichier.

En ce qui concerne les flux de communication avec l'utilisateur, l'ouverture et la fermeture sont effectuées automatiquement.

11.1 Entrées-sorties standard

Ce sont des fonctions pré-définies qui permettent de réaliser les opérations suivantes :

- lecture et écriture caractère par caractère ;
- lecture et écriture ligne par ligne ;
- lecture et écriture formatées.

11.1.1 Échanges caractère par caractère

Dans ces échanges, l'unité de base est le caractère que le système considère comme un octet parfois réduit à une version de l'ASCII sur sept bits.

Les deux fonctions que nous présentons ici sont souvent des macro-expressions du préprocesseur (voir chapitre 7), nous les considérerons comme des fonctions pour simplifier la présentation.

`int getchar(void) ;`

synopsis : cette fonction permet de lire un caractère sur l'entrée standard s'il y en a un. Ce caractère est considéré comme étant du type `unsigned char`.

argument : rien.

retour : la fonction retourne un entier pour permettre la reconnaissance de la valeur de fin de fichier (`EOF`). L'entier contient soit la valeur du caractère lu soit `EOF`.

conditions d'erreur : en fin de fichier la fonction retourne la valeur `EOF`.

`int putchar(int) ;`

synopsis : Cette fonction permet d'écrire un caractère sur la sortie standard.

argument : Elle est définie comme recevant un entier [1] pour être conforme à `getchar()`. Ceci permet d'écrire `putchar(getchar())`.

retour : Elle retourne la valeur du caractère écrit toujours considéré comme un entier.

conditions d'erreur : en cas d'erreur la fonction retourne `EOF`.

1. — La promotion unaire fait que de toute façon la valeur d'un caractère est convertie en une valeur d'entier lors du passage d'argument à une fonction.

LE programme 11.1 réalise la lecture de son fichier standard d'entrée caractère par caractère et reproduit les caractères sur son fichier standard de sortie.

Prog. 11.1 – Lecture et écriture caractère par caractère sur les E/S standard.

```
#include <stdio.h>
void main(){
    int c;
    while((c=getchar()) != EOF)
        putchar(c);
}
```

11.1.2 Échanges ligne par ligne

Dans ces échanges, l'unité de base est la ligne de caractères. La ligne est considérée comme une suite de caractères `char` terminée par un caractère de fin de ligne ou par la détection de la fin du fichier. Le caractère de fin de ligne a une valeur entière égale à `10` et est représenté par l'abréviation `'\n'`.

`char* gets(char*);`

synopsis : lire une ligne sur l'entrée standard ; les caractères de la ligne sont rangés (un caractère par octet) dans la mémoire à partir de l'adresse donnée en argument à la fonction. Le retour chariot est lu mais n'est pas rangé en mémoire. Il est remplacé par le caractère nul `'\0'` de manière à ce que la ligne, une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.

argument : l'adresse d'une zone mémoire dans laquelle la fonction doit ranger les caractères lus.

retour : Si des caractères ont été lus, la fonction retourne l'adresse de son argument pour permettre une imbrication dans les appels de fonction.

conditions d'erreur : Si la fin de fichier est atteinte, lors de l'appel (aucun caractère n'est lu), la fonction retourne un pointeur de caractère de valeur `0` ou `NULL` (défini dans `<stdio.h>`).

remarque : Cette fonction ne peut pas vérifier que la taille de la ligne lue est inférieure à la taille de la zone mémoire dans laquelle il lui est demandé de placer les caractères. Il faut lui préférer la fonction `fgets()` pour tout logiciel de qualité.

`int puts(char*);`

synopsis : cette fonction permet d'écrire une chaîne de caractères, suivie d'un retour chariot sur la sortie standard.

argument : l'adresse d'une chaîne de caractères.

retour : une valeur entière non négative en cas de succès.

conditions d'erreur : Elle retourne la valeur `EOF` en cas de problème.

Le programme 11.2 est une nouvelle version du programme précédent, cette fois-ci ligne par ligne. Le choix d'un tableau de `256` caractères permet d'espérer qu'aucune ligne lue ne provoquera un débordement par la fonction `gets()`. Ceci reste un espoir et peut être démenti si le programme est confronté à une ligne de plus grande taille, dans ce cas le comportement du programme est non défini.

Prog. 11.2 – Lecture ligne par ligne sur les fichiers standard.

```
#include <stdio.h>
void main(){
    char BigBuf[256];
    while(gets(BigBuf)!=NULL)
        puts(BigBuf);
}
```

11.1.3 Échanges avec formats

Les fonctions permettant de faire des E/S formatées sur les fichiers standard d'entrée et de sortie sont les suivantes :

`int scanf(const char*, ...);`

synopsis : lecture formatée sur l'entrée standard.

arguments : comme l'indique la spécification "...", cette fonction accepte une liste d'arguments variable à partir du second argument.

1. le premier argument est une chaîne de caractères qui doit contenir la description des variables à saisir.
2. les autres arguments sont les adresses des variables (conformément à la description donnée dans le premier argument) qui sont affectées par la lecture.

retour : nombre de variables saisies.

conditions d'erreur : la valeur `EOF` est retournée en cas d'appel sur un fichier standard d'entrée fermé.

`int printf(const char*, ...);`

synopsis : écriture formatée sur la sortie standard a

arguments : chaîne de caractères contenant des commentaires et des descriptions d'arguments à écrire, suivie des valeurs des variables.

retour : nombre de caractères écrits.

conditions d'erreur : la valeur `EOF` est retournée en cas d'appel sur un fichier standard de sortie fermé.

Des exemples de formats sont donnés dans le tableau 3.1, page 23. Les possibilités en matière de format sont données de manière exhaustive dans la partie sur les E/S formatées (voir section 11.5).

11.2 Ouverture d'un fichier

Tout fichier doit être ouvert pour pouvoir accéder à son contenu en lecture, écriture ou modification.

L'ouverture d'un fichier est l'association d'un objet extérieur (le fichier) au programme en cours d'exécution. Une fonction d'ouverture spécifie le nom du fichier à l'intérieur de l'arborescence du système de fichiers et des attributs d'ouverture.

L'ouverture d'un fichier est réalisée par la fonction `fopen` selon la description suivante :

`FILE* fopen(const char*, const char* );`

synopsis : ouverture d'un fichier référencé par le premier argument (nom du fichier dans le système de fichiers sous forme d'une chaîne de caractères) selon le mode d'ouverture décrit par le second argument (chaîne de caractères).

arguments :

1. la première chaîne de caractères contient le nom du fichier de manière à référencer le fichier dans l'arborescence. Ce nom est dépendant du système d'exploitation dans lequel le programme s'exécute.
2. Le deuxième argument est lui aussi une chaîne de caractères. Il spécifie le type d'ouverture.

retour : pointeur sur un objet de type `FILE` (type défini dans `<stdio.h>`) qui sera utilisé par les opérations de manipulation du fichier ouvert (lecture, écriture ou déplacement).

conditions d'erreur : le pointeur `NULL` est retourné si le fichier n'a pas pu être ouvert (problèmes d'existence du fichier ou de droits d'accès).

Le type d'ouverture est spécifié à partir d'un mode de base et de compléments. A priori, le fichier est considéré comme une fichier de type texte (c'est-à-dire qu'il ne contient que des caractères ASCII).

Le type d'ouverture de base peut être :

- `"r"` le fichier est ouvert en lecture. Si le fichier n'existe pas, la fonction ne le crée pas.
- `"w"` le fichier est ouvert en écriture. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe la fonction le vide.
- `"a"` le fichier est ouvert en ajout. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier.

Le type d'ouverture peut être agrémenté de deux caractères qui sont :

- `"b"` le fichier est considéré en mode binaire. Il peut donc contenir des données qui sont transférées sans interprétation par les fonctions de la bibliothèque.
- `"+"` le fichier est ouvert dans le mode complémentaire du mode de base. Par exemple s'il est ouvert dans le mode `"r+"` cela signifie qu'il est ouvert en mode lecture et plus, soit lecture et écriture.

La combinaison des modes de base et des compléments donne les possibilités suivantes :

- `"r+"` le fichier est ouvert en lecture plus écriture. Si le fichier n'existe pas, la fonction ne le crée pas. Le fichier peut être lu, modifié et agrandi.
- `"w+"` le fichier est ouvert en écriture plus lecture. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. Le fichier peut être manipulé en écriture et relecture.
- `"a+"` le fichier est ouvert en ajout plus lecture. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier. Le fichier peut être lu.
- `"rb"` le fichier est ouvert en lecture et en mode binaire. Si le fichier n'existe pas, la fonction ne le crée pas.
- `"wb"` le fichier est ouvert en écriture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide.
- `"ab"` le fichier est ouvert en ajout et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier.
- `"r+b"` ou `"rb+"`le fichier est ouvert en lecture plus écriture et en mode binaire. Si le fichier n'existe pas, la fonction ne le crée pas. Le fichier peut être lu, modifié et agrandi.
- `"w+b"` ou `"wb+"` le fichier est ouvert en écriture plus lecture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. Le fichier peut être écrit puis lu et écrit.
- `"a+b"` ou `"ab+"` le fichier est ouvert en ajout plus lecture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier.

La fonction `fopen()` retourne un descripteur de fichier qui servira pour toutes les autres opérations. Le descripteur de fichier doit être déclaré comme dans l'exemple : si le système ne peut pas ouvrir le

fichier, il retourne un descripteur de fichier égal au pointeur `NULL`. Si un fichier qui n'existe pas est ouvert en mode écriture ou ajout, il est créé par le système.

Prog. 11.3 – Ouverture d'un fichier.

```
#include <stdio.h>
void main(){
    FILE* MyFich;
    MyFich = fopen("d:\temp\passwd","r");
}
```

11.3 Fermeture d'un fichier

Avant d'étudier les fonctions permettant d'accéder aux données d'un fichier ouvert, considérons la fonction qui permet de terminer la manipulation d'un fichier ouvert. Cette fonction réalise la fermeture du fichier ouvert. Elle a le prototype suivant :

```
int fclose(FILE*);
```

synopsis : c'est la fonction inverse de `fopen`. Elle détruit le lien entre le descripteur de fichier ouvert et le fichier physique. Si le fichier ouvert est en mode écriture, la fermeture provoque l'écriture physique des données du tampon (voir 11.7).

arguments : un descripteur de fichier ouvert valide.

retour : 0 dans le cas normal.

conditions d'erreur : la fonction retourne `EOF` en cas d'erreur.

Tout programme manipulant un fichier doit donc être encadré par les deux appels de fonctions `fopen()` et `fclose()` comme le montre le programme 11.4.

Prog. 11.4 – Ouverture et fermeture d'un fichier.

```
#include <stdio.h>
void main(){
    FILE* MyFich;
    MyFich = fopen("d:\temp\passwd","r");
    ...
    fclose(MyFich);
}
```

11.4 Accès au contenu du fichier

Une fois le fichier ouvert, le langage C permet plusieurs types d'accès à un fichier :

- par caractère,
- par ligne,
- par enregistrement,
- par données formatées.

Dans tous les cas, les fonctions d'accès au fichier (sauf les opérations de déplacement) ont un comportement séquentiel. L'appel de ces fonctions provoque le déplacement du pointeur courant relatif au fichier ouvert. Si le fichier est ouvert en mode lecture, les opérations de lecture donnent une impression de consommation des données contenues dans le fichier jusqu'à la rencontre de la fin du fichier.

11.4.1 Accès caractère par caractère

Les fonctions suivantes permettent l'accès caractère par caractère :

`int fgetc(FILE*) ;`

synopsis : lit un caractère (`unsigned char`) dans le fichier associé ;

argument : descripteur de fichier ouvert ;

retour : la valeur du caractère lu promue dans un entier ;

conditions d'erreur : à la rencontre de la fin de fichier, la fonction retourne `EOF` et positionne les indicateurs associés (voir section 11.8).

`int getc(FILE*) ;`

synopsis : cette fonction est identique à `fgetc()` mais peut être réalisée par une macro définie dans `<stdio.h>`.

`int ungetc(int, FILE*) ;`

synopsis : cette fonction permet de remettre un caractère dans le buffer de lecture associé à un flux d'entrée.

`int fputc(int , FILE*) ;`

synopsis : écrit dans le fichier associé décrit par le second argument un caractère spécifié dans le premier argument. Ce caractère est converti en un `unsigned char` ;

argument : le premier argument contient le caractère à écrire et le second contient le descripteur de fichier ouvert ;

retour : la valeur du caractère écrit promue dans un entier sauf en cas d'erreur ;

conditions d'erreur : en cas d'erreur d'écriture, la fonction retourne `EOF` et positionne les indicateurs associés (voir section 11.8).

`int putc(int , FILE*) ;`

synopsis : cette fonction est identique à `fputc()` mais elle est réalisée par une macro définie dans `<stdio.h>`.

Les fonctions (macros) `getc()` et `putc()` sont en fait la base de `getchar()` et `putchar()` que nous avons utilisées jusqu'ici.

- `putchar(c)` est défini comme `putc(c,stdout)` ;
- `getchar()` est défini comme `getc(stdin)`.

Comme `getchar()`, la fonction `getc()` retourne un entier pour pouvoir retourner la valeur `EOF` en fin de fichier. Dans le cas général, l'entier retourné correspond à un octet lu dans le fichier et rangé dans un entier en considérant l'octet comme étant du type caractère non signé (`unsigned char`).

Voici un exemple d'écriture à l'écran du contenu d'un fichier dont le nom est donné en argument. Le fichier est ouvert en mode lecture et il est considéré comme étant en mode texte.

Prog. 11.5 – Lecture caractère par caractère d'un fichier après ouverture.

```
#include <stdio.h>
void main(int argc, char** argv){
    if(argc != 2) return 1;

    FILE* MyFich=fopen(argv[1], "r");
    if(MyFic == NULL) {
        printf("Ouverture impossible %s \n", argv[1]);
        return 2;
    }

    int TheCar;
    while((TheCar=fgetc(MyFic))!=EOF)
        fputc(TheCar,stdout);

    fclose(MyFic);
}
```

11.4.2 Accès par ligne

Comme pour les E/S standards, il est possible de réaliser des opérations de lecture et d'écriture ligne par ligne à l'intérieur de fichiers ouverts. Cet accès ligne par ligne se fait grâce aux fonctions :

`char* fgets(char*, int , FILE*);`

synopsis : lit une ligne de caractères dans le fichier associé, les caractères de la ligne sont rangés dans la mémoire à partir de l'adresse donnée en argument à la fonction. Le retour chariot est lu et rangé en mémoire. Il est suivi par le caractère nul `'\0'` de manière à ce que la ligne, une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.

arguments :

1. adresse de la zone de stockage des caractères en mémoire,
2. nombre maximum de caractères (taille de la zone de stockage),
3. et descripteur de fichier ouvert.

retour : adresse reçue en entrée sauf en cas d'erreur ;

conditions d'erreur : à la rencontre de la fin de fichier, la fonction retourne `NULL` et positionne les indicateurs associés (voir section 11.8).

`int fputs(const char*, FILE*);`

synopsis : cette fonction permet d'écrire une chaîne de caractères référencée par le premier argument dans le fichier décrit par le second argument.

argument :

1. le premier argument contient l'adresse de la zone mémoire qui contient les caractères à écrire. Cette zone doit être un chaîne de caractères (terminée par le caractère nul). Elle doit contenir un retour chariot pour obtenir un passage à la ligne suivante.
2. le second argument contient le descripteur de fichier ouvert dans lequel les caractères seront écrits.

retour : une valeur positive si l'écriture s'est correctement déroulée.

conditions d'erreur : en cas d'erreur d'écriture, la fonction retourne `EOF` et positionne les indicateurs associés (voir section 11.8).

Pour la lecture ligne à ligne, il est nécessaire de donner l'adresse d'un tableau pouvant contenir la ligne. De plus, il faut donner la taille de ce tableau pour que la fonction `fgets` ne déborde pas du tableau. La fonction `fgets()` lit au plus `n-1` caractères et elle ajoute un caractère nul (`'\0'`) après le dernier caractère qu'elle a mis dans le tableau. La rencontre du caractère de fin de ligne ou de la fin de fichier provoque la fin de lecture. Le caractère de fin de ligne n'est pas mis dans le tableau avant le caractère nul.

La fonction `fputs()` écrit une ligne dans le fichier. Le tableau de caractères doit être terminé par un caractère nul (`'\0'`). Il faut mettre explicitement la fin de ligne dans ce tableau pour qu'elle soit présente dans le fichier.

Voici le programme précédent écrit à l'aide de ces fonctions :

Prog. 11.6 – Lecture ligne à ligne d'un fichier après ouverture.

```
#include <stdio.h>
void main(int argc, char** argv) {

    if(argc!=2) return 1;

    FILE* TheFic=fopen(argv[1], "r");
    if(TheFic == NULL) {
        printf("Ouverture␣impossible␣%s␣\n", argv[1]);
        return 2;
    }

    char BigBuf[256];
    while( fgets(BigBuf,sizeof(BigBuf), TheFic) != NULL
        )
        fputs(BigBuf,stdout);

    fclose(TheFic);
}
```

Les différences entre `gets()` et `fgets()`, d'une part, et `puts()` et `fputs()`, d'autre part, peuvent s'expliquer par le fait que les fonctions `puts` et `gets` agissent sur les E/S standards qui sont le plus souvent des terminaux (écran + clavier).

Les différences sont les suivantes :

- la fonction `gets()` :
  1. ne nécessite pas que lui soit fournie la taille du tableau de lecture. Il faut espérer que les lignes saisies seront plus courtes que la taille du tableau. Comme ces lignes viennent a priori d'un terminal, elles font souvent moins de `80` caractères.
  2. ne met pas le caractère de fin de ligne dans le tableau. Ce caractère est remplacé par le caractère de fin de chaîne.
- la fonction `puts()` : ne nécessite pas la présence du caractère de fin de ligne dans le tableau. Ce caractère est ajouté automatiquement lors de l'écriture.

11.4.3 Accès par enregistrement

L'accès par enregistrement permet de lire et d'écrire des objets structurés dans un fichier. Ces objets structurés sont le plus souvent représentés en mémoire par des structures. Pour ce type d'accès, le fichier doit être ouvert en mode binaire (voir exemple 11.7 ligne 11). Les données échangées ne sont pas traitées comme des caractères.

L'accès par enregistrement se fait grâce aux fonctions :

- `size_t fread (void* Zone, size_t Taille, size_t Nbr, FILE* fp);`
- `size_t fwrite(void* Zone, size_t Taille, size_t Nbr, FILE* fp);`

Ces fonctions ont une interface homogène. Elles acceptent une liste identique d'arguments, et retournent le même type de résultat. Les arguments sont les suivants :

1. le premier argument, que nous avons appelé `Zone`, est l'adresse de l'espace mémoire à partir duquel l'échange avec le fichier est fait. L'espace mémoire correspondant reçoit les enregistrements lus, ou fournit les données à écrire dans les enregistrements. Il faut, bien entendu, que l'espace mémoire correspondant à l'adresse soit de taille suffisante pour supporter le transfert des données, c'est-à-dire d'une taille au moins égale à (`Taille* Nbr`).
2. le deuxième argument, que nous avons appelé `Taille`, est la taille d'un enregistrement en nombre d'octets.
3. le troisième argument, que nous avons appelé `Nbr`, est le nombre d'enregistrements que l'on désire échanger.
4. le dernier argument, que nous avons appelé `fp`, est un descripteur de fichier ouvert dans un mode de transfert binaire.

Ces deux fonctions retournent le nombre d'enregistrements échangés. Elles ont été conçues de manière à permettre l'échange de plusieurs structures, ce qui explique la présence des deux arguments qui fournissent la taille totale des données à transférer.

Voici un exemple d'utilisation de la fonction `fread()`. Cet exemple est une lecture du contenu d'un fichier appelé `FicParcAuto` avec stockage du contenu de ce fichier dans un tableau en mémoire `ParcAuto`. Les cases du tableau sont des structures contenant un entier, une chaîne de vingt caractères et trois chaînes de dix caractères.

Prog. 11.7 – Lecture d'enregistrements dans un fichier.

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

struct auto {
   int age;
   char couleur[20], numero[10], type[10], marque[10];
} ParcAuto[20];

void main(int argc, char** argv) {

   FILE* TheFic=fopen(argv[1], "rb+");
   if (TheFic == NULL) {
        printf("Ouverture impossible %s \n", argv[1]);
        return 1;
   }

   size_t fait;
   for(int i = 0; i<20; i++){
        fait=fread(&ParcAuto[i], sizeof(struct auto), 1, TheFic);
        if(fait != 1) {
             printf("Erreur lecture %s \n", argv[1]);
             return 2;
        }
   }
   fclose(TheFic);
}
```

Il est possible de demander la lecture des vingt enregistrements en une seule opération, en remplaçant les lignes 18 à 24 par les lignes suivantes :

```
fait=fread(ParcAuto, sizeof(struct auto), 20, TheFic);
```

ou bien encore par :

```
fait=fread(ParcAuto, sizeof(ParcAuto), 1, TheFic);
```

11.5 Entrées-sorties formatées

Nous avons déjà vu comment utiliser `printf` et `scanf`. Nous allons approfondir nos connaissances sur ces fonctions et sur les autres fonctions qui font des conversions de format. Les fonctions que nous allons voir utilisent des formats de conversion entre le modèle des données machines et le modèle nécessaire à la vision humaine (chaîne de caractères).

Les lectures formatées nécessitent :
- le format de description des lectures à faire ;
- une adresse[2] pour chaque variable simple ou pour un tableau de caractères[3].

Les écritures formatées nécessitent :
- le format de description des écritures à faire ;
- les valeurs des variables simples à écrire. Comme dans le cas de la lecture, l'écriture d'un ensemble de caractères est une opération particulière qui peut se faire à partir de l'adresse du premier caractère.

11.5.1 Formats : cas de la lecture

Les formats de conversion servent à décrire les types externes et internes des données à lire. Les formats peuvent contenir :

1. des caractères "blancs" (espace, tabulation). La présence d'un caractère blanc provoque la lecture et la mise à la poubelle des caractères "blancs" (espace, tabulation, retour chariot) qui seront lus ;
2. des caractères ordinaires (ni blanc, ni `%`). Ces caractères devront être rencontrés à la lecture ;
3. des spécifications de conversion, commençant par le caractère `%`.

Une conversion consiste en :

1. un caractère de pourcentage (`%`) ;
2. un caractère (optionnel) d'effacement (`*`) ; dans ce cas la donnée lue est mise à la poubelle ;
3. un champ (optionnel) définissant la taille de la donnée à lire exprimée par une valeur entière en base dix ;
4. un caractère (optionnel) de précision de taille qui peut être : `l`, `h` ou `L`. Ces caractères agissent sur les modes de spécification de la manière suivante :
   (a) si le format initial est du type `d`, `i` ou `n`, les caractères `l` et `h` précisent respectivement que la donnée est du type entier long (`long int`) ou entier court (`short int`) plutôt qu'entier ordinaire (`int`).
   (b) si le format initial est du type `o`, `x` ou `u`, les caractères `l` et `h` précisent respectivement que la donnée est du type entier long non signé (`unsigned long int`) ou entier court non signé (`unsigned short int`) plutôt qu'entier non signé (`unsigned int`).

2. — Ce qui explique pourquoi il faut un & devant les noms de données à lire quand ces noms ne sont pas des tableaux.

3. — Des formats particuliers permettent la lecture de plusieurs caractères à l'intérieur d'un tableau de caractères.

(c) si le format initial est du type `e` ou `f` ou `g`, les caractères `l` et `L` précisent respectivement que la donnée est du type nombre avec point décimal de grande précision (`double`) ou nombre avec point décimal de très grande précision (`long double`) plutôt que du type nombre avec point décimal (`float`).

(d) dans tous les autres cas, le comportement est indéfini.

5. un code de conversion.

Les codes de conversion sont décrits dans le tableau 11.1.

Tableau 11.1 – Code de conversion pour `scanf`.

Code	Conversion réalisée
`%`	lit un `%`
`d`	entier signé exprimé en base décimale
`i`	entier signé exprimé en base décimale
`o`	entier non signé exprimé en base octale
`u`	entier non signé exprimé en base décimale
`x`	entier non signé exprimé en hexadécimal
`e`,`f` ou `g`	nombre avec partie décimale en notation point décimal, ou notation exponentielle
`c`	caractère
`s`	mots ou chaîne de caractères sans blanc
`[spécif.]`	chaîne de caractères parmi un alphabet
`p`	adresse, pour faire l'opération inverse de l'écriture avec `%p`
`n`	permet d'obtenir le nombre d'octets lus dans cet appel

La spécification entre les crochets définit un alphabet [4] de caractères. La donnée lue doit être conforme à cette spécification. La lecture avec un format de spécification retourne une chaîne de caractères.

11.5.2 Formats : cas de l'écriture

Les formats de conversion servent à décrire les types externes et internes des données à écrire. Les formats peuvent contenir :

1. des caractères qui sont recopiés dans la chaîne engendrée par l'écriture ;
2. et des spécifications de conversion.

Une conversion consiste en :

1. un caractère de pourcentage (`%`) ;
2. des drapeaux (flags) qui modifient la signification de la conversion ;
3. la taille minimum du champ dans lequel est insérée l'écriture de la donnée ;
   (a) la taille est donnée en nombre de caractères,
   (b) pour les chiffres, si la taille commence par `-` la donnée est cadrée à gauche.
   (c) pour une chaîne de caractère, si la taille est précédée de `0`, la chaîne est cadrée à droite et est précédée de zéros ;

4. — Cet alphabet est défini soit par la liste des caractères significatifs, soit par le premier caractère suivi d'un tiret et le dernier caractère dans l'ordre croissant de la table ASCII (cf. annexe A). La négation de l'alphabet peut être obtenue en mettant un ∧ après le crochet ouvrant. Pour que le crochet fermant soit dans l'alphabet, il faut qu'il suive immédiatement le crochet ouvrant.

4. un point suivi de la précision. La précision définit le nombre de chiffres significatifs pour une donnée de type entier, ou le nombre de chiffres après la virgule (. !) pour une donnée de type flottant. Elle indique le nombre de caractères pour une chaîne ;
5. un `h`, un `l` ou un `L` signifiant court ou long et permettant de préciser :
   (a) dans le cas d'une conversion d'un entier (format `d, i, o, u, x, X`) que l'entier à écrire est un entier court (`h`) ou long (`l`) ;
   (b) dans le cas d'une conversion d'un nombre avec partie décimale (format `e, f, g, E, G`) que le nombre à écrire est un nombre avec point décimal de très grande précision (`long double`).
6. un code de conversion.

Les champs taille et précision peuvent contenir une `*`. Dans ce cas la taille doit être passée dans un argument à `[sf]printf`. Par exemple les lignes suivantes d'appel à `printf()` sont équivalentes :

```
printf("Valeur␣de␣l'entier␣Indice␣:␣%*d\n", 6, Indice);
printf("Valeur␣de␣l'entier␣Indice␣:␣%6d\n", Indice);
```

Les codes de conversion sont décrits dans le tableau 11.2.

Tableau 11.2 – Code de conversion pour `printf`.

Code	Conversion réalisée
`%` `d`	lit un `%` entier signé exprimé en base décimale
`i`	entier signé exprimé en base décimale
`o`	entier non signé exprimé en base octale
`u`	entier non signé exprimé en base décimale
`x,X`	entier non signé exprimé en hexadécimal
`e,E`	nombre avec partie décimale en notation exponentielle
`f`	nombre avec partie décimale en notation point décimal
`g`	nombre avec partie décimale, `printf` choisit le format `f` ou `e`
`c`	caractère
`s`	chaîne de caractères
`p`	la valeur passée est une adresse
`n`	permet d'obtenir le nombre d'octets écrits

La différence entre `x` et `X` vient de la forme d'écriture des valeurs décimales entre `10` et `15`. Dans le premier cas, elles sont écrites en minuscule (`a-f`), dans le second cas, elles sont écrites en majuscule (`A-F`). De même, le caractère `E` de la notation exponentielle est mis en minuscule par les formats `e` et `g`. Il est mis en majuscule par les formats `E` et `G`. Le nombre maximum de caractères qui peuvent être construits dans un appel aux fonctions de type `fprintf()` ne doit pas dépasser `509`. Les drapeaux sont décrits dans le tableau 11.3.

11.5.3 Conversion sur les entrées-sorties standards

```
int printf(const char* format, ...);
int scanf (const char* format, ...) ;
```

11.5.4 Conversion en mémoire

Les deux fonctions de conversion en mémoire s'appellent `sprintf` et `sscanf`. Les appels sont les suivants :

Tableau 11.3 – Modificateurs de format pour `printf`.

Drapeau	Modification apportée
`-`	la donnée convertie est cadrée à gauche.
`+`	si la donnée est positive le signe `+` est mis.
blanc	si le résultat de la conversion ne commence pas par un signe, un blanc. est ajouté.
`#`	pour format `o` augmente la précision de manière à forcer un `0` devant la donnée. pour format `x` et `X` force `0x` devant la donnée. pour format `e`, `E`, `f`, `g`, et `G` force le point décimal. pour format `g` et `G` les zéros après le point décimal sont conservés.

```
int sprintf(char* string, const char* format, ...);
```

synopsis : conversion de données en mémoire par transformation en chaîne de caractères.

arguments :

1. zone dans laquelle les caractères sont stockés ;
2. format d'écriture des données ;
3. valeurs des données.

retour : nombre de caractères stockés.

```
int sscanf(char* string, const char* format, ...);
```

synopsis : lecture formatée à partir d'une zone mémoire.

arguments :

1. zone dans laquelle les caractères sont acquis ;
2. format de lecture des données ;
3. adresse des variables à affecter à partir des données.

retour : nombre de variables saisies.

conditions d'erreur : la valeur `EOF` est retournée en cas d'erreur empêchant toute lecture.

`sprintf` convertit les arguments `arg1, ..., argn` suivant le format de contrôle et met le résultat dans string (chaîne de caractères).

Inversement, `sscanf` extrait d'une chaîne de caractères des valeurs qui sont stockées dans des variables suivant le format de contrôle.

11.5.5 Conversion dans les fichiers

Deux fonctions `fprintf` et `fscanf` permettent de réaliser le même travail que `printf` et `scanf` sur des fichiers ouverts en mode texte :

```
int fprintf(FILE*, const char*, ...);
```

synopsis : écriture formatée sur un fichier ouvert en mode texte.

arguments :

1. descripteur de fichier ouvert valide dans lequel les caractères sont rangés ;
2. format d'écriture des données ;

3. valeurs des données.

retour : nombre de caractères écrits.

conditions d'erreur : une valeur négative est retournée en cas d'erreur d'écriture.

```
int fscanf(FILE*, const char*, ...);
```

synopsis : lecture formatée dans un fichier ouvert en mode texte.

arguments :

1. descripteur de fichier ouvert dans lequel les caractères sont lus ;
2. format de lecture des données ;
3. adresse des variables à affecter à partir des données.

retour : nombre de variables saisies.

conditions d'erreur : la valeur `EOF` est retournée en cas d'appel sur un fichier standard d'entrée fermé.

Le programme 11.8 ouvre le fichier `d:\temp\passwd` (fichier contenant les identifications des utilisateurs) et qui extrait les différents champs dont les numéros d'utilisateur et de groupe, en mettant ces numéros dans des variables de type entier. Rappelons qu'une ligne de ce fichier contient les champs suivants séparés par le caractère deux points ' :'.

- le nom de l'utilisateur,
- le mot de passe (crypté),
- le numéro d'utilisateur,
- le numéro de groupe dans lequel l'utilisateur est placé à la connexion,
- un champ servant à mettre des informations complémentaires appelé champ `GCOS` (ce champ peut contenir des caractères blancs) ;
- le répertoire de connexion de l'utilisateur,
- le fichier binaire exécutable à charger lors de la connexion (le plus souvent un shell).

Prog. 11.8 – Lecture avec format dans un fichier texte.

```
#include <stdio.h>

void main() {
   int i,res;
   char Nom[10], Passwd[16], Gcos[128], Rep[255], Shell[255];
   int Uid, Gid;

   FILE* TheFic=fopen("\verb"d:\temp\passwd"","r");
   if(TheFic == NULL) {
       printf("Ouverture␣impossible␣%s␣\n","passwd");
       return 1;
   }
   while(!feof(TheFic)){
       res = fscanf(TheFic,"%[^:]:",Nom);
       if(res != 1) break;
       res = fscanf(TheFic,"%[^:]:",Passwd);
       if(res != 1) break;
       res = fscanf(TheFic,"%d:",&Uid);
       if(res != 1) break;
       res = fscanf(TheFic,"%d:",&Gid);
       if(res != 1) break;

       for(i=0; i<128; i++){
           res = fgetc(TheFic);
           if(res == ':') {
               Gcos[i] = '\0';
               break;
           }
           else
              Gcos[i] = res;
       }

       res = fscanf(TheFic,"%[^:]:",Rep);
       if(res != 1) break;
       res = fgetc(TheFic);
       if(res != '\n') {
           ungetc(res,TheFic);
           res = fscanf(TheFic,"%s",Shell);
           if(res != 1) break;
       }
       else
           Shell[0] = '\0';
       printf("%s␣%s␣%d␣%d\n", Nom, Passwd, Uid, Gid,);
       printf("%s␣%s␣%s\n", Gcos, Rep ,Shell);
   }

   fclose(TheFic);
}
```

11.6 Déplacement dans le fichier

Jusqu'à maintenant nous avons vu des fonctions qui modifiaient de manière automatique le pointeur courant dans le fichier correspondant (adresse de l'octet dans le fichier à partir duquel se fera la

prochaine opération d'E/S). Nous allons voir les fonctions qui permettent de connaître la valeur de cette position courante dans le fichier et de la modifier. Ces fonctions associées à la position dans le fichier sont :

`int fseek(FILE*, long, int) ;`

synopsis : change la position courante dans le fichier.

arguments :

1. descripteur de fichier ouvert ;
2. déplacement à l'intérieur du fichier en nombre d'octets ;
3. point de départ du déplacement. Cet argument peut prendre les valeurs suivantes qui selon la norme doivent être définies dans le fichier `<stdio.h>` mais sont souvent dans le fichier `<unistd.h>` sur les machines de type "UNIX system V" :

   SEEK_SET : le déplacement est alors relatif au début du fichier ;

   SEEK_CUR : le déplacement est alors relatif à la position courante ;

   SEEK_END : le déplacement est alors relatif à la fin du fichier ;

retour : `0` en cas de succès

conditions d'erreur : une valeur différente de zéro est retournée si le déplacement ne peut pas être réalisé.

`long ftell(FILE*) ;`

synopsis : retourne la valeur de la position courante dans le fichier.

argument : descripteur de fichier ouvert ;

retour :

1. sur les fichiers binaires : nombre d'octets entre la position courante et le début du fichier.
2. sur les fichiers texte : une valeur permettant à `fseek` de re-positionner le pointeur courant à l'endroit actuel.

conditions d'erreur : la valeur `-1L` est retournée, et la variable `errno` est modifiée.

`int fgetpos(FILE*, fpos_t*) ;`

synopsis : acquiert la position courante dans le fichier.

arguments :

1. descripteur de fichier ouvert ;
2. référence d'une zone permettant de conserver la position courante du fichier (le type `fpos_t` est souvent un type équivalent du type entier long) ;

retour : `0` en cas de succès

conditions d'erreur : une valeur différente de `0` est retournée, et la variable `errno` est modifiée.

`int fsetpos(FILE*, const fpos_t*) ;`

synopsis : change la position courante dans le fichier.

arguments :

1. descripteur de fichier ouvert ;
2. référence d'une zone ayant servi à conserver la position courante du fichier par un appel précédent à `fgetpos()` ;

retour : `0` en cas de succès

conditions d'erreur : une valeur différente de `0` est retournée, et la variable `errno` est modifiée.

`void rewind(FILE*) ;`

synopsis : si le descripteur de fichier ouvert `fp` est valide, `rewind(fp)` est équivalent à `(void) fseek(fp,0L,0)`.

Pour illustrer le déplacement à l'intérieur d'un fichier, nous allons prendre pour exemple la modification de l'âge des voitures dans le fichier `FicParcAuto` vu précédemment. Le programme 11.9 réalise la modification d'un enregistrement dans un fichier en procédant de la manière suivante :

1. il lit un enregistrement du fichier dans une zone en mémoire ;
2. il modifie la zone en mémoire ;
3. il replace le pointeur courant du fichier sur le début de l'enregistrement pour pouvoir réécrire cet enregistrement ;
4. il écrit la zone mémoire dans le fichier.

Prog. 11.9 – Modifications par déplacement dans un fichier.

```
#include <stdio.h>
#include <stddef.h>
struct auto {
   int age;
   char couleur[20], numero[10], type[10], marque[10];
} UneAuto;

void main(int argc, char** argv) {

   FILE* TheFic=fopen(argv[1], "rb+");
   if (TheFic == NULL) {
        printf("Ouverture impossible %s \n", argv[1]);
        return 1;
   }

   size_t fait;
   for(int i=0; i<20; i++){
        fait=fread(&UneAuto, sizeof(UneAuto), 1, TheFic);
        if(fait != 1){
             printf("Erreur lecture fichier parcauto\n");
             return 2;
        }
        UneAuto.age++;
        fait=fseek(TheFic, -sizeof(UneAuto), SEEK_CUR);
        if(fait != 0){
             printf("Erreur deplacement fichier parcauto\n");
             return 3;
        }
        fait=fwrite(&UneAuto, sizeof(UneAuto), 1, TheFic);
        if(fait != 1) {
             printf("Erreur ecriture ; fait=%d \n", fait);
             return 4;
        }
    }
    fclose(TheFic);
}
```

Cette modification est réalisée par le programme 11.9 selon les instructions C suivantes :

- la ligne 18 correspond à une lecture d'un enregistrement du fichier dans la zone mémoire `UneAuto` du type `struct auto`.

- la ligne 23 modifie la valeur du champ `age` dans la structure en mémoire ;
- la ligne 24 modifie la position courante du fichier pour positionner le pointeur courant à l'adresse de début de l'enregistrement qui est en mémoire.
- la ligne 27 écrit dans le fichier le contenu de la zone mémoire `UneAuto`. Cette écriture provoque la modification de l'enregistrement sur disque.

Ce même exemple peut aussi être réalisé avec les fonctions `fgetpos()` et `fsetpos()` comme le montre le programme 11.10. La différence majeure entre ces deux exemples vient du fait que, dans la version 11.10, le programme conserve l'information permettant de re-positionner le pointeur courant dans le fichier, alors que dans le programme 11.9 le programme revient en arrière de la taille d'un enregistrement. Les fonctions `fgetpos()` et `fsetpos()` sont plus appropriées pour des déplacements dans un fichier avec des tailles d'enregistrement variables.

Prog. 11.10 – Déplacements dans un fichier avec `fgetpos()`.

```
void main(int argc, char** argv) {

    FILE* TheFic=fopen(argv[1], "rb+");
    if (TheFic == NULL) {
        printf("Ouverture␣impossible␣%s␣\n", argv[1]);
        return 1;
    }

    size_t fait;
    fpos_t CurPos;
    for(int i=0; i<20; i++) {
        fait=fgetpos(TheFic, &CurPos);
        if(fait != 0){
            printf("Erreur␣acquisition␣Position\n");
            return 2;
        }
        fait=fread(&ParcAuto[i], sizeof(struct auto), 1, TheFic);
        if(fait != 1){
            printf("Erreur␣lecture␣fichier␣parcauto\n");
            return 3;
        }
        ParcAuto[i].age++;
        fait=fsetpos(TheFic, &CurPos);
        if(fait != 0){
            printf("Erreur␣restitution␣Position\n");
            return 4;
        }
        fait=fwrite(&ParcAuto[i], sizeof(struct auto), 1, TheFic);
        if(fait != 1){
            printf("Erreur␣ecriture␣;␣fait=%d␣\n", fait);
            return 5;
        }
    }
    fclose(TheFic);
}
```

11.7 Gestion des tampons

Les E/S sont en général bufferisées[5]. Dans le cas général, l'allocation du buffer se fait de manière automatique lors de la première entrée-sortie. Il est cependant possible d'associer un buffer avec un fichier ouvert par les fonctions décrites ci-après. Cette association doit être faite avant tout échange dans le fichier ouvert. Le buffer se trouve dans l'espace adressable de l'utilisateur. Les appels de fonctions associées à la présence d'un buffer sont :

`void setbuf(FILE*, char*) ;`

synopsis : associe un buffer à un fichier ouvert, dans le cas où le pointeur est `NULL`, les E/S du fichier sont non bufferisées (chaque échange donne lieu à un appel système).

arguments :

1. descripteur de fichier ouvert ;
2. adresse d'une zone mémoire destinée à devenir le buffer d'E/S associé au fichier ouvert, cette zone doit avoir une taille pré-définie dont la valeur est `BUFSIZE`. Elle peut être égale au pointeur NULL, ce qui rend les E/S du fichier non bufferisées.

`int setvbuf(FILE*, char*, int, size_t) ;`

synopsis : contrôle la gestion de la bufferisation d'un fichier ouvert avant son utilisation.

arguments :

1. descripteur de fichier ouvert ;
2. adresse d'une zone mémoire destinée à devenir le buffer d'E/S associé au fichier ouvert, cette zone doit avoir la taille donnée en quatrième argument. Si l'adresse est égale à `NULL`, la fonction alloue de manière automatique un buffer de la taille correspondante.
3. le type de bufferisation, ce paramètre peut prendre les valeurs suivantes définies dans `<stdio.h>` :
   _IOFBF : signifie que les E/S de ce fichier seront totalement bufferisées (par exemple les écritures n'auront lieu que lorsque le tampon sera plein).
   _IOLBF : signifie que les E/S seront bufferisées ligne par ligne (i.e. dans le cas de l'écriture, un retour chariot provoque l'appel système).
   _IONBF : les E/S sur le fichier sont non bufferisées.
4. la taille de la zone mémoire (buffer).

`int flush(FILE*) ;`

synopsis : vide le buffer associé au fichier ;

argument : descripteur de fichier ouvert en mode écriture ou en mode mise-à-jour. Ce descripteur peut être égal à `NULL` auquel cas l'opération porte sur l'ensemble des fichiers ouverts en écriture ou en mise-à-jour ;

retour : `0` dans le cas normal et `EOF` en cas d'erreur.

conditions d'erreur : la fonction retourne `EOF` si l'écriture physique s'est mal passée.

Les E/S sur les terminaux sont bufferisées ligne par ligne. La fonction `fflush()` permet de forcer l'écriture des dernières informations.

5. — Anglicisme que l'on peut traduire par tamponnées. Prenons le cas des écritures. Elles sont d'abord réalisées dans un espace mémoire que l'on appelle tampon (buffer en anglais). Les caractères sont ensuite transférés au système en bloc. Ceci permet de minimiser les appels au système (car le tampon de caractères est dans l'espace mémoire du programme) et a priori d'améliorer les performances. Cette gestion de tampon intermédiaire peut se traduire par une représentation non exacte de l'état du programme par les écritures. En effet, les écritures sont différées et le tampon d'écriture n'est vidé que lorsqu'une fin de ligne est transmise.

11.8 Gestion des erreurs

Les erreurs des fonctions d'E/S peuvent être récupérées par le programme. Des variables sont associées aux erreurs sur chaque flux d'entrée-sortie. Ces variables ne sont pas directement modifiables mais elles sont accessibles à travers un ensemble de fonctions qui permettent de les tester ou de les remettre à zéro.

De plus, pour donner plus d'informations sur les causes d'erreur, les fonctions d'E/S utilisent une variable globale de type entier appelé `errno`. Cette variable est aussi utilisée par les fonctions de bibliothèque servant à réaliser les appels système. La valeur de `errno` n'est significative que lorsqu'une opération a échoué et que l'appel de fonction correspondant a retourné une valeur spécifiant l'échec.

Les fonctions associées à la gestion des erreurs sont :

```
int ferror(FILE*) ;
```

synopsis : Cette fonction retourne une valeur différente de zéro si la variable qui sert à mémoriser les erreurs sur le fichier ouvert correspondant a été affectée lors d'une opération précédente.

argument : le descripteur de fichier ouvert pour lequel la recherche d'erreur est faite.

retour : une valeur différente de zéro si une erreur s'est produite.

```
int feof(FILE*) ;
```

synopsis : Cette fonction teste si l'indicateur de fin de fichier a été affecté sur le fichier ouvert correspondant au descripteur passé en argument.

argument : le descripteur de fichier ouvert sur lequel le test de fin de fichier est désiré.

retour : retourne vrai si la fin de fichier est atteinte.

```
void clearerr(FILE*) ;
```

synopsis : Cette fonction efface les indicateurs de fin de fichier et d'erreur du fichier ouvert correspondant au descripteur passé en argument.

argument : le descripteur de fichier ouvert pour lequel on désire effacer les valeurs de la variable mémorisant les erreurs et de la variable servant à mémoriser la rencontre de la fin de fichier.

retour :

conditions d'erreur :

```
void perror(const char*) ;
```

synopsis : Cette fonction fait la correspondance entre la valeur contenue dans la variable `errno` et une chaîne de caractères qui explique de manière succincte l'erreur correspondante.

argument : Cette fonction accepte un argument du type chaîne de caractères qui permet de personnaliser le message.

Pour illustrer cette gestion d'erreur, nous allons ajouter des tests d'erreur dans l'exemple de parcours du fichier `FicParcAuto` avec modification de l'âge dans les différents champs (Prog. 11.11).

Prog. 11.11 – Gestion des cas d'erreurs pendant la manipulation d'un fichier.

```
int main() {

    FILE* TheFic = fopen("FicParcAuto", "r+b");
    if(TheFic == NULL) {
        perror("Ouverture␣impossible␣FicParcAuto");
        return 1;
    }

    size_t fait;
    while( 1 ){
        fait=fread(&ParcAuto, sizeof(ParcAuto), 1, TheFic);
        if(fait != 1) {
            if(feof(TheFic))
                fprintf(stderr, "Fin␣de␣fichier␣FicParcAuto\n");
            else
                fprintf(stderr, "Erreur␣lecture␣FicParcAuto\n");
            break;
        }
        ParcAuto.age++;
        fait=fseek(TheFic, -sizeof(ParcAuto), SEEK_CUR);
        if (fait != 0){
            perror("Erreur␣deplacement␣FicParcAuto");
            break;
        }
        fait=fwrite(&ParcAuto, sizeof(ParcAuto), 1, TheFic);
        if(fait != 1){
            fprintf(stderr, "Erreur␣ecriture␣;␣fait=%d␣\n",fait);
            break;
        }
        fflush(TheFic);
    }
    clearerr(TheFic);
    fclose(TheFic);

    return 0;
}
```

Chapitre 12

Autres fonctions de la bibliothèque standard

Sommaire

12.1 Fonctions de manipulation de chaînes de caractères

La bibliothèque standard, par le biais de la librairie `<string.h>`, fournit des fonctions de manipulation de chaînes de caractères. Rappelons qu'une chaîne de caractères est un tableau de caractères contenant des caractères quelconques et terminée par le caractère '\0'.

`int strlen(char* s)` ; renvoie la longueur de la chaîne `s` (le caractère de terminaison de chaîne n'est pas compté).

`strcpy(char* chdest, char* chsource)` ; copie `chsource` dans `chdest`.

`strcnpy(char* chdest, char* chsource, int len)` ; copie les `len` premiers caractères de `chsource` dans `chdest`.

`char* strcat(char* ch1, char* ch2)` ; concaténe `ch2` après `ch1`.

`char* strncat(char* ch1, char* ch2, int len)` ; concaténe `ch2` après `len` caractères de `ch1`.

`char* strchr(char* ch, char c)` ; délivre un pointeur sur la première apparition de `c` dans la chaîne `ch` ou `NULL`.

`char* strrchr(char* ch, char c)` ; délivre un pointeur sur la dernière occurence de `c` dans la chaîne `ch` ou `NULL`.

`char* strstr(char* ch, char* ssch)` ; délivre un pointeur sur la première sous-chaîne `ssch` contenue dans la chaîne `ch` ou `NULL`.

`int strcmp(char* ch1, char* ch2)` ; renvoie zéro si `ch1==ch2`, un nombre strictement supérieur à zéro si `ch1>ch2` et un nombre strictement inférieur à zéro sinon.

`int strncmp(char* ch1, char* ch2, int len)` ; même chose que précédemment mais la comparison se fait sur `len` premiers caractères uniquement.

Ces deux dernières fonctions font la différence entre majuscules et minuscules, si l'on veut faire abstraction des maj/min utiliser `int stricmp(char* ch1, char* ch2)` et `int strncmp(char* ch1, char* ch2, int len)`. Il existe également d'autres fonctions, mais elles sont en général très peu utilisées.

12.2 Types de caractères

Il existe des macros expressions définies dans `<ctype.h>` qui permettent de déterminer ou de changer le type d'un caractère. Ces macros expressions de test retournent un résultat non nul si le test est vrai.

- `isalpha(c)` est vrai si `c` est une lettre (alphabétique),
- `isupper(c)` est vrai si `c` est une majuscule (upper case),
- `islower(c)` est vrai si `c` est une minuscule (lower case),
- `isdigit(c)` est vrai si `c` est un chiffre,
- `isspace(c)` est vrai si `c` est un blanc, une interligne ou une tabulation,
- `ispunct(c)` est vrai si `c` est un caractère de ponctuation,
- `isalnum(c)` est vrai si `c` est un caractère alphabétique ou numérique,
- `isprint(c)` est vrai si `c` est un caractère imprimable de `040` à `0176`,
- `isgraph(c)` est vrai si `c` est un caractère graphique de `041` à `0176`,
- `iscntrl(c)` est vrai si `c` est le caractère del ou suppr(`0177`), ou un caractère de contrôle (`<040`),
- `isascii(c)` est vrai si `c` est un caractère ASCII (`<0200`, cf. table en annexe A).

Les macros qui transforment un caractère :

- `toupper(c)` retourne le caractère majuscule correspondant à `c`,
- `tolower(c)` retourne le caractère minuscule correspondant à `c`,
- `toascii(c)` masque `c` avec `0x7f`.

12.3 Fonctions mathématiques

Le fichier d'en-tête `<math.h>` contient des déclarations de fonctions et de macros mathématiques.

Les macros `EDOM` et `ERANGE` (qui se trouvent dans `<errno.h>`) sont des constantes entières non nulles utilisées pour signaler des erreurs de domaine et d'intervalle pour les fonctions ; `HUGE_VAL` est une valeur positive de type `double` :

- Il se produit une erreur de domaine si un argument n'est pas dans le domaine sur lequel la fonction est définie. En cas d'erreur de domaine, `errno` reçoit `EDOM`, la valeur de retour dépend de l'implémentation.
- Il se produit une erreur d'intervalle si le résultat de la fonction ne peut pas être représenté par le type `double`. Si le résultat est trop grand, la fonction retourne `HUGE_VAL` avec le signe approprié, et `errno` reçoit `ERANGE`. Si le résultat est trop petit, la fonction retourne `0`, c'est l'implémentation qui détermine si `errno` reçoit `ERANGE` ou pas.

Dans le tableau 12.1, `x` et `y` sont de type `double`, `n` de type `int`, et toutes les fonctions retournent un `double`. Pour les fonction trigonométriques, les angles sont exprimés en radians.

Tableau 12.1 – Fonctions mathématiques.

Fonction	Définition
`sin(x)`	sinus de x
`cos(x)`	cosinus de x
`tan(x)`	tangente de x
`asin(x)`	arc sinus de x, dans l'intervalle $[-\pi/2, \pi/2], x \in [-1, 1]$
`acos(x)`	arc cosinus de x, dans l'intervalle $[0, \pi], x \in [-1, 1]$
`atan(x)`	arc tangente de x, dans l'intervalle $[-\pi/2, \pi/2]$
`atan2(x,y)`	arc tangente de y/x, dans l'intervalle $[-\pi, \pi]$
`sinh(x)`	sinus hyperbolique de x
`cosh(x)`	cosinus hyperbolique de x
`tanh(x)`	tangente hyperbolique de x
`exp(x)`	fonction exponentielle e^x
`log(x)`	logarithme népérien : $ln(x), x > 0$
`log10(x)`	logarithme de base 10 : $log_{10}(x), x > 0$
`pow(x,y)`	x^y. Il se produit une erreur de domaine si $x = 0$ et $y \leq 0$, ou si $x < 0$ et y n'est pas un entier
`sqrt(x)`	$\sqrt{x}, x \geq 0$
`ceil(x)`	le plus petit entier supérieur ou égal à x, exprimé en `double`
`floor(x)`	le plus grand entier inférieur ou égal à x, exprimé en `double`
`fabs(x)`	valeur absolue de x, $\|x\|$
`ldexp(x,n)`	$x\,2^n$
`frexp(x, int* exp)`	sépare x en une fraction normalisée dans l'intervalle $[1/2, 1]$, qui est retournée, et une puissance de 2, qui est placée dans `*exp`. Si x est nul, les deux parties du résultat sont nulles.
`modf(x, double* ip)`	sépare x en ses parties entière et fractionnaire, toutes deux du même signe que x. Cette fonction place la partie entière dans `*ip` et retourne la partir fractionnaire.
`fmod(x, y)`	reste de x/y, exprimée en virgule flottante, de même signe que x. Si y est nul, le résultat dépend de l'implémentation.

12.4 Fonctions utilitaires

Le fichier d'en-tête `<stdlib.h>` contient des déclarations de fonctions traitant la conversion de nombres, l'allocation de mémoire, et des opérations similaires.

`double atof(const char* s)` ; `atof` convertit `s` en un `double` ; équivaut à `strod(s, (char**) NULL)`.

`double atoi(const char* s)` ; `atoi` convertit `s` en un `int` ; équivaut à `(int) strol(s, (char**) NULL), 10`.

`double atol(const char* s)` ; `atol` convertit `s` en un `long` ; équivaut à `strol(s, (char**) NULL), 10`.

`double strtod(const char* s, char** endp)` ; `strtod` convertit `s` en un `double` sans tenir compte des caractères d'espacement de tête. Elle place dans `*endp` un pointeur sur la partir non convertie de `s`, si elle existe, sauf si `endp` vaut `NULL`. Si la réponse est trop grande, la fonction retourne `HUGE_VAL` avec le signe approprié. Si al réponse est trop petite, la fonction retourne `0`. Dans les deux cas, `errno` reçoit `ERANGE`.

`double strtol(const char* s, char** endp, int base)` ; `strtol` convertit `s` en un `long` sans tenir compte des caractères d'espacement de tête. Elle place dans `*endp` un pointeur sur la partir non convertie de `s`, si elle existe, sauf si `endp` vaut `NULL`. Si la base est comprise entre `2` et `36`, la conversion s'effectue en considérant que l'entrée est écrite dans cette base. Si `base` vaut `0`, la base est `8`, `10` ou `16` ; un `0` en tête indique le format octal, et un `0x` ou un `0X`, le format hexadécimal. Des lettres majuscules ou minuscules représentent des chiffres compris entre `10` et `base-1`. En base `16`, on peut placer un `0x` ou `0X` en tête du nombre. Si la réponse est trop grande, la fonction retourne `LONG_MAX` ou `LONG_MIN`, selon le signe du résultat, et `errno` reçoit `ERANGE`.

`unsigned long strtoul(const char* s, char** endp, int base)` ; `strtoul` est la même fonction que `strtol`, mis à part que le résultat est de type `unsigned long`, et que la valeur de retour en cas d'erreur est `ULONG_MAX`.

`int rand(void)` ; `rand` retourne un entier pseudo-aléatoire compris entre `0` et `RAND_MAX`, `RAND_MAX` vaut au minimum `32767`.

`void srand(unsigned int seed)` ; `srand` prend `seed` comme amorce de la nouvelle séquence de nombres pseudo-aléatoires. L'amorce intiale vaut `1`.

`void* calloc(size_t nobj, size_t size)` ; `calloc` retourne un espace mémoire réservé à un tableau de `nobj` objets, tous de taille `size`, ou bien `NULL` si cette demande ne peut pas être satisfaite. La mémoire allouée est initialisée par des `0`.

`void* malloc(size_t size)` ; `malloc` retourne un espace mémoire réservé à un objet de taille `size`, ou bien `NULL` si cette demande ne peut pas être satisfaite. La mémoire allouée est n'est pas initialisée.

`void* realloc(void* p, size_t size)` ; `realloc` change en `size` la taille de l'objet pointé par `p`. Si la nouvelle taille est plus petite que l'ancienne, seul le début du contenu de l'objet est conservé. Si la nouvelle taille est plus grande, le contenu de l'objet est conservé, et l'espace mémoire supplémentaire n'est pas initialisé. `realloc` retourne un pointeur sur le nouvel espace mémoire, ou bien `NULL` si cette demande ne peut être satisfaite, auquel cas `*p` n'est pas modifié.

`void free(void* p)` ; `free` libère l'espace mémoire pointé par `p`. Elle ne fait rien si `p` vaut `NULL`. `p` doit être un pointeur sur l'espace alloué par `calloc`, `malloc` ou `realloc`.

`void abort(void)` ; `abort` provoque un arrêt anormal du programme.

`void exit(int status)` ; `exit` provoque l'arrêt normal du programme. Les fonctions `atexit` dans l'ordre inverse de leur enregistrement, l'écriture des tampons associés aux fichiers ouverts est forcée, les flots ouverts sont fermés et le contrôle est rendu à l'environnement. La façon dont `status` est retourné à l'environnement dépend de l'implémentation, mais la valeur `0` indique que le programme qui s'arrête a rempli sa mission. On peut aussi utiliser les valeurs `EXIT_SUCCESS` et `EXIT_FAILURE`, pour indiquer respectivement la réussite ou l'échec du programme.

`int atexit(void (*fcn)(void))` ; `atexit` enregistre que la fonction `fcn` devra être appelée lors de l'arrêt normal du programme. Elle retourne une valeur non nulle si cet enregistrement n'est pas réalisable.

`int system(const char* s)` ; `system` passe la chaîne `s` à l'environnement pour que celui-ci l'exécute. Si `s` vaut `NULL`, `system` retourne une valeur non nulle si un interpréteur de commandes est présent. Si `s` ne vaut pas `NULL`, la valeur de retour dépend de l'implémentation. Exemple : `system("dir")` ;.

`char* getenv(const char* name)` ; `getenv` retourne la chaîne d'environnement associée) `name` ou `NULL` si cette chaîne n'existe pas. Les détails dépendent de l'implémentation.

`int abs(int i)` ; `abs` retourne la valeur absolue de son argument de type `int`.

`long labs(long n)` ; `labs` retourne la valeur absolue de son argument de type `long`.

`div_t div(int num, int denom);` `div` calcul le quotient et le reste de la division de `num` par `denom`. Le quotient et le reste sont placés respectivement dans les champs `quot` et `rem`, de type `int`, d'une structure de type `div_t`.

`ldiv_t ldiv(long num, long denom);` `ldiv` calcul le quotient et le reste de la division de `num` par `denom`. Le quotient et le reste sont placés respectivement dans les champs `quot` et `rem`, de type `long`, d'une structure de type `ldiv_t`.

Bien que n'étant pas une instruction du langage mais un appel à une fonction du système, il est intéressant de considérer le `exit()` comme une rupture de séquence. L'instruction `return` provoque la fin d'une fonction ; de même l'appel à la fonction `exit()` provoque la fin du programme. Cette fonction `exit()` peut être aussi associée à une expression. Cette expression est évaluée et la valeur obtenue est retournée au processus père. La valeur `0` signifie que le programme s'est bien passé.

Prenons l'exemple du programme 12.1 qui est sensé ouvrir un fichier. Si ce fichier est absent, la fonction impose l'arrêt brutal du programme.

Prog. 12.1 – Utilisation de l'appel système `exit()`.

```
FILE ouvre(const char* nom_fichier) {
    FILE fid = fopen(nom_fichier, 'r');
    if( fid == NULL ) {
        printf("Ouverture impossible %s\n", nom_fichier);
        exit(1);
    }
    return fid;
}
```

12.5 Fonctions de dates et heures

Le fichier d'en-tête `<time.h>` contient des déclarations de types et de fonctions servant à manipuler la date et l'heure. Certaines fonctions traitent l'heure locale, qui peut être différente de l'heure calendaire (fuseaux horaires). `clock_t` et `time_t` sont des types arithmétiques qui représentent des instants, et `struct tm` contient les composantes d'une heure calendaire :

```
int tm_sec;         // seconde (0-59)
int tm_min;         // minute (0-59)
int tm_hour;        // heure (0-23)
int tm_mday;        // jour (1-31)
int tm_mon;         // mois (0-11)
int tm_year;        // année
int tm_wday;        // jour depuis dimanche (0-6)
int tm_yday;        // jour depuis le 1er janvier (0-365)
int tm_isdst;       // drapeau d'heure d'été (0 ou 1)
```

`tm_isdst` est positif si l'heure d'été est en vigueur, nul sinon, et négatif si cette information n'est pas disponible.

`clock_t clock(void);` `clock` retourne le temps d'utilisation du processeur par le programme depuis le début de son exécution, ou bien `-1` si cette information n'est pas disponible. `clock() / CLK_TK` est une durée en secondes.

`time_t time(time_t* tp);` `time` retourne l'heure calendaire actuelle, ou bien `-1` si l'heure n'est pas disponible. Si `tp` est différent de `NULL`, `*tp` reçoit aussi cette valeur de retour.

`double difftime(time_t time2, time_t time1);` `difftime` retourne la durée `time2-time1`, exprimée en secondes.

`time_t mktime(struct tm* tp);` `mktime` convertit l'heure locale contenue dans la structure `*tp` en heure calendaire, exprimée suivant la même représentation que celle employée par `time`. Les valeurs des composantes de l'heure seront comprises dans les intervalles donnés ci-dessus. La fonction `mktime` retourne l'heure calendaire, ou bien `-1` si celle-ci ne peut pas être représentée.

Les quatre fonctions suivantes retournent des pointeurs sur des objets statiques qui peuvent être écrasés par d'autres appels.

`char* asctime(const struct tm* tp);` `asctime` convertit l'heure représentée dans le structure `*tp` en une chaîne de la forme : `Sun Jan 3 15 :14 :13 1998\n\0`.

`char* ctime(const time_t* tp);` `ctime` convertit l'heure calendaire `*tp` en heure locale. Elle équivaut à : `asctime(localtime(tp))`.

`struct tm* gmtime(const time_t* tp);` `gmtime` convertit l'heure calendaire `*tp` en temps universel (TU). Elle retourne `NULL` si le TU n'est pas disponible.

`struct tm* localtime(const time_t* tp);` `localtime` convertit l'heure calendaire `*tp` en heure locale.

`size_t strftime(char* s, size_t max, const char* fmt, const struct tm* tp);` `strftime` transforme les informations de date et d'heure contenue dans `*tp` suivant le format `fmt`, qui est analogue à un format de `printf`, et place le résultat dans `s` (cf. [KR94] pour le détail des formats).

12.6 Messages d'erreur

On utilise la macro `assert` (dans le fichier `<assert.h>`) pour insérer des messages d'erreur dans les programmes : `void assert(int exp)`. Si `exp` vaut `0` au moment où `assert(exp)` est exécutée, la macro insert imprime sur la console un message de la forme :

`Assertion failed : exp, file nom_de_fichier, line nn`

Puis, elle appelle `abort` pour arrêter l'exécution du programme. Le nom du fichier source et le numéro de ligne sont donnés par les macros `__FILE__` et `__LINE__` du préprocesseur.

Si `NDEBUG` est défini au moment où `<assert.h>` est inclus, la macro `assert` n'est pas prise en compte.

Annexe A

Table des caractères ASCII

b7 b6 b5 / Bits b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	Controle		Symboles Nombres		Majuscule		Minuscule	
0 0 0 0	0 NUL 0	16 DLE 20	32 SP 40	48 0 60	64 @ 100	80 P 120	96 ‘ 140	112 p 160
0 0 0 1	1 SOH 1	17 DC1 21	33 ! 41	49 1 61	65 A 101	81 Q 121	97 a 141	113 q 161
0 0 1 0	2 STX 2	18 DC2 22	34 ” 42	50 2 62	66 B 102	82 R 122	98 b 142	114 r 162
0 0 1 1	3 ETX 3	19 DC3 23	35 # 43	51 3 63	67 C 103	83 S 123	99 c 143	115 s 163
0 1 0 0	4 EOT 4	20 DC4 24	36 $ 44	52 4 64	68 D 104	84 T 124	100 d 144	116 t 164
0 1 0 1	5 ENQ 5	21 NAK 25	37 % 45	53 5 65	69 E 105	85 U 125	101 e 145	117 u 165
0 1 1 0	6 ACK 6	22 SYN 26	38 & 46	54 6 66	70 F 106	86 V 126	102 f 146	118 v 166
0 1 1 1	7 BEL 7	23 ETB 27	39 ’ 47	55 7 67	71 G 107	87 W 127	103 g 147	119 w 167
1 0 0 0	8 BS 10	24 CAN 30	40 (50	56 8 70	72 H 110	88 X 130	104 h 150	120 x 170
1 0 0 1	9 HT 11	25 EM 31	41) 51	57 9 71	73 I 111	89 Y 131	105 i 151	121 y 171
1 0 1 0	10 LF 12	26 SUB 32	42 * 52	58 : 72	74 J 112	90 Z 132	106 j 152	122 z 172
1 0 1 1	11 VT 13	27 ESC 33	43 + 53	59 ; 73	75 K 113	91 [133	107 k 153	123 { 173
1 1 0 0	12 FF 14	28 FS 34	44 , 54	60 < 74	76 L 114	92 \ 134	108 l 154	124 | 174
1 1 0 1	13 CR 15	29 GS 35	45 − 55	61 = 75	77 M 115	93] 135	109 m 155	125 } 175
1 1 1 0	14 SO 16	30 RS 36	46 . 56	62 > 76	78 N 116	94 ^ 136	110 n 156	126 ~ 176
1 1 1 1	15 SI 17	31 US 37	47 / 57	63 ? 77	79 O 117	95 ¨ 137	111 o 157	127 DEL 177

Legende : dec CHAR hex oct

Annexe B

Mots réservés du C

Tableau B.1 – Liste des mots réservés.

Type	Classe	Instruction	Opérateur	Étiquette
`int`	`auto`	`if`	`sizeof`	`case`
`char`	`extern`	`else`		`default`
`short`	`static`	`while`		
`long`	`register`	`do`		
`unsigned`	`typedef`	`for`		
`float`		`switch`		
`double`		`break`		
`struct`		`continue`		
`union`		`goto`		
`enum`		`return`		
`void`				

Annexe C

Quelques pointeurs sur Internet

Sommaire

Voici quelques pointeurs sur le langage C qui peuvent être bien utiles. À consulter ... Je ne garantit ni la pérennité de ces sites, ni l'exhaustivité des adresses proposées ! Je maintiens également une page Web contenant une collection d'adresses sur des cours et des librairies en langage C et C++ : `http://www.enspm.u-3mrs.fr/derrode/SiteInfo/SiteInfo.html`.

C.1 Quelques cours de programmation

Les quelques liens proposés ici permettent d'accéder à un grand nombre de support de cours à télécharger (formats postscript ou pdf), ou a consulter « on line » (html).

- `http://www.geocities.com/CapeCanaveral/Lab/2922/book.html`, Guides de programmation tous langages : C, C++, Java, Javascript, VRML, ...
- `http://www.esil.univ-mrs.fr/~tourai/main/Enseignement.html`, Guides de programmation, langages C, C++, Java, Javascript, VRML, ..., M. Touraïvane, Esil (Université de Marseille).
- `http://handy.univ-lyon1.fr/service/cours/lamot/cours.c.html`, Cours de langage C, G.-H. Lamot.
- `http://www-igm.univ-mlv.fr/~dr/C_CPP_index.html`, Cours d'informatique, Université de Marne-La-Vallée.
- `http://physinfo.ulb.ac.be/cit_courseware/cscourse.htm`, Cours de programmation (en anglais).
- `http://www.lysator.liu.se/c/c-www.html` Liste de pointeurs sur des cours, notamment en C (en anglais).
- `http://fr.dir.yahoo.com/informatique_et_multimedia/langages_de_programmation/`, Yahoo : Langages de programmation.
- `http://www.cprogramming.com/` C++ Programming tutorial (en anglais).
- `http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_toc.html` The GNU C Library, compatible C ANSI.

C.2 Librairies scientifiques et graphiques

Les librairies proposées ci-dessous ne sont pas forcément toutes compatibles avec la programmation sous VC++, à vérifier. L'installation de ces librairies n'est pas forcément aisée et nécessite sans doute un peu d'expérience ...

- `http://fltk.org/index.html`, The Fast Light Tool Kit, pour développer une interface graphique, indépendante de la plateforme de développement (Unix, Linux, Windows, ...).
- `http://public.kitware.com/`, VTK, The Visualization toolkit (graphisme - 3D).
- `http://www.nr.com/`, Numerical recipes, algorithmes scientifiques (optimization, tri, transformée de Fourier rapide, ...).
- `http://sources.redhat.com/gsl/`, The GNU Scientific Library, algorithmes scientifique : transformée de Fourier, pseudo-générateurs de nombres aléatoires, fonctions spéciales, ...

C.3 Sources et sites de programmeurs

Quelques sites où vous pourrez trouver du code source et des astuces intéressantes.

- `http://www.mathtools.net/`, Programmation scientifique en Matlab, C, C++, Java, ...
- `http://www.devx.com/sourcebank/`, SourceBank, banque de codes tous langages
- `http://www.codeguru.com/`, Code Guru (visual C++, Microsoft Foundation classes)

Liste des figures

Liste des tableaux

Liste des programmes

Bibliographie

[98990] ISO/IEC 9899. *Programming language C.* ISO/IEC, 1990.

[Dij68] E.W. Dijkstra. GO TO statements considered harmful. *Communications of the ACM*, 11(3) :147–148, Mars 1968.

[JR78] S.C. Johnson and D.M. Ritchie. Portability of C programs and the UNIX operating system. *Bell System Technical Journal*, 57(6) :2021–2048, July/August 1978.

[KR94] B.W. Kernighan and D.M. Ritchie. *Le langage C ANSI, version française.* Manuels informatiques. Masson, Paris, deuxième edition, 1994. Les solutions aux exercices sont proposées chez le même éditeur par C.L. Tondo et S.E. Gimpel [TG95].

[RK78] D.M. Ritchie and B.W. Kernighan. *The C programming language.* Prentice Hall Inc., Englewood cliffs, New Jersey, Mars 1978.

[Str86] B. Stroustrup. *The C++ programming language.* Prentice Hall Inc., Englewood cliffs, New Jersey, 1986.

[TG95] C.L. Tondo and S.E. Gimpel. *Le langage C ANSI - Solutions.* Manuels informatiques. Masson, Paris, deuxième edition, 1995. Solutions aux exercices proposés dans le livre de B.W. Kernighan et D.M. Ritchie [KR94].

[Wir74] N. Wirth. On the composition of well structured programs. *ACM computing survey*, 6(4) :247–259, Décembre 1974.