Support de Cours de Langage C

Christian Bac

6 novembre 2000

Table des matières

Avant-propos

Voici les conventions d'écriture suivies dans ce support :

– le *style italique* n'est quasiment jamais utilisé car je ne l'aime pas, je le réserve pour les phrases en anglais comme *The C programming Language* ;
– les `caractères` de type imprimante à boule (teletype) sont utilisés pour les noms de fichiers et les parties de programme, comme `/usr/include/stdio.h` ;
– le **style gras** est utilisé lorsqu'un mot me semble important, par exemple **instructions** ;
– les caractères sont plus larges dans l'énoncé de règles ou de parties de syntaxe telles que la syntaxe de la conversion explicite : (type) expression ;
– les mots qui ne sont pas français mais sont souvent utilisés dans le jargon des informaticiens comme "quote" sont considérés comme de type neutre et se voient appliquer le genre masculin. Ainsi je peux appeler un index-noeud du système de fichiers UNIX : un inode.
– dans les dessins représentant l'espace mémoire occupé par les variables ou les tableaux, les noms entourés par des pointillés sont des adresses et pas des variables.

Voici quelques mots que j'utilise de manière courante :

Bit (Binary digiT) ou élément binaire, la plus petite partie manipulable par un ordinateur. Comme son nom l'indique un élément binaire peut prendre deux valeurs 0 ou 1.

Octet ou *byte*, le groupe de bits qui permet de supporter la représentation d'un caractère. Ce groupe est le plus souvent (comme son nom l'indique) constitué de huit bits. En langage C, l'octet sert dans les calculs d'occupation d'espace mémoire et en particulier les variables plus compliquées comme le mot machine ont une taille donnée en nombre d'octets.

Pile ou pile d'exécution, c'est une partie de l'espace mémoire d'une application qui permet au programme de tenir à jour des séries de variables actives. Cet espace est géré comme une pile d'assiettes, c'est-à-dire que l'on peut ajouter de l'espace (faire grandir la pile) ou diminuer cet espace. Les opérations d'agrandissement ou de réduction de la pile sont faites de manière automatique lors des appels de fonctions et respectivement des retours de fonctions. La pile est gérée à partir d'un espace mémoire de taille fixe qui est attribué de manière automatique par le système d'exploitation. Le processeur tient à jour sa relation avec la pile à travers un registre interne qui décrit le sommet de pile et un autre registre qui maintient un lien sur le contexte (arguments de l'appel et variables locales).

Ce cours de langage C a la généalogie suivante :

Génèse
1. Il a été créé en 1985 lorsque j'étais au centre de calcul du CNET Paris A, pour assurer la formation en interne.
2. La première version était «mise en page» grâce à une version d'Emacs sur Multics améliorée par M. Arditti. Cette version d'Emacs permettait de taper les accents.
3. Les exercices associés ont été réalisés sur une machine de type SM90.
4. Les personnes m'ayant aidé dans la réalisation de cette première version sont : M. Auguste et M. Sabatier du CNET.

Troff
1. Lorsque le DWB (Documentation WorkBench) a été disponible suite à une intense participation du CNET à travers la personne de M. Gien, dans l'organisation de la conférence Européenne sur UNIX à Paris en 1986, j'ai décidé de faire migrer ce cours sous une version troff.
2. J'ai alors choisi d'utiliser les macros qui me semblaient devoir être la future référence car elles faisaient partie de la distribution d'UNIX system V, les macros mm (je crois que j'ai été le seul).
3. Les premières figures sont apparues dans ce support grâce à la disponibilité de l'utilitaire "pic".
4. Les personnes m'ayant aidé dans la réalisation de cette version sont : M. Fondain et M. Horn du CNET.

Utilisation en formation continue Cette version m'a suivi lorsque j'ai quitté le CNET pour venir travailler à l'INT en 1989. Elle a alors servi de support de cours pour la mise en place d'un ensemble de cours dus à l'arrivée du système UNIX dans les services opérationnels, par le service national de la formation de France Télécom.

Macintosh J'ai arrété de modifier cette version en 1991, pour deux raisons :
1. premièrement, j'ai écrit des addendas avec Microsoft Word pour Macintosh. Ces addendas avaient pour but d'expliquer les modifications apportées par la normalisation du langage.
2. deuxièmement, j'ai constitué un jeu de transparents à partir de ce cours en supprimant des phrases et en modifiant les polices de caractères de manière à le rendre lisible une fois projeté. Ce jeu de transparents est devenu mon nouveau support de cours et je l'ai fait évolué de manière indépendante.

LaTeX Finalement, j'ai décidé en 1997 de reconstituer ce support en utilisant LaTeX, avec plusieurs buts :
1. pour moi, ce fut un très bon apprentissage de LaTeX ;
2. grâce à des outils comme LaTeX2HTML, je peux intégrer ce support dans un cours interactif sur le World Wide Web ;
3. j'ai mesuré la limite des transparents surtout lorsqu'ils sont utilisés par les élèves comme support de cours.

Je tiens à souligner la participation active de Mmes. Monget et Carpentier, Ms. Volt et Lalevée dans la relecture et la correction de ce support. Je suis aussi redevable, à ce dernier de moult conseils relatifs à l'utilisation de LaTeX2e.

Chapitre 1

Historique et présentation

Ce chapitre essaye de placer le langage C [BK78] dans son contexte historique et technique, de manière à approcher l'état d'esprit du langage qui nous permet de deviner les règles sous-jacentes au langage sans pour cela avoir à les mémoriser.

1.1 Historique

Le langage C [DR78] est lié à la conception du système UNIX [1] par les Bell-Labs. Les langages ayant influencé son développement sont :

- le langage BCPL de M. Richards 1967 ;
- le langage B développé aux Bell-Labs 1970.

Ces deux langages partagent avec le langage C :

- les structures de contrôle ;
- l'usage des 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 langage **C** sont les suivantes :

- 1970 diffusion de la famille PDP 11.
- 1971 début du travail sur le langage C, car le PDP 11 peut adresser un octet ce qui pose un problème vis-à-vis du langage B.
- 1972 la première version de C est écrite en assembleur par Brian W. Kernighan et Dennis M. Ritchie.
- 1973 Alan Snyder écrit un compilateur C portable (thèse MIT).
- 1975 Steve C. Johnson écrit et présente le PCC (Portable C Compiler). C'était à l'époque le compilateur le plus répandu. Le PCC a longtemps assuré sa propre norme puisqu'il était plus simple de porter le PCC que de réécrire un compilateur C (25 % du code du PCC à modifier).
- 1987 début de la normalisation du langage par l'IEEE avec constitution d'un comité appelé : X3 J-11.
- 1989 sortie du premier document normalisé appelé norme ANSI X3-159.
- 1990 réalisation du document final normalisé auprès de l'ISO : ISO/IEC 9899 [ISO90] ;
- 1999 première révision de la norme.

[1] A l'époque UNIX était une marque déposée des laboratoires Bell. Aujourd'hui, c'est devenu un nom quasiment commun :-)

Jusqu'en 1987, il n'y avait pas de norme. Le livre "The C Programming Language" [BK78] de B. W. Kernighan et D. M. Ritchie définit le langage. Ce livre contient une description précise du langage appelée *"C Reference Manual"*. Ce livre qui a lui aussi été remis au gout du jour en 1988 [BK88] ainsi que la norme ISO [ISO90] sont à la base de ce cours.

Le livre de Philippe Dax [Dax92] est lui aussi une bonne introduction pour l'apprentissage de ce langage.

1.2 Présentation du langage

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

Le langage C a été conçu pour l'écriture de systèmes d'exploitation et du logiciel de base. Plus de 90 % du noyau du système UNIX est écrit en langage C. Le compilateur C lui-même est écrit en grande partie en langage C ou à partir d'outils générant du langage C [SJ78]. Il en est de même pour les autres outils de la chaîne de compilation (assembleur, éditeur de liens, préprocesseur). De plus, tous les utilitaires du système 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. Par le biais des bases de données, il est utilisé dans les applications de gestion. De nombreux logiciels du domaine des ordinateurs personnels, tels que Microsoft Word ou Microsoft Excel, sont eux-aussi écrits à partir de langage C ou de son successeur orienté objet, C++ [Str86].

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. C'est-à-dire qu'un programme de langage C peut être constitué de plusieurs modules. Chaque module est un fichier source que l'on peut compiler de manière autonome pour obtenir un fichier objet. L'ensemble des fichiers objets participants à un programme doivent être associés pour constituer un fichier exécutable.

Lorsque nous parlons du langage C, dans cette première partie, nous faisons référence à ce que sait faire le compilateur lui-même. Comme nous le verrons plus loin dans la section 1.4, plusieurs outils interviennent dans la transformation d'un ensemble de fichiers sources, constituant un programme, en un fichier binaire exécutable, qui est le résultat de ce que l'on appelle communément la compilation. Il serait plus juste de dire les compilations suivies par les traductions d'assembleur en objet, suivies de la réunion des fichiers objets.

> Règle fondamentale
>
> Le langage C se limite aux fonctionnalités qui peuvent être traduites efficacement en instructions machine.

La règle de fonctionnement édictée ci-dessus doit permettre de détecter ce qui est fait directement par le compilateur lui-même et ce qui ne peut pas être fait. Essayons d'illustrer cette règle par quelques exemples :

- Le compilateur C est capable de générer des instructions machine qui permettent de manipuler des éléments binaires ou des groupes d'éléments binaires (bits). Il peut donc réaliser des masques pour sélectionner des groupes de bits, ou encore faire des décalages de bits à l'intérieur de mots machine.

- Il permet les manipulations algébriques (addition, multiplication, etc.) de groupes d'octets qui représentent des valeurs entières ou des valeurs avec point décimal.
- Il permet de manipuler des caractères en considérant qu'un caractère est représenté par un octet à partir du code ASCII[2].
- Il ne permet pas de manipuler directement des tableaux. En particulier, il ne permet par 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 dont le nom commence par **str** comme `strcat()`.
- 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, depuis la normalisation du langage. 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 opérations plus compliquées, le programmeur doit écrire ses propres fonctions ou faire appel aux fonctions prédéfinies de la bibliothèque du langage C (voir chapitres 16 et 17). Ces fonctions sont, elles aussi, standardisées. La rédaction de la norme a supprimé un petit problème qui venait de la référence sous-jacente aux opérations possibles sur le processeur du PDP 11 de Digital Equipment sur lequel ont été conçues les premières versions du compilateur C. Cette machine était cependant suffisamment générale pour que cela ne transparaisse pas.

1.3 Idées fondamentales

Il est difficile de présenter les différents concepts du langage indépendemment les uns des autres, c'est pourquoi, certains concepts, comme la visibilité des variables, sont abordés dans les premiers chapitres pour n'être approfondis que plus tard.

Nous allons décrire les principales composantes d'un programme écrit en langage C. Comme il est montré dans 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. Ces fichiers sources sont aussi appelés modules et ce type de programmation est appelé programmation modulaire. Les notions de visibilités associées à la programmation modulaire sont approfondies dans le chapitre 9.

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

Prenons pour exemple un programme qui enregistre une série de noms d'élèves. Chaque nom d'élève est suivi d'une note. Le programme trie la liste des élèves par rang de notes ou par ordre alphabétique et affiche la liste des élèves triée par rang de notes. Avant de se terminer, il stocke la liste de élèves et leur note par rang alphabétique dans un fichier sur disque. Ce type d'application peut se découper en trois parties :

- la partie interactive qui échange les informations avec l'utilisateur ;
- la partie de calcul qui dans ce cas est un tri (par note ou par nom) ;
- la partie stockage des données sur disque une fois ces données triées.

Chacune des parties peut être regroupée dans un ou plusieurs fichiers de langage C que l'on appelle aussi module. Chaque fichier (voir fig. 1.2) contient les éléments suivants dans un ordre quelconque :

- des références à des variables ou des fonctions externes (sous forme de déclarations). Ces références décrivent les types des variables ou l'interface des fonctions utilisées dans ce module mais définies dans d'autres modules ;

[2] ASCII : Americans Standard Code for Information Interchange. ISO 646 :1983, Information processing - ISO 7-bit coded character set for information interchange.

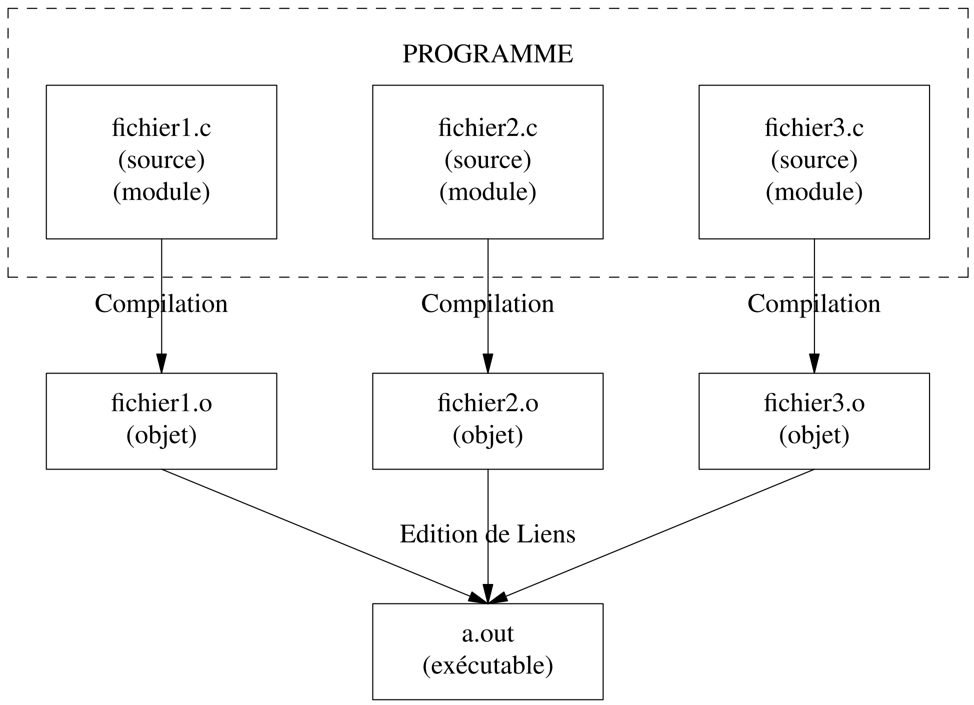

FIG. 1.1 – Structure d'un programme C

- des définitions de variables globales et de fonctions, qui peuvent être référencées[3] dans les autres fichiers ;
- des lignes de directives de compilation (pour le préprocesseur).

Une fonction (Fig. 1.3) est construite à partir :

- d'une interface constituée du type et du nom de la fonction suivis par les types et les noms des paramètres ;
- d'un bloc, appelé aussi corps de la fonction.

La fonction `main()` est particularisée, en ce sens que l'exécution du fichier binaire exécutable, conçu à partir de l'ensemble des fichiers source, commence par elle.

Un bloc est constitué :

- d'une accolade ouvrante ;
- des définitions des variables locales au bloc ;
- des instructions ;
- d'une accolade fermante.

Une instruction peut-être :

- un bloc[4] ;
- ou une expression[5] suivie de " ;" ;
- ou une instruction de contrôle de flot (test, boucle, etc.).

Nous parlerons en détail des possibilités du préprocesseur dans le chapitre 15.

[3]Nous verrons plus en détail l'ensemble des possibilités associées aux déclarations et aux définitions dans les chapitres 3 et 9.

[4]On voit apparaître une définition récursive à l'intérieur du langage : à savoir un bloc contient des instructions qui peuvent être des blocs qui contiennent des instructions, ...

[5]Les expressions sont explicitées dans le chapitre 5.

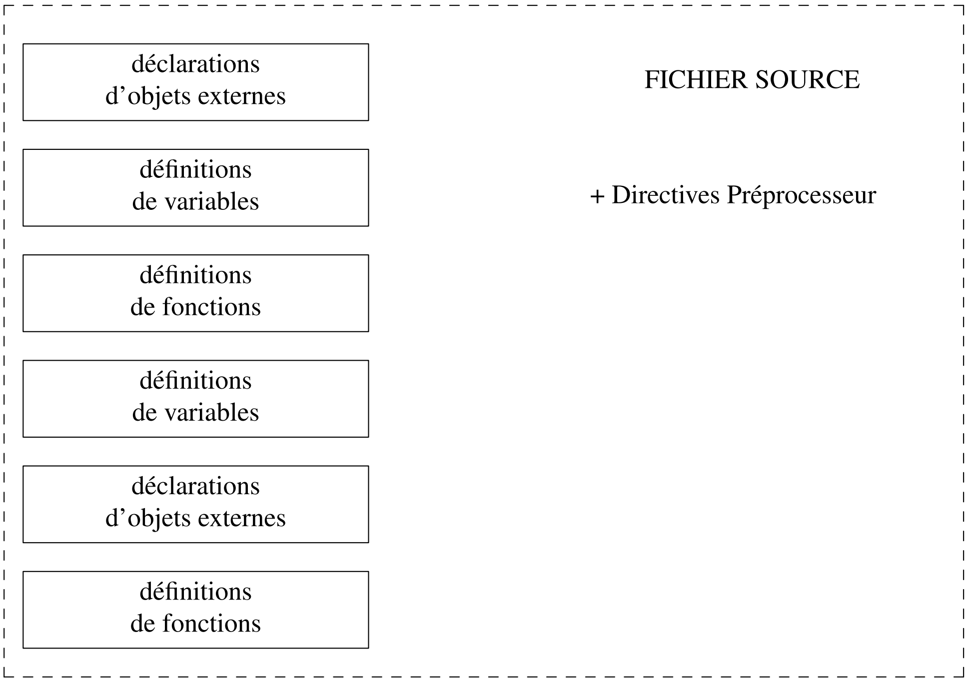

FIG. 1.2 – Fichier source

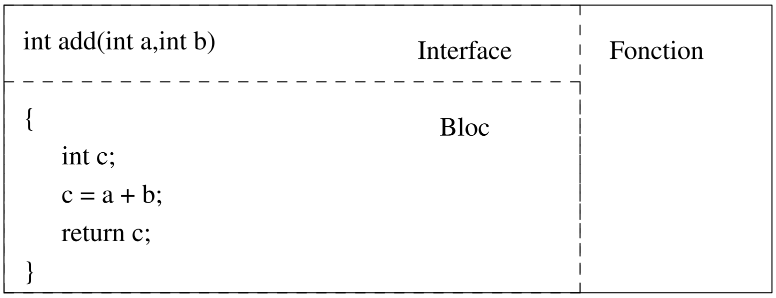

FIG. 1.3 – Structure d'une fonction C

1.4 Enchaîneur de passes

L'étude des différentes actions entreprises lors de ce que nous appelons de manière abusive la compilation, permet de mieux comprendre quelles caractéristiques du programme sont apportées par quel outils.

Nous prenons le cas d'un système non intégré car les actions sont facilement séparables. Dans un système de type UNIX, pour obtenir un fichier exécutable à partir d'un source C, la commande usuelle est `cc` [6] :

cc options nom_ du_fichier.c

Ces "compilateurs" sont en fait des enchaîneurs de passes ; nous allons voir l'usage, le fonctionnement, et les options de `cc`. Il sert à appeler les différents utilitaires nécessaires à la transformation d'un programme C en un fichier exécutable. L'enchaineur de passes cc met en œuvre cinq utilitaires :

– le préprocesseur, que nous appellerons `cpp` ;

[6] cc ou tout autre "compilateur" comme gcc.

– le compilateur C, que nous appellerons `c0+c1` car il peut être découpé en deux parties ;
– l'optimiseur de code, appelé `c2` ;
– l'assembleur, que nous nommerons `as` ;
– l'éditeur de liens, dont le nom le plus courant est `ld`.

1.4.1 Préprocesseur

Le préprocesseur est un utilitaire qui traite le fichier source avant le compilateur. C'est un manipulateur de chaînes de caractères. Il retire les parties de commentaires, qui sont comprises entre "/*" et "*/". Il prend aussi en compte les lignes du texte source ayant un "#" en première colonne pour créer le texte que le compilateur analysera. Ses possibilités sont de trois ordres (voir chap. 15) :
– inclusion de fichiers ;
– définition d'alias et de macrofonctions ;
– sélection de parties de texte ;

1.4.2 Compilateur

Le compilateur lit ce que génère le préprocesseur et crée les lignes d'assembleur correspondantes. Le compilateur lit le texte source une seule fois du début du fichier à la fin. Cette lecture conditionne les contrôles qu'il peut faire, et explique pourquoi toute variable ou fonction doit être déclarée avant d'être utilisée.

1.4.3 Optimiseur de code

L'optimiseur de code élimine les parties du code assembleur qui ne sont pas utiles. Il remplace des séquences d'instructions par des instructions plus sophistiquées et propres au processeur. Cette opération donne un code plus compact et plus rapide. Ensuite, il optimise les sauts. Dans les premières versions de compilateur, il est arrivé que sur certaines machines l'optimisation crée de petits problèmes qui se résument par : après l'optimisation, le programme ne marche plus.

1.4.4 Assembleur

L'assembleur prend le code généré par le compilateur, éventuellement modifié par l'optimiseur, et génère un fichier en format relogeable. Ce fichier possède des références insatisfaites qui seront résolues par l'éditeur de liens. Sur les machines utilisant un système de type UNIX, ce fichier est suffixé par **.o** [7].

1.4.5 Editeur de liens

L'éditeur de liens prend le ou les fichiers en format relogeable et les associe pour créer un module chargeable. Il se sert de bibliothèques pour résoudre les références indéfinies, en particulier la bibliothèque standard (`libc.a`). Il utilise aussi un module spécial, `crt0.o`, qui contient le code de démarrage du programme.

Par défaut sur un système de type UNIX, l'éditeur de lien met le résultat de l'édition de liens dans un fichier qu'il appelle `a.out`.

[7] De mauvaises langues prétendent que sur d'autres types de système un mauvais esprit aurait osé les suffixer par **.OBJ**

1.4.6 Quelques options de cc

La figure 1.4 donne un schéma de fonctionnement de `cc`, relativement aux options qui lui sont passées. Les **options** de l'enchaineur de passes sont précédées d'un tiret[8] ("-"). Voici les options les plus couramment utilisées :

-c provoque la génération d'un module objet non exécutable, il s'arrête avant l'édition de liens.
`cc -c toto.c` → `toto.o`

-E lance le préprocesseur seul, `cpp`, qui écrit sur la sortie standard ou génère un fichier suffixé par ".i".
`cc -E toto.c` → `stdout ou toto.i`

-S génère le fichier assembleur après passage du préprocesseur et du compilateur. Le fichier est suffixé par ".s".
`cc -S toto.c` → `toto.s`

-O optimise le code généré (utilisation de c2).

-o nom donne le nom au fichier exécutable au lieu de a.out.
`cc -o toto toto.c` → `toto`

-s enlève la table des symboles du fichier exécutable.

-n génère un code exécutable pur, partageable par plusieurs processus (code réentrant).

-v option bavarde, `cc` annonce ce qu'il fait.

Deux options sont utiles sur le compilateur GNU gcc pour forcer la vérification d'une syntaxe correspondant à la norme ANSI :

-ansi avec cette option le compilateur se comporte comme un compilateur de langage C ANSI sans extensions de langage correspondant au C GNU ;

-pedantic cette option demande au compilateur de refuser la compilation de programme non ansi.

L'utilisation de ces deux options est recommandée lors de l'apprentissage du langage C en utilisant le compilateur gcc.

[8] Selon la tradition du système UNIX.

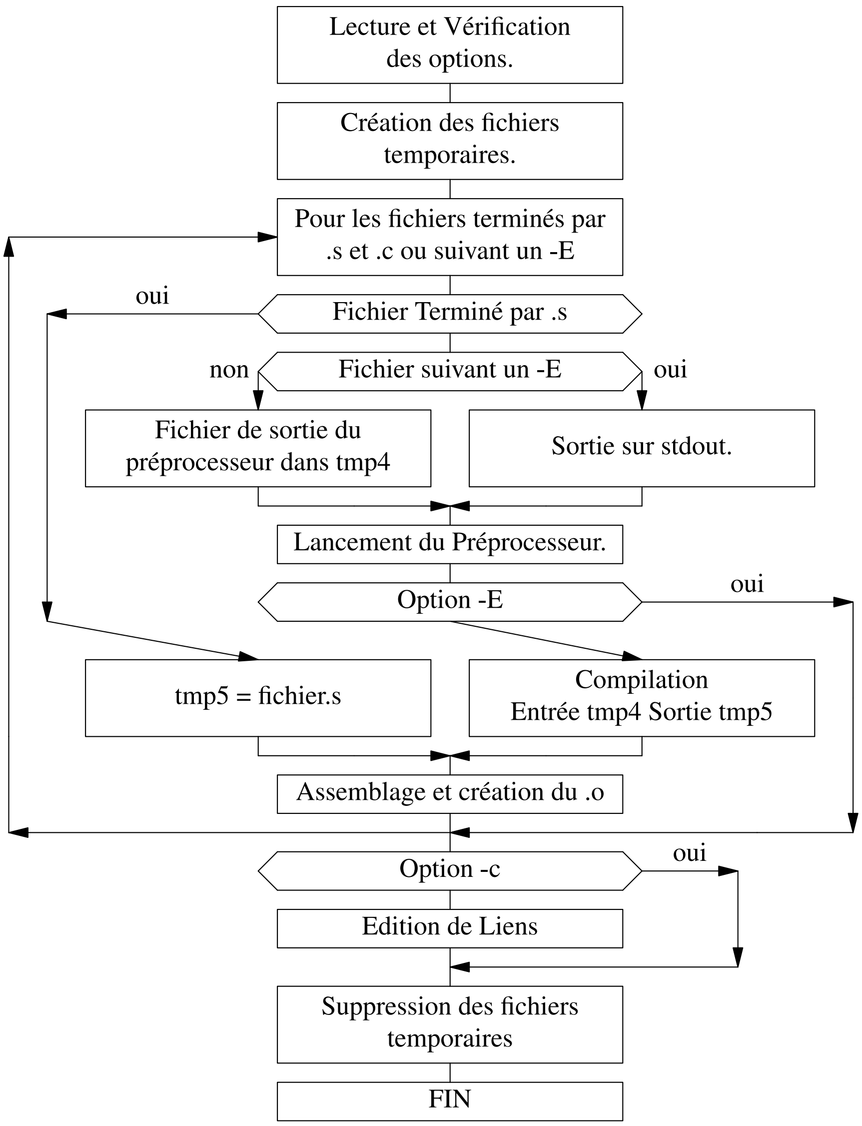

Fig. 1.4 – Schéma de fonctionnement de CC

Chapitre 2

Généralités sur la syntaxe

Ce chapitre introduit les premiers concepts indispensables à la compréhension d'un programme C, à savoir les règles qui constituent la syntaxe de base du langage. Ces rêgles sont utilisées par les compilateurs pour déterminer si une série de caractères correspond à un mot réservé, à un nom ou à une constante.

2.1 Mise en page

Le format du texte est libre. La mise en page n'a aucune signification pour le compilateur. Elle est importante pour la lisibilité du programme. Les lignes de directives de précompilation commencent par un "#".

2.1.1 Identificateurs

Un identificateur est construit selon le modèle :
- à partir de l'alphabet : "`a-z, A-Z, 0-9, _`" ;
- Il peut avoir jusqu'à trente et un caractères significatifs à l'intérieur d'une unité de compilation. La norme mentionne que les noms peuvent être limités à six caractères entre unités de compilation et que les éditeurs de liens peuvent ne pas tenir compte des majuscules et minuscules. Cette contrainte n'est pas suivie par les compilateurs modernes.
- Il commence par une lettre ou `_`.

2.1.2 Espaces lexicaux

Les différents identificateurs d'un programme en langage C sont classés dans les espaces lexicaux suivants :
- le premier espace contient les identificateurs relatifs aux types synonymes, variables et fonctions ;
- le second espace est réservé aux étiquettes pour les branchements inconditionnels ;
- le troisième espace est utilisé pour les noms de modèles de structures, d'unions ou d'énumérations ;
- pour chaque modèle de structure ou d'union, un espace de noms est créé pour contenir les noms de champs de la structure ou de l'union.

2.2 Mots réservés

Ce sont les mots prédéfinis du langage C. Ils ne peuvent être réutilisés pour des identificateurs. Ils sont associés avec différentes entités du langage :

structures de données `char const double float int long short signed unsigned void volatile`

classes d'allocation `auto extern register static`

constructeurs `enum struct typedef union`

instructions de boucle `do for while`

sélections `case default else if switch`

ruptures de séquence `break continue goto return`

divers `asm entry fortran sizeof`

2.3 Constantes

Les constantes servent dans l'expression des tailles, l'initialisation des variables et dans les expressions. Les constantes de type "chaîne de caractères" ont un statut particulier : elles permettent de constituer des tableaux de caractères sans nom. Les caractères de ces tableaux peuvent être constants.

Nous approfondirons l'expression des constantes dans la section 3.2 qui traite des types de variables. Voici cependant quelques exemples de constantes :
- constante entière : `10` ;
- constante flottante : `121.34` ;
- caractère simple : `'a'` ;
- chaîne de caractères : `"message"`.

2.4 Instructions

Une **instruction** est :
- soit une instruction simple,
- soit un instruction composée.

Une **instruction simple** est toujours terminée par un " ;". Les instructions composées sont contenues entre deux accolades.

Chapitre 3

Types et variables

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 visibilité 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 précédents numéros 1 et 2 sont associés au type de la variable ; les points 3 et 4 sont associés à la classe mémoire de la variable.

3.1 Types de base

Ce sont les types prédéfinis du compilateur. Ils sont au nombre de six[1] :

`void` c'est le type vide. Il a été introduit par la norme ANSI. Il est surtout utilisé pour préciser les fonctions sans argument ou sans retour. Il joue un rôle particulier dans l'utilisation des pointeurs (voir chapitre 10).

`int` c'est le type entier. Ce type se décline avec des qualificateurs pour préciser sa taille (`long` ou `short`), et le fait qu'il soit uniquement positif (`unsigned`) ou positif et négatif (`signed`) [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`.

`char` ce type est très proche de l'octet. Il représente un entier sur huit bits. Sa valeur peut évoluer entre -128 et +127. Il est le support des caractères au sens commun du terme. Ces caractères sont représentés par la table ASCII. Comme le type `int` le type `char` peut être qualifié de manière à être signé ou non signé. La norme ANSI introduit un type permettant de supporter des alphabets comprenant plus de 255 signes, ce type est appelé `wchar_t`. Il est défini dans le fichier `<stddef.h>`.

`float` ce type sert pour les calculs avec des parties décimales.

[1] Puisqu'on vous dit qu'il n'y a pas de type booléen en C !!!
[2] Dans le cas le plus courant une variable du type entier peut contenir une valeur positive ou négative.

`double` c'est un type qui permet de représenter des valeurs ayant une partie décimale avec une plus grande précision que le type `float`. Comme nous le verrons dans l'expression des constantes (sec. 3.2.2) et dans les calculs (sec. 3.8), ce type est le plus courant pour représenter des valeurs avec parties décimales.

`long double` ce type est récent, il permet de représenter des nombres avec parties décimales qui nécessitent une très grande précision.

3.1.1 Types entiers

Les mots `short` et `long` peuvent être utilisés seuls ou avec le mot `int`, donnant la possibilité d'avoir des définitions du type : `short int` ou `long int`. Ces définitions peuvent aussi s'écrire de manière abrégée : `short` ou `long`.

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 (5.3).

A priori, les types entiers sont signés ; c'est à dire qu'ils peuvent contenir des valeurs positives ou négatives. Par exemple, la valeur d'une variable du type `char` peut évoluer entre `-128` et `+127`.

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. Par exemple, la valeur d'une variable du type `unsigned char` ne peut évoluer qu'entre `0` et 255.

Le qualificateur `signed` permet d'insister sur le fait que la variable peut prendre des valeurs positives ou négatives. Il fait pendant au qualificateur `unsigned` comme l'opérateur "`+`" unaire fait pendant à l'opérateur "`-`" unaire.

3.1.2 Types avec parties décimales

Comme nous l'avons déjà dit, les types avec parties décimales sont au nombre de trois :

`float` ce type sert pour les calculs avec des parties décimales. Il est souvent représenté selon la norme ISO/IEEE 754.

`double` c'est le type de plus grande précision permettant de représenter des valeurs avec parties décimales. Lui aussi est souvent basé sur la norme ISO/IEEE 754.

`long double` ce type est récent et permet de représenter des nombres avec parties décimales sur une très grande précision, si la machine le permet.

3.1.3 Tailles des types

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 :

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

où `sizeof` est un opérateur qui donne la taille en nombre d'octets du type dont le nom est entre parenthèses.

La taille de l'entier est le plus souvent la taille des registres internes de la machine, c'est par exemple seize bits sur une machine de type ordinateur personnel[3] et trente-deux bits sur les machines du type station de travail.

[3] Pour des problèmes de compatibilité avec les anciennes versions, les compilateurs pour PC génèrent le plus souvent du code compatible 386.

Type **Année**	**PDP 11** **1970**	**PC 486** **1989**	**SUN4** **1993**	**Alpha** **1994**
char	8 bits	8bits	8bits	8bits
short	16 bits	16 bits	16 bits	16 bits
int	16 bits	16 bits	32 bits	32 bits
long	32 bits	32 bits	32 bits	64 bits
float	32 bits	32 bits	32 bits	32 bits
double	64 bits	64 bits	64 bits	64 bits
long double	64 bits	64 bits	64 bits	128 bits

TAB. 3.1 – Longueur des types de base sur quelques machines

La taille des variables ayant une partie décimale est le plus souvent cadrée sur la norme ISO/IEEE 754. Les machines supportant un type `long double` différent du type `double` sont assez rares.

Le tableau 3.1 donne quelques exemples de tailles pour des machines dont les registres ont les tailles 16, 32 et 64 bits.

3.2 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.

3.2.1 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 ou seize (octal ou hexadécimal).

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 du type `long`.

Les constantes peuvent être suffixées par un "l" ou "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 sans signe (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 :

  1. 120L, 0364L, 0x1faL
  2. 120l, 0364l, 0x1fal

– constante entière non signée :

  1. 120U, 0364U, 0x1faU

2. 120u, 0364u, 0x1fau

– constante longue entière non signée :
  1. 120UL, 0364UL, 0x1faUL, 120uL, 0364uL, 0x1fauL
  2. 120Ul, 0364Ul, 0x1faUl, 120ul, 0364ul, 0x1faul

3.2.2 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 avec la notation utilisant le point décimal, son type implicite est `double`.

12134e-2 la même constante exprimée en notation exponentielle

+12134E-2 la notation exponentielle accepte le "E" majuscule, et le "+" unaire.

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".

3.2.3 Constantes caractères

Les constantes du type caractère simple sont toujours entourées d'apostrophes (simples *quotes* aigu). Elles peuvent être représentées selon quatre méthodes :

1. lorsque le caractère est disponible au clavier sauf pour la barre de fraction inversée et l'apostrophe, le caractère correspondant est donné par 'car', par exemple `'a'`.
2. un caractère peut aussi être représenté par sa valeur exprimée dans la table ASCII en utilisant une notation en base huit. Cette valeur est précédée à l'intérieur des apostrophes par une barre de fraction inversée.
   '\0' : octet du nul, il sert à délimiter les fins de chaînes de caractères.
   '\012' : saut de ligne (*Line Feed*, LF) ;
   '\015' : retour chariot (*Carriage Return*, CR) ;
   '\011' : tabulation horizontale (*Horizontal Tabulation*, HT) ;
3. de même, un caractère peut être représenté par sa notation en base seize.
   '\x0' : caractère nul ;
   '\xA' : saut de ligne ;
   '\xC' : retour chariot ;
   '\x9' : tabulation horizontale ;
4. un certain nombre d'abréviations est aussi disponible :
   '\a' : alert (sonnerie, BEL) ;
   '\b' : backspace (BS) ;
   '\f' : saut de page (FF) ;
   '\n' : fin de ligne (LF) ;
   '\r' : retour chariot (CR) ;

m	e	s	s	a	g	e	\0

FIG. 3.1 – Chaîne de caractères constante

'*t*' : tabulation horizontale (HT) ;

'*v*' : tabulation verticale (VT) ;

Pour spécifier qu'une constante caractère est du type (`wchar_t`), elle doit être précédée d'un "L". Par exemple, `L'a'` est la constante de type caractère long contenant le caractère "a".

3.2.4 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 (tous les bits à 0) à partir des caractères contenus dans la chaîne. Cette suite d'octets peut être placée par le système dans une zone de mémoire en lecture seulement.

La zone correspondante est en fait un tableau de `char`. La chaîne est référencée par l'adresse du tableau. Par exemple, la chaîne de caractères `"message"` est générée par le compilateur selon le schéma de la figure 3.1.

3.3 Qualificatifs

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.

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.

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.

3.4 Taille et Normalisation

La taille des types entiers et avec partie décimale sont définies sur chaque machine à partir de deux fichiers :

`limits.h` pour les types entiers ;

`float.h` pour les types avec partie décimale.

Ces fichiers contiennent des définitions qui s'adressent au précompilateur et donnent les tailles et valeurs maximales des types de base et des types non signés.

Les programmes 3.1 et 3.2 sont respectivement des exemples de fichiers `<limits.h>` et `<float.h>`.

Programme 3.1 Exemple de fichier `limits.h`

```
/* Number of bits in a char.  */
#define CHAR_BIT     8
/* No multibyte characters supported yet.  */
#define MB_LEN_MAX     1

/* Min and max values a signed char can hold.  */
#define SCHAR_MIN     (-128)
#define SCHAR_MAX     127

/* Max value an unsigned char can hold.  (Min is 0).  */
#define UCHAR_MAX     255U

/* Min and max values a char can hold.  */
#define CHAR_MIN SCHAR_MIN
#define CHAR_MAX SCHAR_MAX

/* Min and max values a signed short int can hold.  */
#define SHRT_MIN (-32768)
#define SHRT_MAX 32767

/* Max value an unsigned short int can hold. (Min is 0). */
#define USHRT_MAX 65535U

/* Min and max values a signed int can hold.  */
#define INT_MIN (-INT_MAX-1)
#define INT_MAX 2147483647

/* Max value an unsigned int can hold.  (Min is 0).  */
#define UINT_MAX 4294967295U

/* Min and max values a signed long int can hold.  */
#define LONG_MIN (-LONG_MAX-1)
#define LONG_MAX 2147483647

/* Max value an unsigned long int can hold. (Min is 0). */
#define ULONG_MAX 4294967295U
```

Programme 3.2 Exemple de fichier `float.h`

```
/*      Float definitions */

#define FLT_MANT_DIG        24
#define FLT_EPSILON         1.19209290e-07f
#define FLT_DIG             6
#define FLT_MIN_EXP         -125
#define FLT_MIN             1.17549435e-38f
#define FLT_MIN_10_EXP      -37
#define FLT_MAX_EXP         128
#define FLT_MAX             3.40282347e+38f
#define FLT_MAX_10_EXP      38

/*      Double definitions */

#define DBL_MANT_DIG        53
#define DBL_EPSILON         2.2204460492503131e-16
#define DBL_DIG             15
#define DBL_MIN_EXP         -1021
#define DBL_MIN             2.2250738585072014e-308
#define DBL_MIN_10_EXP      -307
#define DBL_MAX_EXP         1024
#define DBL_MAX             1.79769313486231570e+308
#define DBL_MAX_10_EXP      308
```

La normalisation a aussi introduit un ensemble de types prédéfinis comme `size_t` qui est le type de la valeur retournée par l'opérateur `sizeof`. Les définitions exactes de ces types sont dans le fichier `<stddef.h>`. Le programme 3.3 est un extrait d'un exemple de fichier `<stddef.h>`.

Programme 3.3 Exemple de fichier `stddef.h`

```
typedef long ptrdiff_t;
typedef unsigned long size_t;
typedef int wchar_t;
```

3.5 Définition de variables

Nous appellerons **identificateur** soit un nom de fonction, soit un nom de variable. En langage C, une **définition** de variable a les rôles suivants :

définir le domaine de valeur de cette variable et les opérations légales sur cette variable ;
$\Longrightarrow$ grâce au type.

réserver l'espace mémoire nécessaire lors de l'exécution au support de la variable ;
$\Longrightarrow$ 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 (règles de visibilité).
$\Longrightarrow$ 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. 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 9 sur la visibilité. Les **classes mémoire** sont :

Déclaration	Classe mémoire
int a ;	définition d'une variable globale
main (int argc,	paramètres passés dans la pile,
char *argv[])	donc automatiques
{	
int b ;	définition d'une variable locale à main donc automatique
static char c[50] ;	variable locale à main mais implantée avec les globales
}	
extern int b ;	déclaration d'une variable qui est définie dans un autre fichier
	(rien à voir avec celui de main) variable globale externe
coucou(const int c)	paramètre constant, variable locale
{	la fonction coucou s'engage à ne pas modifier c
volatile char c ;	variable locale volatile
register int a ;	variable locale à coucou, à mettre en registre si possible
if (b == 1)	référence à la variable b externe
}	

TAB. 3.2 – Variables et classes mémoire

`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. 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.a.d 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 statique 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 statique 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. Il est de toute façon limité (4 pour les données, 4 pour les pointeurs sur un 680X0). Seules les variables locales peuvent être déclarées `register`.

Le tableau 3.2 donne quelques exemples de définitions et déclarations de variables.

3.6 Types dérivés

Un type dérivé est créé à partir de types standards pour l'usage propre à un programme. Les types dérivés sont les **tableaux**, les **structures** et les **pointeurs**.

3.6.1 Tableaux et structures

Les tableaux sont des paquets de données de même type. Ils sont reconnus par la présence de crochets ouvrants et fermants lors de la déclaration ou de la définition de l'objet. La taille du tableau est donnée entre les crochets lors de la définition. Pour simplifier le travail du compilateur, le rang des éléments du tableau ne peut évoluer qu'entre 0 et la taille du tableau -1.

Les structures sont des ensembles de données non homogènes. Les données peuvent avoir des types différents. Les structures sont déclarées ou définies selon le modèle :
- struct
- un nom de structure facultatif
- {
- la liste des données contenues dans la structure
- }
- la liste des variables construites selon ce modèle.

Prenons pour exemple les définitions suivantes :

```
int tab[10];
struct st1 {
    int a1;
    float b1;
    long c1;
} objst1;
```

Dans cet exemple :

1. `tab` est un tableau de 10 entiers, et les éléments du tableau sont référencés par `tab[0]` jusqu'à `tab[9]` ;
2. `st1` est un nom de modèle de structure et `objst1` est un objet de type `struct st1`. Les différentes parties de la structure `objst1` sont accessibles par `objst1.a1`, `objst1.b1` et `objst1.c1`.

Les tableaux et les structures sont parfois appelés agglomérats de données.

3.6.2 Pointeurs

Le **pointeur** est une variable destinée à contenir une adresse mémoire. Le compilateur connaissant la taille de l'espace adressable de la machine, il sait la taille nécessaire pour contenir un pointeur. Un pointeur est reconnu syntaxiquement par l'* lors de sa définition. Le pointeur est associé à un type d'objet. Ce type est celui des objets qui sont manipulables grâce au pointeur. Ce type est utilisé en particulier lors des calculs d'adresse qui permettent de manipuler des tableaux à partir de pointeurs (voir Chap. 10).

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[4] valeurs qui sont des adresses de variables du type entier (`int`).

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

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 (voir Chap. 9 pointeurs et tableaux) 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.

Si l'on déclare les variables de la manière suivante :

```
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`.

3.7 Initialisation

Les variables peuvent être initialisées lors de leur définition. Par défaut, les variables de type global sont initialisées avec la valeur qui correspond à tous les bits à zéro (0 pour un entier ou 0.0 pour un nombre à virgule flottante). Les variables locales, quant-à elles ne sont pas initialisées avec des valeurs par défaut. Vous veillerez donc à les initiliser vous même pour vous prémunir contre le risque d'utiliser une variable sans en connaitre la valeur initiale. Sur les compilateurs anciens, pour les catégories de variables de type tableau ou structure, seules les variables globales peuvent être initialisées.

Il est possible de réaliser des initialisations selon trois techniques suivant le type de la variable initialisée :

1. dans le cas d'une variable simple, il suffit de faire suivre la définition du signe égal (=) et de la valeur que l'on veut voir attribuée à la variable.
2. dans le cas de tableaux ou structures, il faut faire suivre la définition du signe égal suivi d'une accolade ouvrante et de la série de valeurs terminée par une accolade fermante. Les valeurs sont séparées par des virgules. Elles doivent correspondre aux éléments du tableau ou de la structure. La norme préconise que lorsque le nombre de valeurs d'initialisation est inférieur au nombre d'éléments à initialiser, les derniers éléments soient initialisés avec la valeur nulle correspondant à leur type.
3. les tableaux de caractères peuvent être initialisés à partir d'une chaîne de caractères constante.

Le tableau 3.3 donne des exemples d'initialisation.

3.8 Conversion de type

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

`int i = 10;`	Entier i initialisé à 10
`int j = 12,` `k = 3, l;`	entiers j initialisé à 12 ; k initialisé à 3 et l non initialisé.
`int *p1 = &i;`	Pointeur d'entier initialisé à l'adresse de i
`char d = '\n';`	Caractère initialisé à la valeur du retour chariot.

`float tf[10] = {`
`3.1, 4.5, 6.4,`
`9.3 };`
	Tableau de dix flottants dont les quatre premiers
	sont initialisés à 3.1 4.5 6.4
	et 9.3, les autres sont initialisés à 0.0.

`char t1[10] =` `"Coucou";`	Tableau de 10 caractères initialisé avec les caractères `'C' 'o' 'u' 'c' 'o' 'u' '\0'` Les trois derniers caractères sont aussi initialisés avec la valeur `'\0'`.

`struct tt1{`
`int i;`
`float j;`
`char l[20];`
`} obst = {`
`12,`
`3.14,`
`"qwertyuiop" };`
	modèle de structure contenant
	un entier,
	un flottant,
	et un tableau de 20 caractères.
	variable (`obst`) du type `struct tt1`,
	avec le premier champ (`obst.i`) initialisé à 12,
	le deuxième champ (`obst.j`) initialisé à 3.14,
	et le troisième champ (`obst.l`) initialisé à partir
	de la chaîne "qwertyuiop"

`char t2[] =` `"bonjour";`	Tableau de caractères initialisé avec la chaîne "bonjour". La taille du tableau est calculée selon le nombre de caractères + 1 (pour le nul).
`char t3[10] = { 'a',` `'b','c','d','e'};`	Tableau de 10 caractères dont les 5 premiers sont initialisés.
`char *p2 = t1;`	Pointeur sur un caractère initialisé à l'adresse du premier caractère du tableau.
`char *p3 =` `"Bonjour les";`	Pointeur sur un caractère initialisé à l'adresse de la chaîne. La chaîne peut être mise dans une zone de mémoire accessible en lecture seulement.

TAB. 3.3 – Exemples d'initialisations

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, additionné à d, le résultat est transformé en int et rangé dans i.

TAB. 3.4 – Exemples de conversion implicite

applique des règles de conversion implicite. Ces règles ont pour but la perte du minimum d'information dans l'évaluation de l'expression. Ces règles sont décrites dans la table ci-dessous.

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, des entiers courts, des champs de bits et des énumérations de la même façon que des entiers. Le principe permet d'utiliser à la place d'un entier :

- des caractères et des entiers courts signés ou non signés ;
- des champs de bits et des énumérations signés ou non signés.

Des exemples de conversions implicites sont donnés dans le tableau 3.4.

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 :

$$\boxed{\text{(type) expression}}$$

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 entre
`"i = f + d ;"`
et
`"i = (int) f + (int) d ;"`
du fait de la perte des parties fractionnaires.

3.9 Déclaration versus définition

Maintenant que nous connaissons les différents types du langage C, nous pouvons introduire la différence entre déclaration et définition. Cette différence sera vue de manière approfondie dans le chapitre 9. Nous appelerons :

- **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.

De manière simpliste, une déclaration fait référence à une définition dans une autre partie du programme. Elle peut se traduire par "je sais qu'il existe une variable ayant ce type et portant ce nom".

Chapitre 4

Eléments de base

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.

Ce modèle écran clavier qui est repris par le système UNIX est celui des interactions en langage C à partir des fichiers standard d'entrée et de sortie.

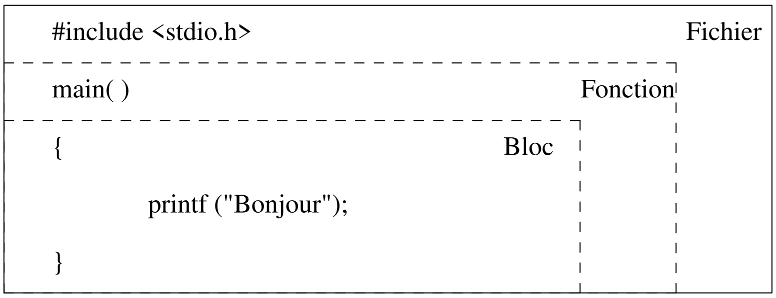

FIG. 4.1 – Programme qui écrit Bonjour

Les entrée-sorties en langage C ne sont pas prises en charge directement par le compilateur mais elles sont réalisées à travers des fonctions de bibliothèque. 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.

4.1 Bonjour

Le premier programme que tout un chacun écrit lorsqu'il apprend un nouveau langage répond à la question suivante : Comment écrire un programme principal qui imprime `Bonjour` ? Le programme[1] est donné dans la figure 4.1.

Il est constitué de la manière suivante :

[1] Pour rendre hommage comme il se doit aux concepteurs du langage, il aurait fallu faire écrire "Hello World !".

- la première ligne est une directive pour le préprocesseur qui demande à celui-ci de lire le fichier représentant l'interface standard des entrées sorties. Le contenu de ce fichier est décrit de manière approfondie dans le chapitre 16. Cette première ligne doit être présente dans tout module faisant appel à des fonctions d'entrées sorties et en particulier aux fonctions `printf()` et `scanf()` ;
- il n'y a pas de variable globale ;
- le programme contient une seule fonction `main()` qui joue le rôle de programme principal. Cette fonction est utilisée sans argument ;
- la définition de la fonction `main()` commence comme pour toute fonction par une accolade ouvrante ;
- elle ne contient pas de variable locale ;
- la seule instruction correspond à un appel de fonction qui provoque l'écriture sur le terminal. Cette instruction est une demande d'écriture d'une chaîne de caractères (constante) ;
- la fonction `main()` est terminée par une accolade fermante.

La fonction `printf()`[2] est une fonction[3] qui reçoit un nombre d'arguments variable. Ces arguments sont transformés en chaîne de caractères. Cette transformation fait appel à une notion de format pour les entiers, réels. Comme nous le verrons plus loin, le format définit le type de la donnée et le mode de représentation de la donnée.

4.2 Lire et écrire

Une fonction `scanf()` fait le pendant à la fonction `printf()`. Elle permet de lire des valeurs sur le clavier. Le programme 4.1 est un exemple de lecture et d'écriture à l'écran d'une chaîne de caractères.

Programme 4.1 Lecture et écriture de chaîne par scanf() et printf()

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

Ce programme ne contient toujours pas d'argument à main(). Il contient cependant la définition d'une variable globale. Cette variable est un tableau de quatre-vingt caractères destiné à recevoir les caractères lus au clavier. Il n'y a toujours pas de variable interne. Les seules instructions sont les appels aux fonctions de lecture et d'écriture (scanf() et printf()).

4.3 Quelques opérations

Les affectations peuvent être écrites comme dans de nombreux autres langages informatiques. En voici quelques exemples :

```
a = b + c ;
```

[2]La fonction `printf()` fait partie de la bibliothèque standard. Sur un système de type UNIX, l'interface qui décrit cette fonction est contenue dans le fichier `stdio.h` et le module objet correspondant est contenu dans le fichier `libc.a`. Ces fichiers sont stockés dans des parties de l'arborescence qui dépendent de l'installation du compilateur (voir chap. 16). Le fichier `stdio.h` est utilisé lors de la compilation pour vérifier la cohérence d'utilisation des fonctions. Le module objet est associé au programme lors de l'édition de liens (chap. 1).

[3]La notion de fonction sera explicitée au chapitre 5.

`%d`	entier décimal
`%f`	flottant
`%c`	caractère (1 seul)
`%s`	chaîne de caractères

TAB. 4.1 – Conversions usuelles de printf et scanf

```
a = b * c ;
a = b / c ;
a =  b - c ; etc
```

4.4 Plus sur `printf()` et `scanf()`

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 caractère qui la représente est exprimée en base 16 (hexadécimal).

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 "%".

Les types de conversion les plus usuels sont donnés dans la table 4.1.

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 16.5.

Le tableau 4.2 donne un résumé des déclarations de variables et des formats nécessaires à leurs manipulations pour printf et scanf. Le & est un opérateur du langage C dont nous parlerons plus tard. Pour scanf, il faut le mettre devant le nom de la variable, sauf pour les variables du type tableau de caractères.

L'exemple 4.2, montre qu'il est possible de faire l'écriture ou la lecture de plusieurs variables en utilisant une seule chaîne de caractères contenant plusieurs descriptions de formats.

Programme 4.2 Lectures multiples avec scanf()

```
main()
{
    int i=10;
    float l=3.14159;
    char p[50]="Bonjour";
    printf("%d bonjour %f %s",i,l,p);
    scanf("%d%f%s",&i,&l,p);
    printf("Apres lecture au clavier : %d %f %s",i,l,p);
}
```

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("%d",k) ;`	décimal
`long k ;`	`scanf("%lo",&k) ;`	`printf("%o",k) ;`	octal
`long k ;`	`scanf("%lx",&k) ;`	`printf("%x",k) ;`	hexadécimal
`unsigned long k ;`	`scanf("%lu",&k) ;`	`printf("%u",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 plus courte des deux
`double m ;`	`scanf("%lf",&m) ;`	`printf("%f",m) ;`	point décimal
`double m ;`	`scanf("%le"&m) ;`	`printf("%e",m) ;`	exponentielle
`double m ;`		`printf("%g",m) ;`	la plus 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) ;` `scanf("%s",&p[0]) ;`	`printf("%s",p) ;`	chaîne de caractères

TAB. 4.2 – Exemples de printf et scanf

Chapitre 5

Opérateurs et expressions

Le langage C est connu pour la richesse de ces opérateurs. Il apporte aussi quelques notions innovantes en matière d'opérateurs. 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 au niveau des opérateurs 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.

La sélection faite ci-après est guidée par le nombre d'opérandes mis en cause par l'opérateur et non par l'utilisation des opérateurs.

5.1 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 5.1 donne la liste des opérateurs unaires.

Nous allons prendre quelques exemples pour expliquer l'utilisation de base de ces opérateurs sur les variables décrites dans le programme 5.1.

Opérateur	Utilisation
`&`	opérateur d'adresse
`*`	opérateur d'indirection sur une adresse
`--`	opérateur de décrémentation
`++`	opérateur d'incrémentation
`sizeof`	opérateur donnant la taille en octet
`!`	NOT logique, il sert à inverser une condition
`-`	moins unaire : inverse le signe
`+`	plus unaire : sert à confirmer
`~`	complément à un

TAB. 5.1 – Liste des opérateurs unaires

Programme 5.1 Déclaration de variables et d'un pointeur

```
int var=10, *pint=&var, nvar=0;
long f=20L;
```

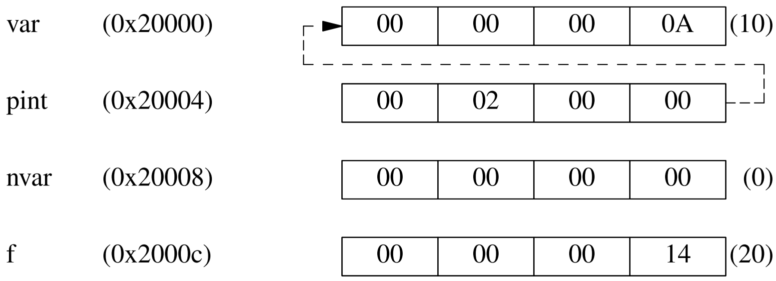

FIG. 5.1 – Exemple de relation entre pointeur et variable

5.1.1 Opérateur d'adresse

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).
`pint = &var ;`

5.1.2 Opérateur d'indirection

Le `*` est l'opérateur d'indirection. Il permet d'accéder à une variable à partir d'une adresse (souvent contenue dans un pointeur). Dans notre programme 5.1, `*&var` donne la valeur `10`, de même que `*pint`, puisque `pint` a été initialisé à partir de l'adresse de `var`.

5.1.3 Utilisation du & et de l'*

Prenons un exemple d'utilisation des opérateurs d'adresse et d'indirection en supposant que l'espace mémoire associé aux données du programme débute en `0x20000` et que la taille d'un entier est de quatre octets. Nous supposons de plus que la taille d'un pointeur est elle-aussi de quatre octets.

La figure 5.1 montre l'espace mémoire associé aux définitions du programme 5.1, et en particulier, l'association entre le pointeur `pint` et la variable `var`.

L'opération `pint = &nvar ;` est traduite dans la figure 5.2.

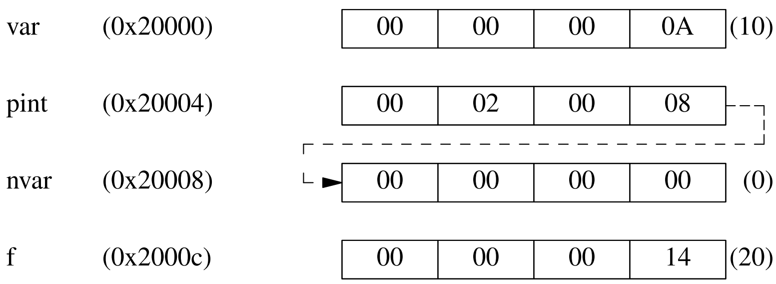

FIG. 5.2 – Modification de relation entre pointeur et variable

Une fois cette opération réalisée, l'expression `*pint = var - 2` ; se traduit au niveau du processeur[1] par les opérations suivantes :

1. mettre la valeur de `var` dans le registre R0 (R0 ← @ 0x20000, soit R0 ← 10) ;
2. soustraire 2 à cette valeur (R0 ← R0 -2, soit R0 ← 8) ;
3. mettre la valeur de `pint` dans le registre A0 (A0 ← @ 0x20004, soit A0 ← 0x20008) ;
4. mettre la valeur qui est dans R0 à l'adresse contenue dans le registre A0, c'est à dire dans var (@ A0 ← R0, soit 0x20008 ← 8).

Nous reparlerons des relations entre les pointeurs et les variables dans le chapitre 10 sur les tableaux et les pointeurs.

5.1.4 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`, ce qui implique en partant de `var` ayant une valeur de `8` que cette variable contient la valeur `7` ;
- de même, `var++` est équivalent à `var = var + 1`, ce qui implique en partant de `var` ayant une valeur de `8` que cette variable contient la valeur `9`.

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`. A la fin de cette opération `i` vaut `1` et `j` vaut `1`. En anticipant sur l'opérateur de succession 5.2.6, 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, j`.

5.1.5 Opérateur de taille

`sizeof` donne la taille en octets de la variable : `sizeof f` donne la valeur 4[2].

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).

Le type de la valeur calculée par sizeof est `size_t` comme nous l'avons vu dans la section 3.4 ce type est un type équivalent défini dans le fichier `<stddef.h>`.

[1] Cet exemple suppose un processeur simple ayant des registres internes banalisés appelés R0, R1,... pour les registres de données et A0, A1 pour les registres d'adresses. L'@ est utilisé pour signifier le contenu de la variable à l'adresse.

[2] Un entier long sur les machines les plus courantes est représenté sur 32 bits soit 4 octets.

Type d'opérateurs	Opérateurs	Usage

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

Masquage	`&` `|` `^`	et, ou, ou exclusif
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	`=`	affectation
Succession	`,`	succession

TAB. 5.2 – Liste des opérateurs binaires

5.1.6 Not Logique

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 `9`.

5.1.7 Plus et moins unaires

Le moins unaire inverse le signe de l'expression qui le suit. Le plus unaire a été introduit par la norme ; il n'existait pas dans les versions C 72. Il sert à confirmer le signe de l'expression.

5.1.8 Complément à un

L'opérateur `~` donne le complément à un de l'expression entière qui le suit. C'est un opérateur utilisé dans les calculs de masquage et nous le considérons comme un opérateur portant sur les bits de la valeur associée.

Si nous gardons l'exemple de notre variable `var` contenant la valeur `10` ; cette valeur est équivalente en binaire à :
`0000 0000 0000 0000 0000 0000 0000 1010`
et la valeur `~var` est équivalent en binaire à :
`1111 1111 1111 1111 1111 1111 1111 0101`
soit la valeur décimale `-11`. Comme son nom ne l'indique pas la somme des valeurs `var + ~var` est égale à `-1`, dont la valeur en binaire est :
`1111 1111 1111 1111 1111 1111 1111 1111`
Ceci explique pourquoi l'inversion de signe est aussi appelée complément à deux puisque `-var` est équivalent à `~var +1`.

5.2 Opérateurs binaires

Le tableau 5.2 donne la liste des opérateurs binaires.

5.2.1 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.

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`.

5.2.2 Opérateurs niveau bit

Ces opérateurs servent à construire des masques de manière à tester certains bits dans des variables de type entier (`char short int long`). Ces opérations servent à stocker des valeurs qui s'expriment dans de petits intervalles. Par exemple, il suffit de trois bits pour exprimer une valeur entre 0 et 7. Ainsi, la représentation des droits d'accès sur un fichier[3] dans le système UNIX est un exemple caractéristique d'utilisation.

Les opérations binaires sont utilisées de manière interne par les systèmes d'exploitation pour l'interfaçage des périphériques. Elles sont aussi beaucoup utilisées dans le cadre des réseaux où le contenu d'un espace mémoire est souvent dépendant d'informations représentées sur peu de bits. Elles permettent aussi de manipuler les images une fois que celles-ci ont été numérisées.

5.2.3 Opérateurs de masquage

Les trois premiers opérateurs binaires de niveau bit servent à constituer des masques ou à positionner des bits à l'intérieur de variables ou encore à constituer des valeurs ayant certains bits positionnés.

Pour reprendre l'exemple des droits d'accès sur un fichier dans le système UNIX, considérons que le type et les droits d'accès d'un fichier sont contenus dans une variable du type `unsigned short`, et voyons comment nous pouvons extraire ou fixer certains droits d'accès au fichier correspondant.

Dans l'exemple 5.2, le mode initial est fixé à `0100751` sachant que la partie haute représente le type du fichier et les modes spéciaux (bits appelés sst) et que les neuf autres bits représentent les droits d'accès (rwx respectivement pour le propriétaire du fichier, pour le groupe auquel appartient le fichier et pour les autres utilisateurs). La valeur `0100751` sert à représenter un fichier en mode : lecture écriture et exécutable pour le propriétaire ; lecture et exécutable pour le groupe ; exécutable pour les autres.

Voici l'explication des différentes lignes de l'exemple 5.2 :

- la ligne 3 met dans `res` la partie du mode qui spécifie le type de fichier, après cette instruction la variable `res` contient la valeur `0100000` ;

[3] Les droits sont exprimés sur trois bits que l'on appelle communément les bits r(ead) w(rite) et (e)x(ecute). Les droits sont exprimés pour trois populations : le propriétaire du fichier, le groupe propriétaire et les autres. Un quatrième groupe de trois bits (sst) est utilisé pour définir des comportements particuliers.

Programme 5.2 Utilisation des opérateurs de masquage

```
1.  unsigned short int mode = 0100751 ;
2.  unsigned short int res ;
3.  res = mode & 0170000 ;
4.  res = mode | 024 ;
5.  res = mode ^ 0222 ;
```

- la ligne 4 met dans `res` la valeur courante du mode en conservant le type de fichier et en forçant les bits numéro 5 et 3 à 1. Ces bits représentent respectivement le droit en écriture pour le groupe et le droit en lecture pour les autres. Ainsi, cette instruction met dans la variable `res` la valeur `0100775` ;
- la ligne 5 met dans `res` la valeur de mode en inversant les droits d'accès en mode écriture pour les trois populations d'utilisateurs. Après cette instruction, la variable `res` contient la valeur `100573`, effaçant ainsi le droit d'écriture pour le propriétaire et donnant ce droit au groupe propriétaire du fichier et aux autres utilisateurs.

5.2.4 Opérateurs binaire de décalage

Les opérateurs de décalage servent à manipuler les valeurs contenues dans les variables de type entier plutôt non signé. Ces opérateurs servent à réaliser des tests ou effacer certaines parties d'une variable.

Le décalage peut se faire vers la gauche ou vers la droite.

Le décalage vers la droite peut être signé si l'expression est signée, le résultat est cependant dépendant de l'implémentation (ce qui veut dire que pour écrire un programme portable il ne faut pas utiliser le décalage à droite sur une valeur négative).

De manière simple (en ne parlant pas des cas de débordement), un décalage vers la gauche d'une position revient à une multiplication par deux et un décalage vers la droite correspond à une division par deux.

Lorsqu'on réalise les définitions et instructions suivantes :

Programme 5.3 Utilisation des opérateurs de décalage

```
1. unsigned short int mode = 0100751 ;
2. res = mode << 4 ;
3. res = res >> 4 ;
4. res = (mode >> 12) << 12;
```

- la ligne 2 met dans `res` la partie du mode moins la partie qui spécifie le type de fichier, après cette ligne la variable `res` contient la valeur `017220` ;
- la ligne 3 met dans `res` la valeur courante du mode en ne conservant que les droits d'accès du fichier. Cette ligne met dans la variable `res` la valeur `000751` ;
- la ligne 4 met dans `res` la valeur de mode avec les douze bits correspondants aux droits mis à zéro. Cette ligne met dans la variable `res` la valeur `010000`.

5.2.5 Opérateurs binaire de relation

Les opérateurs de relation servent à réaliser des tests entre les valeurs de deux expressions. Comme nous le verrons dans le chapitre 6 ces opérateurs sont surtout utilisés à l'intérieur des instructions de contrôle de flux (tests).

Les opérateurs de relation algébrique sont au nombre de six : (`<  <=  >  >=  ==  !=`). Ces opérateurs peuvent être utilisés avec des variables de type entier ou des variables ayant une partie

arithméthique	`+=` `-=` `*=` `/=` `%=`
masquage	`&=` `|=` `^=`
décalage	`>>=` `<<=`

TAB. 5.3 – Liste des opérateurs binaires d'affectation

décimale. Notez le double égal pour le test qui est souvent source de confusion avec le simple égal qui décrit l'affectation.

Les deux autres opérateurs de tests (`&&` `||`) 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 est satisfaite. Ces opérateurs servent donc à décrire des relations logiques du type nécessité de deux conditions (et logique) ou suffisance d'une des deux conditions (condition ou). 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 correspondant 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'`.

5.2.6 Opérateur binaire 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.

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.

5.2.7 Opérateurs binaires d'affectation

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 5.3.

Le tableau 5.4 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 ;`

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.

expression	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 5 à i
`i -= j`	105	`i = i - j`	retranche j 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 ^= 4`	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

TAB. 5.4 – Exemples d'opérateurs binaires d'affectation

Operateur	Usage
ex1 ? ex2 : ex3	Retourne ex2 si ex1 est vrai retourne ex3 si ex1 est faux

TAB. 5.5 – Opérateur ternaire

Les possibilités offertes en matière d'expressions sont de ce fait peu utilisées.

5.2.8 Opérateur ternaire

Cet opérateur met en jeu trois expressions (ex1,ex2,ex3).

La première expression (ex1) est un test.

Si le test est vérifié, l'évaluation de la deuxième expression est réalisée et le résultat de cette expression est propagé comme résultat de l'expression globale.

Dans le cas contraire (le test est faux), l'évaluation de la troisième expression est réalisée et le résultat de cette expression est propagé comme résultat de l'expression globale.

Prenons pour exemple l'expression `a == b ? c : d`.

Cette expression retourne la valeur contenue dans la variable `c` si la valeur contenue dans la variable `a` est égale à celle de la variable `b`. Dans le cas contraire, l'expression retourne la valeur contenue dans la variable `d`.

L'expression `max = a >= b ? a : b` est évaluée de la manière suivante :

1. si la valeur de `a` est supérieure ou égale à la valeur de `b`, alors constituer une valeur intermédiaire avec la valeur de `a` ;
2. sinon constituer une valeur intermédiaire avec la valeur de `b` ;
3. puis stocker cette valeur intermédiaire dans la variable `max`.

<table>
<tr><th colspan="3">PRECEDENCE DES OPERATEURS</th></tr>
<tr><td>Classe d'opérateur</td><td>Opérateur(s)</td><td>Associativité</td></tr>
<tr><td>Parenthésage</td><td><code>()</code></td><td>de gauche à droite</td></tr>
<tr><td>Suffixes</td><td><code>()</code>[4] <code>[] -> .</code>
<code>++ --</code></td><td>de gauche à droite</td></tr>
<tr><td>Unaires</td><td><code>& * + - ~ !</code>
<code>sizeof sizeof()</code></td><td>de droite à gauche</td></tr>
<tr><td>Changement de type</td><td><code>(type)</code></td><td>de droite à gauche</td></tr>
<tr><td>Multiplicatifs</td><td><code>* / %</code></td><td>de gauche à droite</td></tr>
<tr><td>Additifs</td><td><code>+ -</code></td><td>de gauche à droite</td></tr>
<tr><td>Décalages</td><td><code><< >></code></td><td>de gauche à droite</td></tr>
<tr><td>Comparaisons</td><td><code>< <= > >=</code></td><td>de gauche à droite</td></tr>
<tr><td>Egalités</td><td><code>== !=</code></td><td>de gauche à droite</td></tr>
<tr><td>et bit à bit</td><td><code>&</code></td><td>de gauche à droite</td></tr>
<tr><td>ou exclusif bit à bit</td><td><code>^</code></td><td>de gauche à droite</td></tr>
<tr><td>ou bit à bit</td><td><code>|</code></td><td>de gauche à droite</td></tr>
<tr><td>et logique</td><td><code>&&</code></td><td>de gauche à droite</td></tr>
<tr><td>ou logique</td><td><code>||</code></td><td>de gauche à droite</td></tr>
<tr><td>Condition</td><td><code>?:</code></td><td>de droite à gauche</td></tr>
<tr><td>Affectations
Affectations
Affectations</td><td><code>= += -= *= /= %=</code>
<code>&= |= ^=</code>
<code><<= >>=</code></td><td>de droite à gauche</td></tr>
<tr><td>Succession</td><td><code>,</code></td><td>de gauche à droite</td></tr>
</table>

TAB. 5.6 – Précédence des opérateurs

5.2.9 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.

REGLE de précédence :
La priorité des opérateurs du langage C est décroissante de haut en bas selon le tableau 5.6.

Lorsque deux opérateurs se trouvent dans la même ligne du tableau 5.6, la priorité d'évaluation d'une ligne de C dans laquelle se trouvent ces opérateurs, est donnée par la colonne de droite (associativité).

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 (selon le bon sens commun, méfiez vous des relations opérateurs de test et opérateurs bit à bit) ;
- opérateur ternaire (associativité de droite à gauche) ;

Une EXPRESSION est : une suite syntaxiquement correcte d'opérateurs et d'opérandes.

TAB. 5.7 – Définition d'une expression

Une expression ramène toujours une valeur, même si la valeur n'est pas utilisée. Une expression est fausse si son résultat est nul. Elle est vraie si son résultat est non nul.

TAB. 5.8 – Expression vraie ou fausse

– opérateurs binaires d'affectation (associativité de droite à gauche) ;
– succession.

5.3 Expressions

Cette suite doit avoir un nombre d'opérandes cohérent avec le nombre et le type de ses opérateurs.

Prenons la variable `i` qui est définie `int i=10;` l'expression `i` donne la valeur `10` .

5.4 Instructions

Maintenant que nous connaissons la notion d'expression nous pouvons introduire celle d'**instruction**. Une **instruction** est :
– soit une instruction simple,
– soit un bloc.

Une **instruction simple** est :
– soit une instruction de contrôle,
– soit une expression suivie de " ;".

Une instruction simple est toujours terminée par un " ;".

Un **bloc** a la structure suivante :
– une accolade ouvante "{"
– une liste de définitions locales au bloc (optionnelle)
– une suite d'instructions
– une accolade fermante "}".

Chapitre 6

Instructions de contrôle

A partir de maintenant, lorsque nous employons le terme **instruction**, il peut prendre de manière indifférente l'une des trois significations suivantes :

1. instruction simple ;
2. ou instruction de contrôle de flot ;
3. ou bloc d'instructions.

6.1 Instructions conditionnelles

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

6.1.1 Test

L'opérateur de test se présente sous les deux formes présentées dans la table 6.1.

L'expression n'est pas forcément un test qui retourne la valeur 0 ou +1. Ce peut être un calcul ou une affectation, car il n'y a pas de type booléen.

Comme nous l'avons déjà vu : une expression est vraie si elle ramène un résultat non nul ; elle est fausse si le résultat est nul.

Voici quelques exemples de tests en langage C :

if(expression) instruction
if(expression) instruction1 else instruction2

TAB. 6.1 – Syntaxes du if

switch	(expression)	{
	case value1 :	inst 10
		inst 11
		inst 12
	case value2 :	inst 20
		inst 21
		inst 22
	etc.	
	case valueN :	inst N0
		inst N1
		inst N2
	default :	inst 30
		inst 31
		inst 32
	}	

TAB. 6.2 – Syntaxe du switch

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

Les tests sont souvent imbriqués :

```
if ( i == 1 ) inst1
else if ( i == 2 ) inst2
else if ( i == 3 ) inst3
else if ( i == 4 ) inst4
else if ( i == 5 ) inst5
etc....
```

6.1.2 Table de branchement

Pour éviter les imbrications d'instructions `if`, le C possède une instruction qui crée une table de branchement : c'est l'instruction **switch**.

La syntaxe du **switch** est résumée dans la table 6.2.

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 intruction **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 par prédéfini 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.

Les programmes 6.1, 6.2 et 6.3 sont des exemples d'utilisation de table de sauts.

Programme 6.1 Exemple de table de sauts

```
switch(exp){
    case 21:
        inst_10
        inst_11
        inst_12
    case 12:
        inst_20
        inst_21
        inst_22
        break;
    default:
        inst_30
        inst_31
        inst_32
        break;
}
```

Le premier exemple 6.1 est symbolique :

1. lorsque l'expression `exp` donne une valeur égale à 21, les instructions numérotées 10, 11, 12, 20, 21 et 22 sont exécutées ;
2. lorsque l'expression `exp` donne une valeur égale à 12, les instructions 20, 21 et 22 sont exécutées ;
3. dans tous les autres cas, les instructions 30, 31 et 32 sont exécutées.

Programme 6.2 Exemple de table de sauts

```
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;
    default:printf("ceci n est pas un nombre\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;
    }
```

Dans l'exemple 6.2 :

1. lorsque i est égal à 6 ou 8, le programme écrit "`le nombre est superieur 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 superieur a 5`”, puis écrit “`le nombre est impair`” ;
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`.

Programme 6.3 Exemple de table de sauts

```
switch(c = getchar()){
    case '0': case '1':
    case '2': case '3':
    case '4': case '5':
    case '6': case '7':
    case '8': case '9':
                        nb_chiffre++;
                        break;
    case 'a': case 'b':
    case 'c': case 'd':
    case 'e': case 'f':
    case 'g': case 'h':
    case 'i': case 'j':    /* toutes les valeurs des */
    case 'k': case 'l':    /* lettres doivent etre */
    case 'm': case 'n':    /* presentes dans les case */
    case 'o': case 'p':
    case 'q': case 'r':
    case 's': case 't':
    case 'u': case 'v':
    case 'w': case 'x':
    case 'y': case 'z':
                        nb_lettre++;
                        break;
    default :
                        nb_others++;
                        break;
}
```

Dans l'exemple 6.3, la sélection incrémente des variables `nb_chiffres`, `nb_lettres`, ou `nb_others`, selon que le caractère lu au clavier est un chiffre, ou une lettre ou aucun des deux.

6.2 Instructions répétitives

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

6.2.1 while

La syntaxe du **while** est décrite dans la table 6.3.

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 6.1.

Dans l'exemple 6.4 la fonction `getchar()` est une fonction qui lit un seul caractère sur le clavier. Nous supposerons que la valeur EOF est connue et correspond à ce que retourne la fonction `getchar()` lorsque la fin de fichier est détectée sur le clavier (par la frappe du caractère

while(expression) instruction

TAB. 6.3 – Syntaxe du while

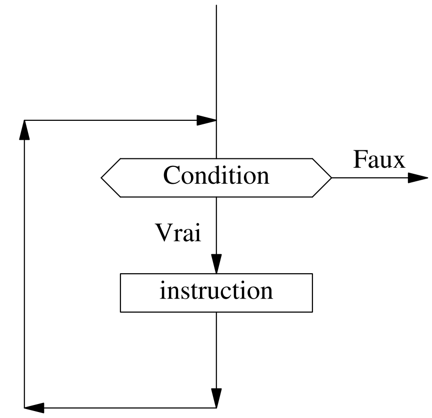

FIG. 6.1 – Organigramme du while

Controle D sur un système de type UNIX par exemple). Nous reparlerons de manière approfondie de `getchar()` dans la section 16.1.1.

Programme 6.4 Lecture d'une ligne avec while

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

Dans cet exemple 6.4, les caractères sont lus un par un au clavier et sont rangés dans les cases du tableau `tab`. Cet exemple n'est pas sécurisé en ce sens qu'il n'y a pas de vérification de débordement du tableau. Nous verrons (Prog. 6.6) que cet exemple peut-être réécrit à l'aide d'une boucle **for** de manière à ne pas déborder du tableau.

L'exemple 6.5 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).

6.2.2 for

La syntaxe du **for** est donnée dans la table 6.4.

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.

Programme 6.5 Recopie d'une chaîne avec une boucle while()

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

for(exp1 ; exp2 ; exp3) instruction

TAB. 6.4 – Syntaxe du for

Le **for** est équivalent à un **while** plus une instruction d'initialisation et une instruction de rebouclage comme il est montré dans le tableau 6.5.

Le **for** correspond à l'organigramme 6.2.

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++)`
2. `for (i = 0 , j = n ; i < j ; i++ , j--)`
3. `for ( ; ; ) inst`

Le premier 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++`).

Le deuxième exemple montre comment avoir plusieurs initialisations et plusieurs expressions dans l'instruction de rebouclage, en utilisant l'opérateur de succession "`,`".

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 plutôt un bloc d'instructions dans lequel se trouvent des conditions de sortie de la boucle (voir section 6.3 sur les ruptures de séquence).

6.2.3 do while

La syntaxe du **do while** est donnée dans la table 6.6.

A l'inverse du **while**, le **do while** place son test en fin d'exécution, d'où au moins une exécution. Il ressemble aux REPEAT UNTIL d'ALGOL ou de PASCAL. L'organigramme correspond à celui

`for(exp1 ; exp2 ; exp3) inst`	est équivalent à	`exp1 ;`
		`while(exp2) {`
		`inst`
		`exp3 ;`
		`}`

TAB. 6.5 – Comparaison du for et du while

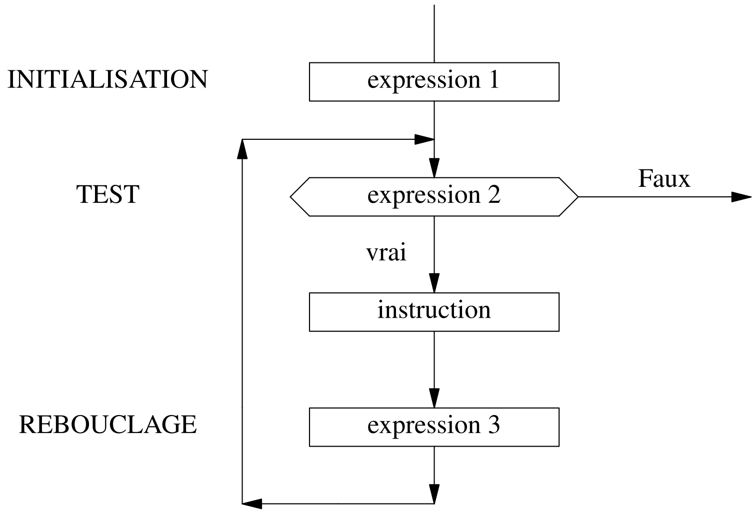

FIG. 6.2 – Organigramme du for

Programme 6.6 Lecture d'une ligne avec for

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

do instruction while(expression) ;

TAB. 6.6 – Syntaxe du do while

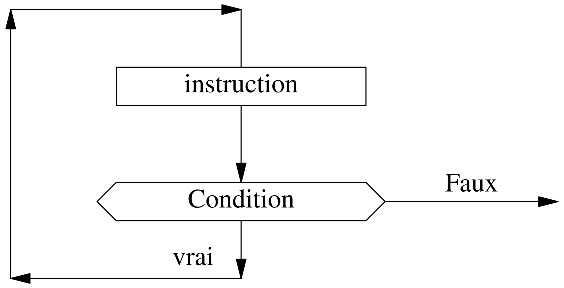

FIG. 6.3 – Organigramme du do while

du **while** mais inversé.

Programme 6.7 Décomposition des puissances de dix d'un nombre avec un do while()

```
int s[10], i=0, n=5634  ;
do
    s[i++] = n % 10 ;
while( (n /= 10) > 10 ) ;
```

Dans l'exemple donné dans le programme 6.7 les variables `i`, `n` et le tableau d'entiers `s` sont modifiés de la manière suivante :

1. `s[0]` reçoit 4 et `i` reçoit 1 puis `n` reçoit `563` ; `n` étant alors supérieur à 10 la boucle continue ;
2. `s[1]` reçoit 3 et `i` reçoit 2 puis `n` reçoit `56` ; `n` étant alors supérieur à 10 la boucle continue ;
3. `s[2]` reçoit 6 et `i` reçoit 3 puis `n` reçoit `5` ; `n` étant alors inférieur à 10 la boucle se termine.

Lorsque la boucle est terminée, le tableau d'entier contient les valeurs 4, 3, et 6 respectivement dans les éléments de rangs 0, 1 et 2.

6.2.4 Remarques sur les répétitives

Comme nous l'avons vu dans la section 2.4, une instruction peut être une instruction simple ou un bloc.

Les différentes boucles peuvent être imbriquées et, comme nous le verrons dans le chapitre 7 sur la programmation structurée, il faut s'astreindre à une certaine mise en page pour faciliter la lecture du programme.

6.3 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 cinq genres qui correspondent à leur niveau de travail.

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

Deux appels de fonctions de la bibliothèque permettent aussi de modifier l'ordre d'exécution d'un programme. Il s'agit de l'appel aux fonctions `void exit(int status)` qui termine l'exécution du programme et `void longjmp(jmp_buf env, int val)` qui permet de sauter à un point de reprise mis dans le programme.

6.3.1 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.

Programme 6.8 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++ ;
}
```

Prennons l'exemple 6.8, 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. A 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. et i-j contient une valeur qui correspond au nombre de caractères blancs.

6.3.2 break

Nous avons déjà vu une utilisation du **break** dans le **switch**. Plus généralement, il permet de sortir de la boucle d'itération. Il n'est pas aussi général qu'en ADA puisqu'il ne permet pas de sortir d'un bloc quelconque et surtout il ne peut pas être labellé, c'est-à-dire qu'il ne peut sortir que d'un niveau d'accolade.

Dans l'exemple 6.9, nous reprenons l'exemple 6.8, de manière à créer une boucle qui compte le nombre de caractères jusqu'au retour chariot en utilisant des instructions `break`. Lorsque le retour chariot ('\r') est rencontré, le **break** provoque la sortie de la boucle **for**. Il en serait de même avec un **while** ou un **do/while**.

Programme 6.9 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 6.10, nous reprenons l'exemple 6.4, de manière à créer une boucle qui remplit

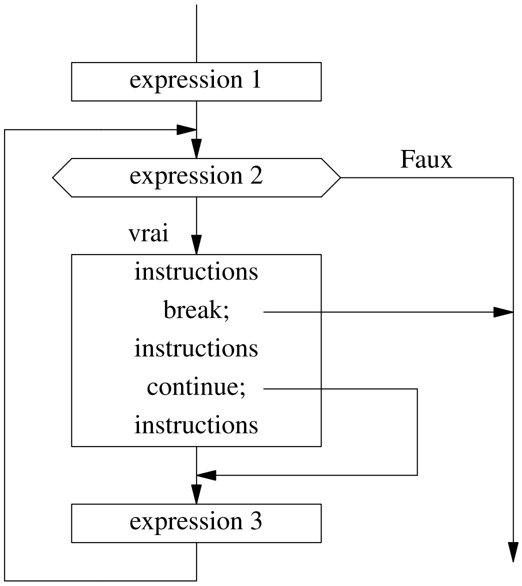

FIG. 6.4 – break et continue dans un for

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.

Programme 6.10 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 6.4, 6.5 et 6.6 sont des organigrammes qui montre les effets du **break** et du **continue** sur les boucles **for**, **while**, et **do while**.

6.3.3 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 appellée label. Un label est une chaîne de caractères suivie de " : " .

Exemple :

6.3.4 return

L'instruction `return` provoque la terminaison de l'exécution de la fonction dans laquelle elle se trouve et le retour à la fonction appelante. 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.

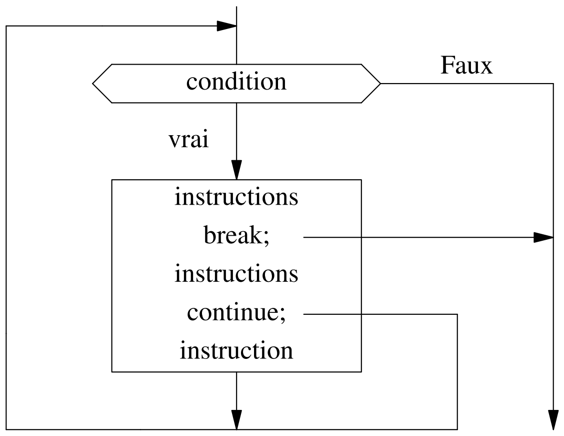

FIG. 6.5 – break et continue dans un while

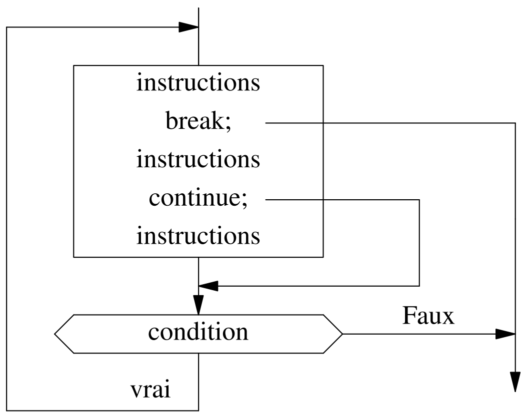

FIG. 6.6 – break et continue dans un do while

Programme 6.11 Utilisation de l'infâme goto

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

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

Cette expression est évaluée et la valeur calculée est retournée à la fonction appelante. 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. 6.12) 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` et `-2` dans tous les autres cas.

Programme 6.12 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 7

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.

7.1 Historique

L'historique de ce mouvement est à peu près le suivant :

- 1965 Dijkstra [Dij65] propose la suppression de GOTO dans les langages. Ceci est associé avec les premiers développements d'ALGOL.
- 1968 Dijkstra [Dij68] propose les 3 structures fondamentales en programmation.
- 1971 Wirth [Wir71] le père du langage Pascal écrit un article sur la programmation par raffinements successifs.
- 1974 Wirth [Wir74] explicite la manière d'écrire un programme bien structuré.

De nombreux autres articles ont été écrits sur ce sujet.

7.2 Idées fondamentales

La première règle en programmation structurée est de minimiser le nombre de GOTO dans le programme. Cependant la programmation structurée ne se caractérise pas par l'absence de GOTO, mais par la présence de structures dans le programme. Les idées de base sont les suivantes :

- La programmation peut se faire en utilisant un nombre réduit de structures de programmes.
- La présentation du programme doit refléter la structure du programme.
- Les tâches doivent être éclatées en sous-tâches (raffinement).
- Le raffinement doit être fait en tenant compte des données manipulées par les tâches. En particulier, les données manipulées par une tâche doivent être déclarées près de la tâche.
- Chaque sous-tâche doit avoir un point d'entrée unique et un point de sortie unique. Il y eut une longue polémique sur la définition d'un point de sortie unique.

Les structures fondamentales sont au nombre de quatre[1] :

1. **séquence** : les instructions sont exécutées de manière séquentielle.
2. **sélection** : qui représente un choix dans le code à exécuter. La sélection peut se présenter sous trois formes :
   - **if else** : représentant une alternative.

[1] Dijkstra a démontré que toute structure de programme peut en fait se ramener aux trois premières.

   - **if orif** : permettant de décrire un choix multiple.
   - **case** : destiné à créer des tables de vérité.
3. **itération** : pour répéter un traitement tant qu'une condition est vérifiée. Deux types de boucles ont été choisies :
   - la boucle **while** : qui place le test avant le traitement.
   - la boucle **until** : qui place le test après le traitement.
4. **sortie** est destinée à terminer une sous-tâche. Deux types de sortie sont possibles :
   - **escape** termine la sous-tâche.
   - **cycle** permet de sauter les instructions suivantes d'une boucle et de passer au cycle suivant de cette boucle.

Il est possible de distinguer trois niveaux dans la structuration d'un programme :
- La sémantique propre au langage et ses restrictions.
- Le respect des recommandations au niveau de l'utilisation des structures de programme.
- La présentation du texte du programme.

7.3 Langage C et Programmation Structurée

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 est peu contraignant au niveau de la compilation et il offre peu de restrictions sémantiques.

7.3.1 Ambitions du langage C

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. Ceci va à l'encontre de la programmation structurée telle que Wirth l'a décrit. En Pascal, le programmeur travaille sur une machine virtuelle, la machine Pascal. 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.

7.3.2 C et Structures Fondamentales

Le tableau 7.1 montre que le C propose des structures de programmation équivalentes à celles recommandées dans la programmation structurée.

7.4 Quelques exemples

Nous allons écrire à chaque fois trois exemples de code permettant de réaliser la même tâche :
- Le premier exemple est écrit de manière la moins structurée possible.
- Le deuxième exemple est écrit en respectant les structures mais sans mise en page.
- Le dernier exemple est écrit en respectant les structures et avec une mise en page qui tente de refléter ces structures.

Type	Structures	Instruction C correspondante
sélection	if else	if (expression) instruction else instruction
	if orif	n'existe pas.
	case	switch.
itération	while	while(expression) instruction
	until	do instruction while(expression) ;
		for(expression ;expression ;expression) instruction
sortie	escape	break ou return
	cycle	continue

TAB. 7.1 – C et Structures Fondamentales

7.4.1 Conditions

La tâche a pour but de déterminer quel entier `i` ou `j` a la plus grande valeur absolue.

Le programme 7.1 est écrit de manière la moins structurée possible. La difficulté lors de la

Programme 7.1 Ensemble de tests non structuré

```
    if( i == j) goto T3;
    if( i <= 0 ) goto L1;
    if( j <= 0 ) goto L2;
    if( i > j) goto T2;
    goto T1;
L2: if( i > -j ) goto T2;
    goto T1;
L1: if( j <= 0 ) goto L3;
    if( -i > j) goto T2;
    goto T1;
L3: if( -i > -j ) goto T2;
T1: printf(" le plus grand est j : %d\n",j);
    goto T4;
T2: printf(" le plus grand est i : %d\n",i);
    goto T4;
T3: printf("  i et j sont egaux\n");
T4:
```

lecture de ce programme 7.1 est de trouver le chemin qui arrive sur les étiquettes. Pour la recherche d'erreur dans un code de ce type, il faut déterminer comment le programme est arrivé sur une ligne.

Le programme 7.2 est écrit de manière structurée mais sans mise en page. Dans ce cas, il est difficile de savoir quelles conditions ont mené à une ligne. En particulier il est délicat de trouver la condition correspondant à un `else`.

Le programme 7.3 est écrit de manière structurée avec mise en page. La mise en page fait apparaitre de manière plus évidente l'imbrication des `if else`.

Une autre façon claire d'écrire du code avec des `if else` imbriqués est de mettre des accolades et des commentaires pour identifier les accolades fermantes. Cette technique est illustrée par le programme 7.4.

Certains programmeurs annotent les accolades ouvrantes et fermantes avec des labels identifiant le type de la structure et le numéro de cette structure en commentaires. Cette technique

Programme 7.2 Ensemble de tests structuré non mis en page

```
if( i == j ) printf("  i et j sont egaux\n");
else
if( i < 0 )
if( j < 0 )
if( -i > -j ) printf(" le plus grand est i : %d\n",i );
else          printf(" le plus grand est j : %d\n",j );
else
if( -i > j )  printf(" le plus grand est i : %d\n",i );
else          printf(" le plus grand est j : %d\n",j );
else
if( j < 0 )
if( i > -j )  printf(" le plus grand est i : %d\n",i );
else          printf(" le plus grand est j : %d\n",j );
else
if( i > j )   printf(" le plus grand est i : %d\n",j );
else          printf(" le plus grand est j : %d\n",j );
```

Programme 7.3 Ensemble de tests structuré et mis en page

```
if( i == j ) printf("  i et j sont egaux\n");
else
  if( i < 0 )
    if( j < 0 )
      if( -i > -j ) printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
    else
      if( -i > j )  printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
  else
    if( j < 0 )
      if( i > -j )  printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
    else
      if( i > j )   printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
```

Programme 7.4 Ensemble de tests structuré et mis en page avec commentaires

```
if( i == j ) printf("  i et j sont egaux\n");
else {
  if( i < 0 ) {
    if( j < 0 )
      if( -i > -j ) printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
    else
      if( -i > j )  printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
  } /* fin i < 0 */
  else {
    if( j < 0 ) {
      if( i > -j )  printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
    } /* fin j < 0 */
    else {
      if( i > j )   printf("le plus grand est i : %d\n",i );
      else          printf("le plus grand est j : %d\n",j );
    } /* fin j > 0 */
  } /* fin i > 0 */
} /* fin i != j */
```

est illustrée dans le programme 7.5. L'avantage de ces solutions apparaît lorsque la condition s'étend sur plusieurs pages d'un terminal. Cela évite de remonter dans le source en cherchant le `if` correspondant à un `else`.

7.4.2 Table de branchement

Il est possible de réaliser la même tâche en utilisant une table de branchement. Dans l'exemple 7.6, le calcul donnant la condition doit être commenté car les différentes valeurs possibles ne sont pas évidentes à trouver.

7.4.3 Boucles

Nous allons réaliser un parcours de tableau de caractères (tabc) avec arrêt lors de la rencontre du caractère `'\0'`.

```
int i;
char tabc[100];
```

Le programme 7.7 réalise ce parcours de tableau, il est écrit de manière non structurée.

Le programme 7.8 réalise ce parcours de tableau, il est écrit de manière structurée.

Le programme 7.9 réalise ce parcours de tableau, il est écrit de manière structurée avec mise en page.

Programme 7.5 Ensemble de tests structuré, mis en page et étiqueté

```
if( i == j ) printf("  i et j sont egaux\n");
else {  /*  ELIF 1 */
  if( i < 0 ) { /* ELIF1.IF1 */
    if( j < 0 )
      if( -i > -j ) printf(" le plus grand est i : %d\n",i );
      else          printf(" le plus grand est j : %d\n",j );
    else
      if( -i > j )  printf(" le plus grand est i : %d\n",i );
      else          printf(" le plus grand est j : %d\n",j );
  } /* fin ELIF 1.IF1 */
  else { /* ELIF 1.ELIF1 */
    if( j < 0 )  {
      if( i > -j )  printf(" le plus grand est i : %d\n",i );
      else          printf(" le plus grand est j : %d\n",j );
    } /* fin j < 0 */
    else {
      if( i > j )   printf(" le plus grand est i : %d\n",i );
      else          printf(" le plus grand est j : %d\n",j );
    } /* fin j > 0 */
  } /* fin ELIF1.ELIF1 */
} /* fin ELIF 1 */
```

Programme 7.6 Ensemble de tests par table de branchement

```
if( i == j ) printf("  i et j sont egaux\n");
else {
 /* l'expression (i<0) + 2*(j<0)) donne la valeur : */
    /*      0  si i>0 et j>0     */
    /*      1  si i<0 et j>0     */
    /*      2  si i>0 et j<0     */
    /*      3  si i<0 et j<0     */
    switch( (i<0) + 2*(j<0)) {
    case 0 :
        if( i > j ) printf("le plus grand est i : %d\n",i );
        else        printf("le plus grand est j : %d\n",j );
        break;
    case 1 :
        if( -i > j )  printf("le plus grand est i : %d\n",i );
        else          printf("le plus grand est j : %d\n",j );
        break;
    case 2 :
        if( i > -j )  printf("le plus grand est i : %d\n",i );
        else          printf("le plus grand est j : %d\n",j );
        break;
    case 3 :
        if( -i > -j ) printf("le plus grand est i : %d\n",i );
        else          printf("le plus grand est j : %d\n",j );

    } /* switch */
} /* fin i != j */
```

Programme 7.7 Boucle réalisée par des tests et sauts

```
i = 0;
B1:
if( tabc[i] == '\0' ) goto B2;
i = i + 1;
if( i < 100 ) goto B1;
B2:
```

Programme 7.8 Boucle réalisée par un for()

```
for(i = 0 ; i<100 ; i++) if( tabc[i] == '\0' )  break;
```

Programme 7.9 Boucle réalisée par un for() et mise en page

```
for(i = 0 ; i<100 ; i++)
    if( tabc[i] == '\0' )  break;
```

Chapitre 8

Fonctions

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 quasi 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 de variables différents.

8.1 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, voir figure 8.1 contient :

- une interface ;
- un corps de fonction.

L'interface complète d'une fonction contient :

- la déclaration du type et du nom de la fonction ;

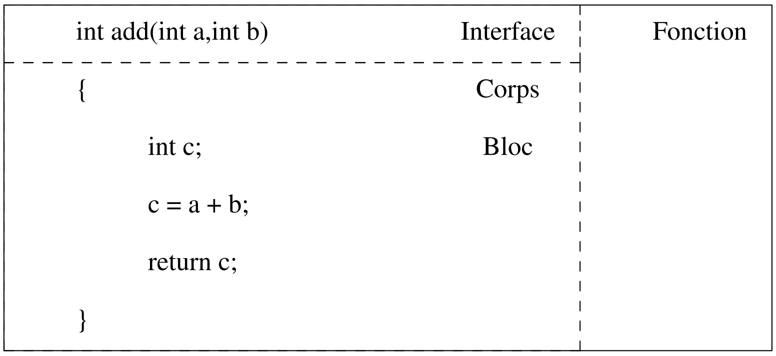

Fig. 8.1 – Structure d'une fonction

Code C	Explications
int plus(int a,int b) { a = a + b ; return a ; }	fonction plus qui retourne un résultat de type entier et qui a deux arguments a et b de type entier.

void add(int a,int b,int *c)
 	
{
*c = a + b ;
}
	fonction add qui ne retourne pas de résultat
	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 le dernier paramètre

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 entier

TAB. 8.1 – Exemples de définition de fonctions

Définition	Appel

int plus(int a,int b)
{
a = a + b ;
return(a) ;
}
	int x=4,y=6,z ;
	
	z = plus(1,23) ;
	z = plus(x,y) ;

TAB. 8.2 – Exemples d'appels de fonctions

- une parenthèse ouvrante ;
- la déclaration des types et des noms des paramètres ;
- 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 8.1 donne des exemples de définition de fonctions en C.

8.2 Passage des paramètres

En langage C, **les passages de paramètres se font par valeur**, c'est à dire que la fonction appelante 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 (voir chap.). Cette variable est accessible de manière interne par la fonction à partir de l'argument formel correspondant.

Le tableau 8.2 donne deux exemples d'appels à la fonction `plus()` définie dans le tableau 8.1.

8.2.1 Passage de constantes

La figure 8.2 donne des indications sur une vision de l'espace mémoire pendant l'exécution des étapes de l'appel de de la fonction correspondant à la ligne `z = plus(1,23) ;` .

Etape 1 les valeurs 1 et 23 sont empilées dans des variables anonymes dont les types correspondent aux types de paramètres de la fonction dans ce cas deux entiers ;

Etape 2 la fonction `plus()` est appelée, ce qui se traduit par l'empilement de l'adresse de retour et du pointeur de contexte courant. La fonction `plus()` dispose de deux variables qu'elle connait sous les noms "a" et "b". Ces deux variables sont les copies des valeurs dont nous avons parlé au point précédent. Elle ont donc pour valeurs initiales respectivement 1 et 23 ;

Etape 3 la fonction `plus()` modifie la valeur de la variable "a" en y mettant la valeur 24 ;

Etape 4 la fonction se termine en retournant cette valeur 24, cette valeur est stockée dans un registre du processeur réservé à cet effet ;

Etape 5 les variables créées par l'appel sont détruites (l'espace sur la pile est restitué, le pointeur de pile est remis à sa valeur initiale).

Etape 6 la valeur stockée dans le registre du processeur est affectée à la variable z (correspondant à l'affectation).

La vision donnée par les shémas de la figure 8.2, de même que les autres dessins de ce chapitre, est simpliste en ce sens qu'elle considére que les cases mémoire sur la pile font toutes la même taille et permettent de stocker des adresses ou des entiers. Elle devrait être raffinée pour tenir compte de la taille des types mais alors elle deviendrait spécifique à une architecture. De même, elle considére que le sommet de pile est au dessus du bas de pile, et que la taille de la pile est suffisante pour mettre tout nouvel élément.

8.2.2 Passage de variables

Le deuxième appel à la fonction `z = plus(x,y) ;` provoque les opérations suivantes avec un espace mémoire représenté dans la figure 8.3 :

Etape 1 les valeurs des variables `x` et `y` (4 et 6) sont calculées, ces valeurs sont empilées dans des variables dont les types correspondent aux types de paramètres de la fonction ;

Etape 2 la fonction `plus()` est appelée et elle dispose de deux variables qu'elle nomme "a" et "b". Ces deux variables sont les copies des valeurs de `x` et `y` dont nous avons parlé au point précédent. Ces deux variables ont donc pour valeurs initiales 4 et 6 ;

Etape 3 la fonction `plus()` modifie la valeur de la variable "a" en y mettant la valeur 10 ;

Etape 4 la fonction se termine en retournant cette valeur 10 ;

Etape 5 les variables créées par l'appel sont détruites.

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. Par exemple, lorsque lors du deuxième appel, la fonction `plus()` modifie la variable "a" cela ne modifie pas la variable "x".

8.3 Utilisation de pointeurs en paramètres

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

[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.

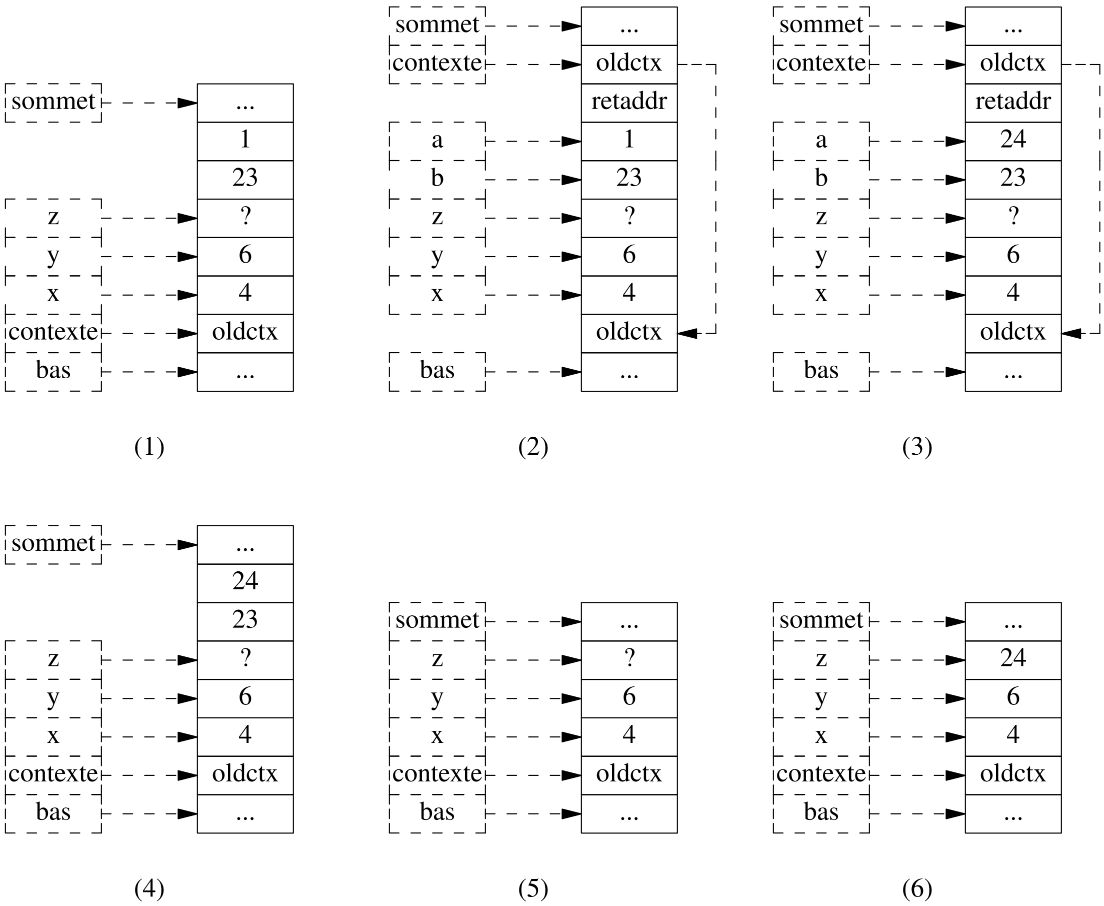

FIG. 8.2 – Pile et passage de constantes

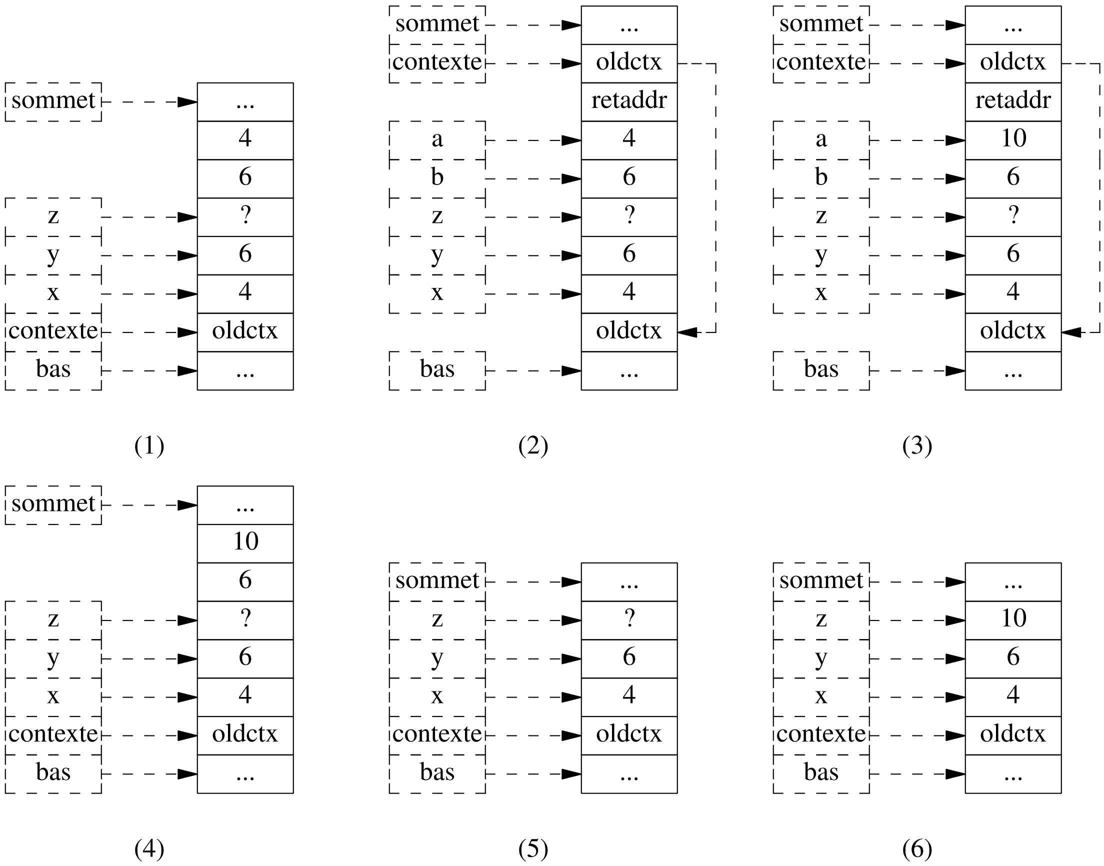

FIG. 8.3 – Pile et passage de variables

Déclaration	Appel

`add(int a,int b,int *c)`
`{`
`   *c=a+b;`
`}`
	`int x=5,y=7,z;`
	
	`add(x,y,&z);`
	
	`add(43,4,&x);`

TAB. 8.3 – Pointeurs et appels de fonctions

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

TAB. 8.4 – Conversions de type unaire

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.

Le tableau 8.3 est un exemple d'utilisation de la fonction `add()` qui accepte comme arguments deux entiers et une adresse d'entier. Cette fonction utilise le troisième argument pour communiquer le résultat de ses calculs.

Comme le montre la figure 8.4, lors du premier appel `add(x,y,&z);` la variable `z` est modifiée et elle prend la valeur 12 (5+7).

Lors du deuxième appel `add(43,4,&x)`; la variable `x` est modifiée et elle prend la valeur 47 (43+4).

8.4 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 3.8. 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 8.4.

8.5 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` dont la syntaxe est décrite dans la table 8.5.

Dans le corps de la fonction, le résultat est généré par le **`return`**. L'expression qui suit le **`return`** est évaluée, et la valeur obtenue est retournée. Au niveau de la fonction appelante, le retour de fonction peut être utilisé comme la valeur d'une expression. Si nous prenons le cas de la fonction

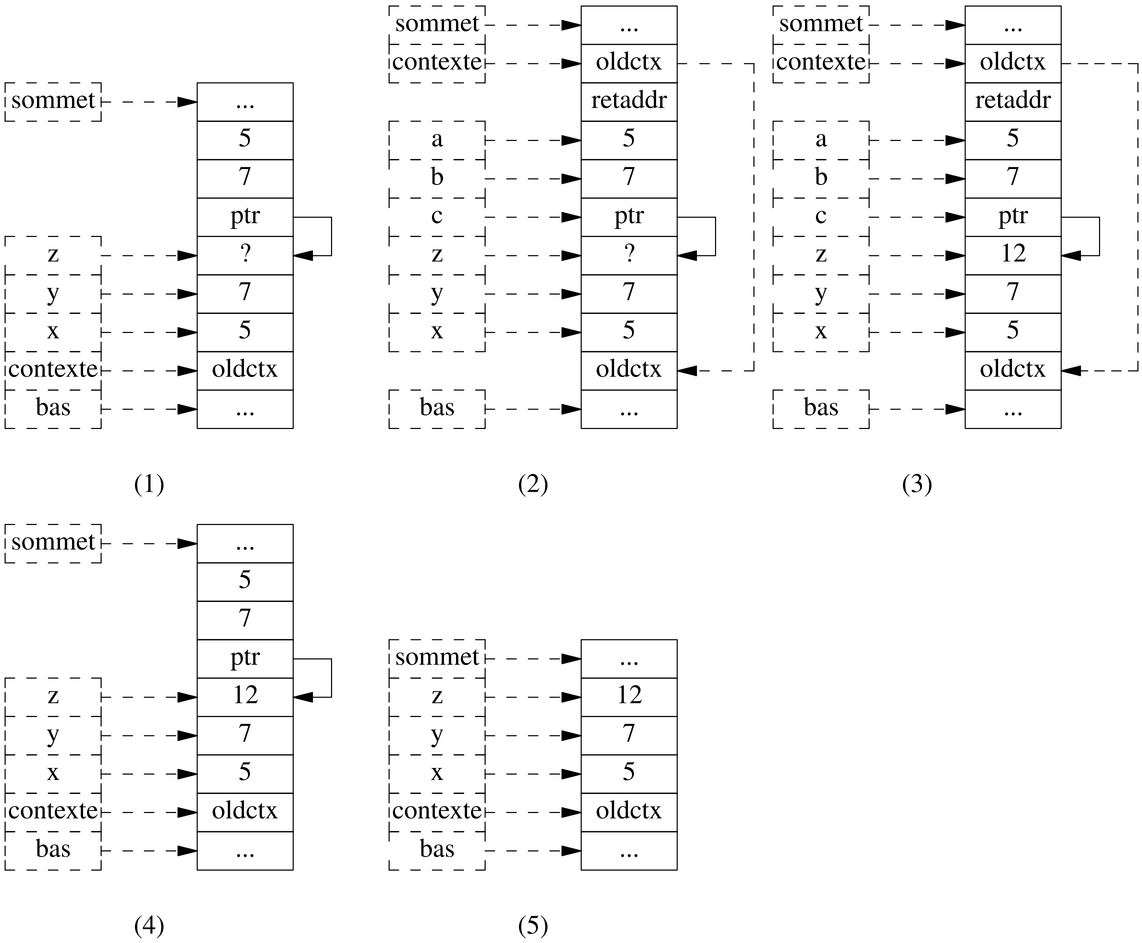

FIG. 8.4 – Pile et passage de variables avec référence

return expression ;

TAB. 8.5 – Syntaxe du return

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 5. Ce qui nous permet d'écrire par exemple :

```
z = z * plus(x,y) ;
```

8.6 Récursivité

En C, toute fonction peut appeler toute fonction dont elle connaît le nom (nous reviendrons sur ces problèmes dans le chapitre 9 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 :

```
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; }
```

8.7 Paramètres de la fonction main()

La structure des arguments de la fonction `main()` reflète la liaison entre le langage C et le système d'exploitation en particulier le système UNIX. Dans un système de type UNIX, les paramètres de la fonction `main()` sont passés par le **shell** dans la majorité des cas. Ils sont passés par une des fonctions du type `exec(3)` entre applications. Ces paramètres ont une structure prédéfinie. 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 le 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 contient le nombre d'arguments qui ont été passés lors de l'appel du binaire exécutable (nombre de mots dans la ligne de commande) ;
- 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 du binaire exécutable appelé ;
- le nombre de pointeurs valides dans le premier tableau est donné par le contenu de la variable entière (nombre d'arguments `argc`) ;

- le deuxième tableau contient les variables d'environnement du `shell` au moment de l'appel du fichier 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 donnée par un marqueur. Ce marqueur est un pointeur NULL, c'est-à-dire, un pointeur qui contient l'adresse 0. Dans ce cas, cette adresse est du type (`char *`) ou bien encore adresse d'un caractère ;
- la chaîne pointée par la première entrée du tableau `argv` contient le nom de la commande elle-même.

La figure 8.5 donne la vision interne des paramètres issus de la commande :
`echo essai de passage de parametres`

8.8 Pointeur de fonction

En langage C, le nom d'une fonction est considéré comme une adresse de manière identique au nom d'un tableau. Le nom d'une fonction correspond à l'adresse de la première instruction à l'intérieur du code exécutable, une fois l'édition de liens réalisée.

8.9 Etapes d'un appel de fonction

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

Mise en pile des paramètres la fonction appelante 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.

Si les paramètres sont des variables, la fonction appelante empile une copie constituée à partir de la valeur courante de la variable. Ceci explique pourquoi la valeur d'un argument formel peut être différentes de celle de son argument réel.

Saut à la fonction appelée La fonction appelante 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 concervée 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 appelante au niveau des registres du processeur (sauf les registres scratch qui contiennent la valeur de retour de la fonction). Pour celà, 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

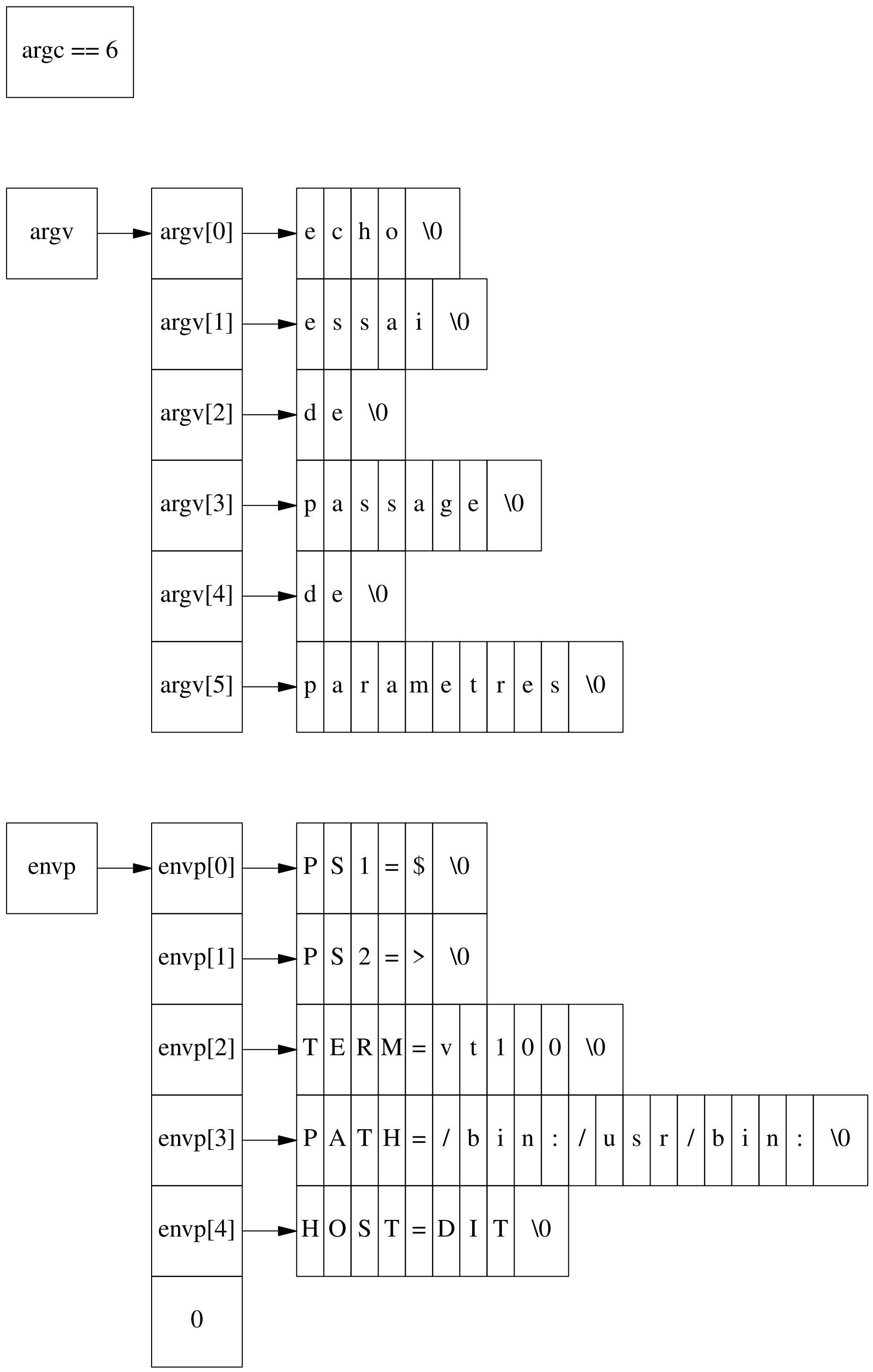

FIG. 8.5 – Paramètres de main()

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 appelante.

Récupération du résultat et effacement des paramètres la fonction appelante dépile les paramètres qu'elle avait empilé 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 9

Compilations séparées

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.

A 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.

9.1 Programme

Comme nous l'avons déjà dit 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 9.1.

9.2 Fichier source

Comme le montre shématiquement la figure 9.2, 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,
- 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.

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

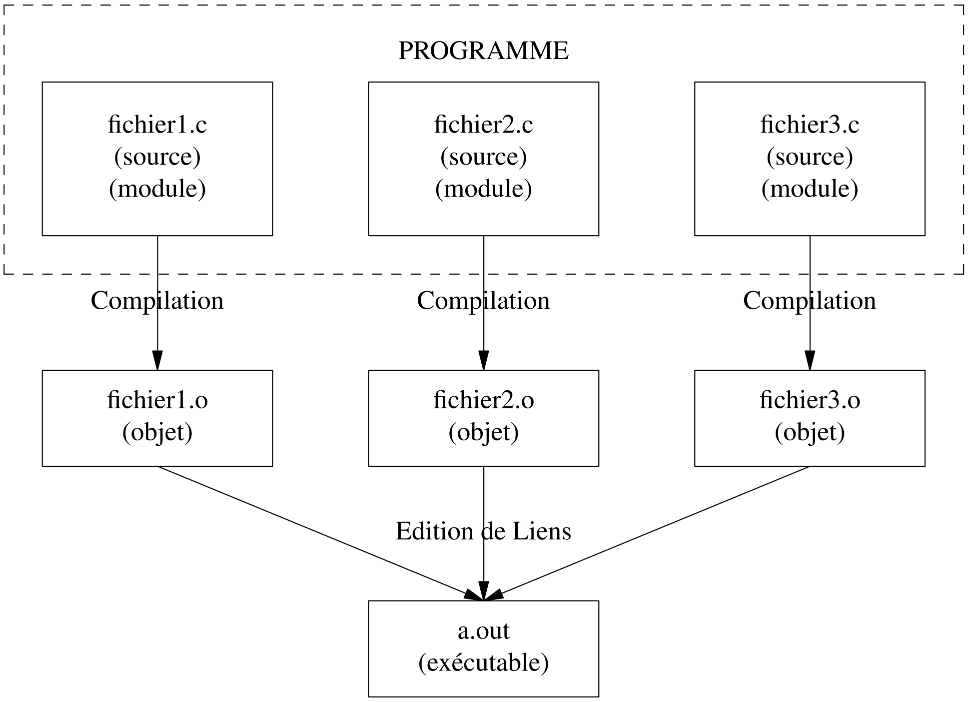

FIG. 9.1 – Du source à l'exécutable

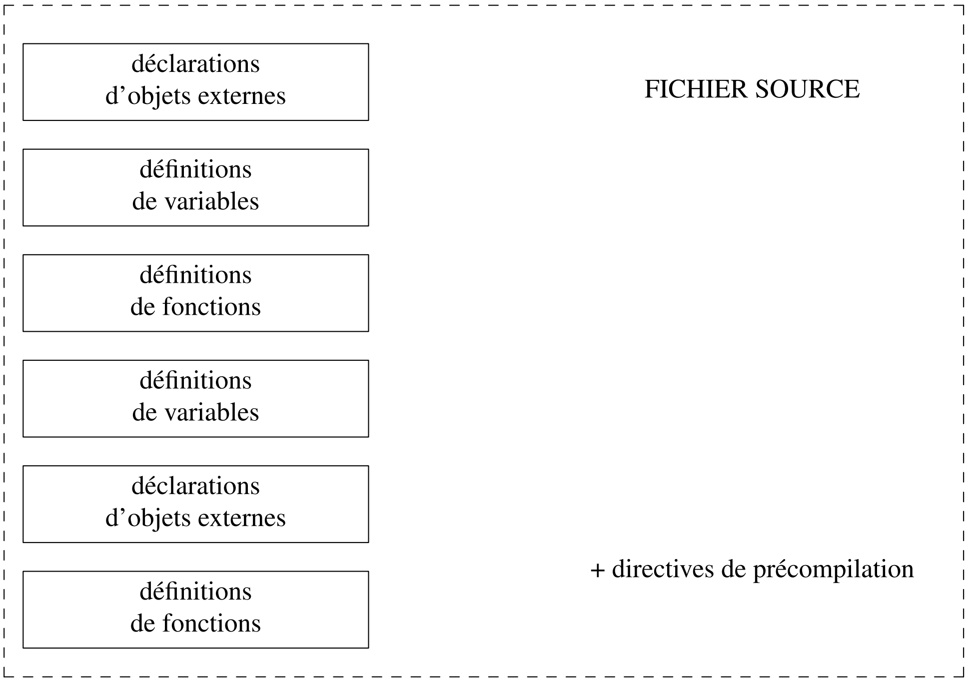

FIG. 9.2 – Survol d'un fichier source

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

TAB. 9.1 – Règle fondamentale de visibilité

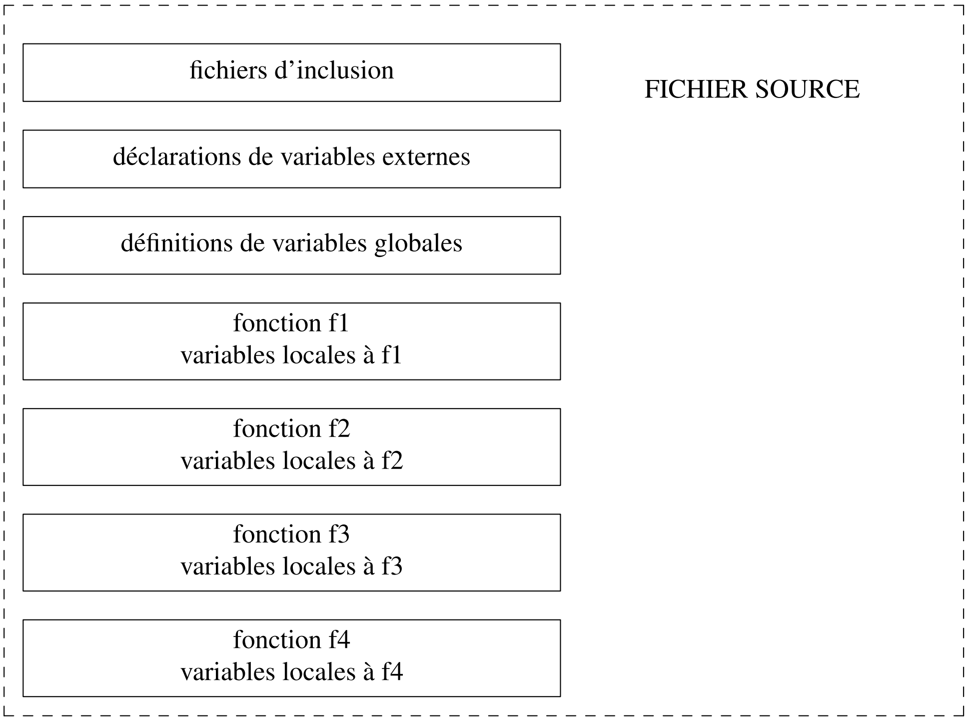

FIG. 9.3 – Exemple de visibilité

9.3 Visibilité

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

Comme il a été écrit dans le chapitre 1, le compilateur ne lit le fichier source qu'une seule fois, et 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 9.3 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 :

```
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)    /* Parametre de f3 */
{
    a = 10;         /* Modification du parametre local a f3 */
}
```

FIG. 9.4 – Masquage de nom

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 9.4 :
- 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 entier. 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 ;
- 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 entier. L'affectation `a = 10` réalisée dans la fonction `f3()` modifie l'argument local de type float et non la variable globale.

9.3.1 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

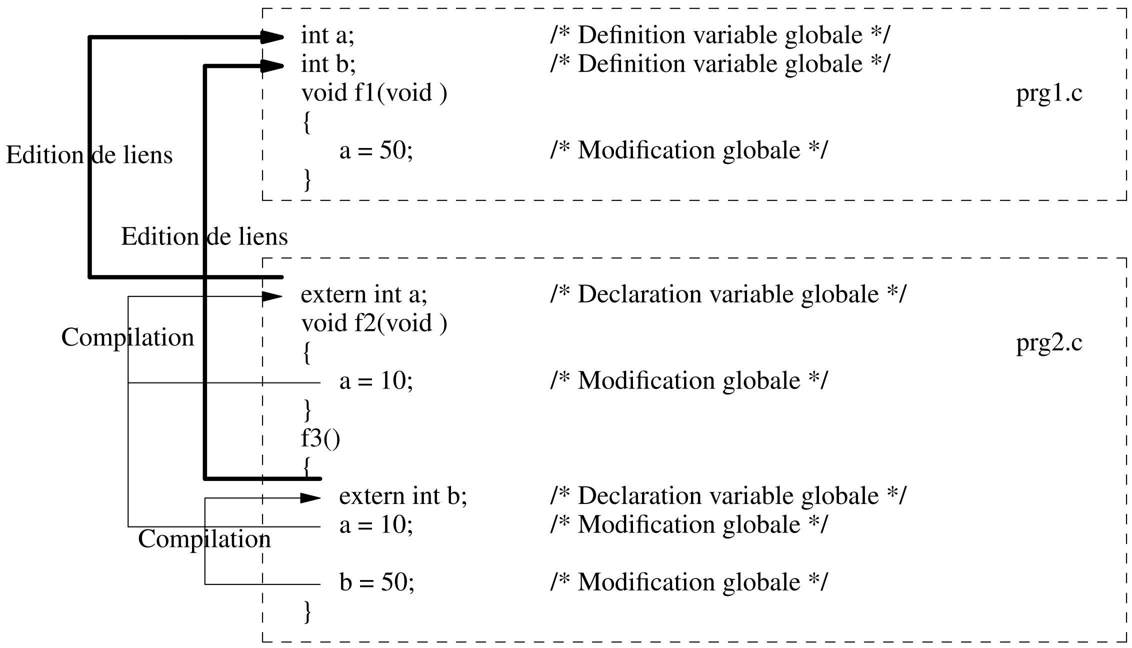

FIG. 9.5 – Visibilité des variables entre modules

fichiers doivent être associés par une édition de liens, sinon celle-ci se terminera avec des références non résolues.

La figure 9.5 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.c`. Ces variables peuvent être utilisées par les fonctions contenues dans ce fichier. De plus ces variables sont 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.c` ; elle peut donc être utilisée par toutes les fonctions définies dans `prg2.c`. 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.

9.4 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 8 sont décrites par les prototypes du programme 9.1.

[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.

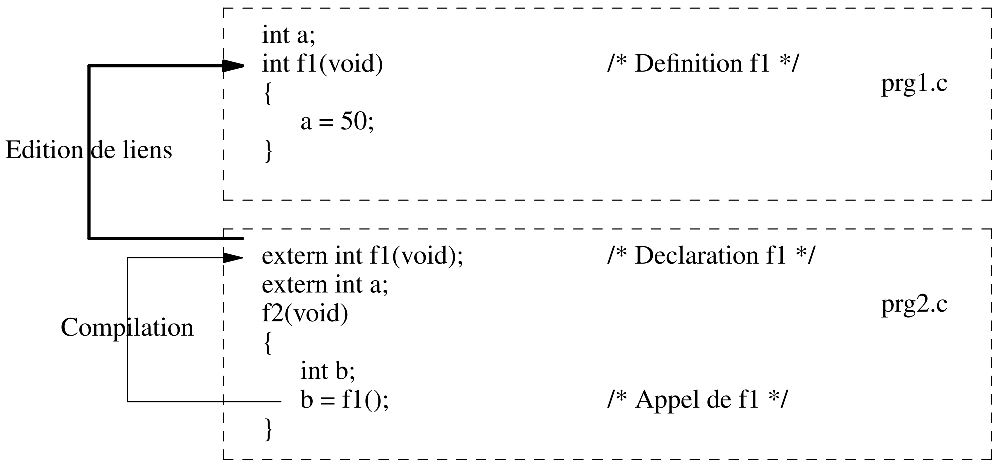

FIG. 9.6 – Visibilité des fonctions entre modules

Comme le montrent les lignes 6 et 7 du programme 9.1, 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.

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 9.1.

Programme 9.1 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);
```

9.5 Fonctions externes

Une fois connues, les fonctions des autres modules peuvent être utilisées. Comme le montre la figure 9.6, si la fonction `f1()` est définie dans le fichier `prg1.c` et que l'on veut l'utiliser dans le fichier `prg2.c`, il faut mettre en début de `prg2.c`, le prototype `extern int f1(void);`.

9.6 Fonctions définies ultérieurement

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 9.7, la déclaration `extern int f2(void)` permet l'utilisation d'une fonction sans en connaître le corps.

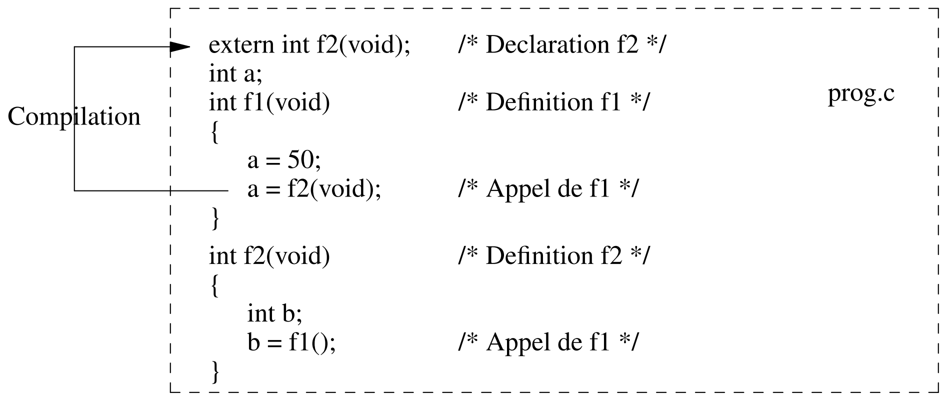

FIG. 9.7 – Visibilité des fonctions dans un module

9.7 Vérifications des prototypes

Dans la première version du langage, la liste des types d'arguments n'était pas mise dans le prototype des fonctions, ce qui n'informait pas le compilateur sur le nombre et le type des arguments. Le compilateur ne pouvait donc pas réaliser de contrôle de cohérence sur le nombre et le type des arguments passés lors de l'appel. Lorsqu'on travaille avec des vieux fichiers sources en langage C, la programmation doit donc être très soignée.

L'utilitaire `lint`, souvent présent avec la chaîne de compilation, fait des vérifications de ce type. La normalisation du langage C a introduit de manière complète l'utilisation des prototypes de manière à résoudre ce type de problème.

9.8 Multiples déclarations et définitions

En fait en langage C, 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 toujour associer la définition avec une initialisation,
- d'utiliser le mot `extern` devant les déclarations.

L'exemple suivant montre la déclaration multiple d'une variable.

`int a ;`
`void f1(void){ a= a+1 ;   }`
`int a ;`
`void f4(void){ a -- ;}   }`
	f1 et f4 manipulent la même variable a

Il est cependant plus académique d'écrire une seule définition et des déclarations.

`int a ;`
`void f1(void){ a= a+1 ;   }`
`extern int a ;`
`void f4(void){ a -- ;}   }`
	f1 et f4 manipulent la même variable a

9.8.1 Fichiers d'inclusion

Les fichiers d'inclusion sont destinés à contenir des déclarations [3] d'objets des types suivants :
- types non prédéfinis,
- 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 9.8 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 qui peuvent être utilisées par plusieurs modules sont `a` et `b` qui sont définies dans le module `prg1.c` ;
- les fonctions utilisables sont `f1()` qui est définie dans le module `prg1.c` et `f3()` qui est définie dans le module `prg2.c` ;
- le module `prg2.c`, 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.

9.8.2 Réduction de la visibilité

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 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 de 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 9.9, 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.

9.8.3 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.

[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.

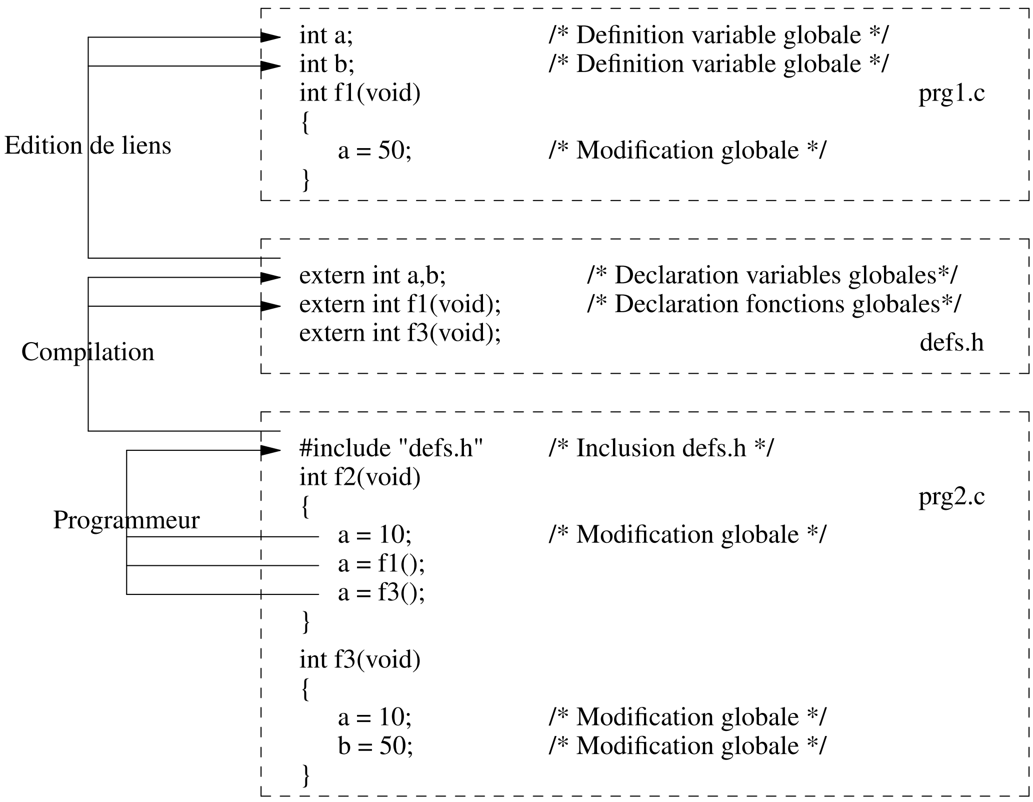

FIG. 9.8 – Utilisation de fichier d'inclusion

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

Trois choses 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. A 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 9.10 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.

Edition de liens

Compilation

Programmeur

prg1.c

```
int a;                  /* Variable globale du programme*/
static int b,c;         /* Variables globales au fichier f1*/
/* extern int f2();*/             /* Impossible  */
int f1()
{
     a = 50;            /* Modification globale au fichier f1 */
     b = 10;
}
```

defs.h

```
extern int a,c;         /* c du fichier prg2.c*/
extern int f1(),f3();             /* Declaration fonctions globales*/
```

prg2.c

```
#include "defs.h"       /* Inclusion defs.h */
static int f2()         /* Fonction utilisable dans prg2.c seulement */
{
     a = 10;            /* Modification globale */
     c = f1();          /* Variable du fichier prg2.c */
     a = f3();
}
int c;
int f3()
{
     a = 10;            /* Modification globale */
     c = 10;            /* Variable du fichier prg2.c */
     /* b = 50; */      /* Impossible */
}
```

FIG. 9.9 – Réduction de visibilité

```
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++)        /* Modification globale */
          f1();
}
main()
{
     f2();
}
```

FIG. 9.10 – Variables locales statiques

Chapitre 10

Pointeurs et tableaux

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.

10.1 Tableaux à une dimension

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

Comme le montre la figure 10.1, 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 10.2 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 10.1, il est aussi possible de ne pas spécifier la taille du tableau ou de ne pas initialiser tous les élements du tableau. Dans ce programme, `tb1` est défini comme un tableau de 6 entiers initialisés, et `tb2` est défini comme un tableau de 10

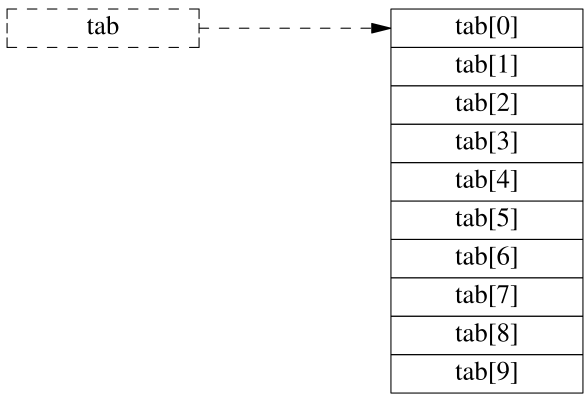

FIG. 10.1 – Tableau de dix entiers

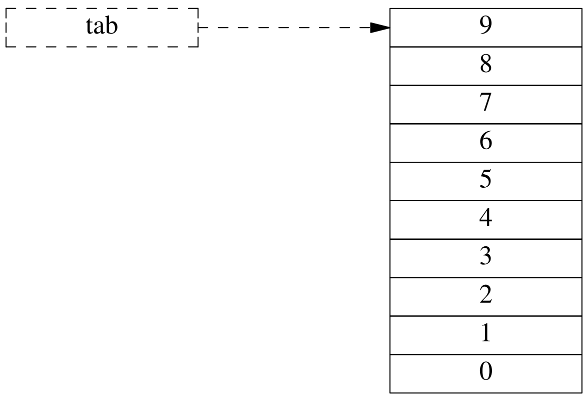

FIG. 10.2 – Tableau de dix entiers initialisé

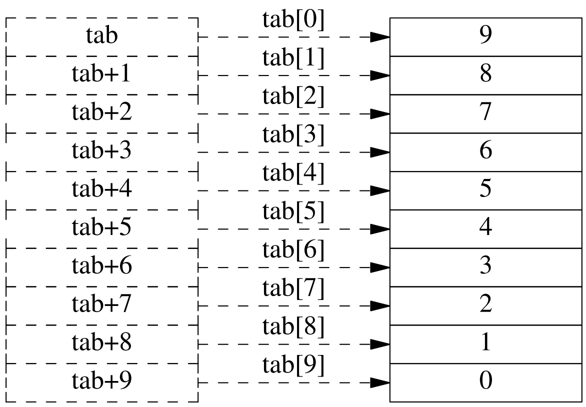

FIG. 10.3 – Adresses dans un tableau de dix entiers

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 (ensemble des octets à zéro quelle que soit la taille des éléments).

Programme 10.1 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].

10.2 Arithmétique d'adresse et tableaux

Comme nous venons de le dire le nom d'un tableau, correspond à l'adresse du premier élément du tableau, de plus les éléments d'un tableau sont du même type. Ces éléments 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 10.1). La figure 10.3 reprend l'exemple de la figure 10.2 en le complétant par les adresses de chaque élément. Ces adresses sont exprimées à partir du début du tableau.

[1] tb1= est une hérésie qui mérite l'exclusion de la communauté des utilisateurs du langage C.

tab →	tab[0][0]	tab[0][1]	tab[0][2]	tab[0][3]	tab[0][4]
tab+1 →	tab[1][0]	tab[1][1]	tab[1][2]	tab[1][3]	tab[1][4]
tab+2 →	tab[2][0]	tab[2][1]	tab[2][2]	tab[2][3]	tab[2][4]
tab+3 →	tab[3][0]	tab[3][1]	tab[3][2]	tab[3][3]	tab[3][4]
tab+4 →	tab[4][0]	tab[4][1]	tab[4][2]	tab[4][3]	tab[4][4]
tab+5 →	tab[5][0]	tab[5][1]	tab[5][2]	tab[5][3]	tab[5][4]
tab+6 →	tab[6][0]	tab[6][1]	tab[6][2]	tab[6][3]	tab[6][4]
tab+7 →	tab[7][0]	tab[7][1]	tab[7][2]	tab[7][3]	tab[7][4]

FIG. 10.4 – Tableau à deux dimensions

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

- 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 10.3, 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 $n^{\text{ième}}$ objet contenu dans le tableau à partir de l'adresse initiale. Dans ces conditions, `tab + n` est l'adresse du $n^{\text{ième}}$ 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 une valeur du type `ptrdiff_t` qui correspond au nombre d'objets entre les deux adresses. Ainsi, `&tab[5] - &tab[3]` doit donner la valeur 2 exprimée dans le type `ptrdiff_t`. De même, `&tab[3] - &tab[5]` retourne la valeur -2 exprimée dans le type `ptrdiff_t`.

10.3 Tableaux multidimensionnés

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 10.4 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];`.

10.4 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

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 ième objet à partir de l'objet courant.

TAB. 10.1 – Addition d'un entier à un pointeur

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.

TAB. 10.2 – Soustraction de deux pointeurs

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 niéme objet à partir de l'adresse initiale ;
- soustraction de deux pointeurs du même type ; ce qui calcule le nombre d'objet entre les adresses contenues dans les pointeurs (cette valeur est exprimée par un nombre entier).

Le tableau 10.1 est un exemple de manipulations en relation avec les pointeurs et les tableaux en utilisant les variables : `long x[10], *px , y;`

L'addition décrite dans la dernière ligne du tableau 10.1 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 10.2 est un exemple de soustraction à partir des définitions de variables suivantes :
`int i,tab[20], *pt1, *pt2;`

[2] Ce qui en général est égal à n'importe quoi.

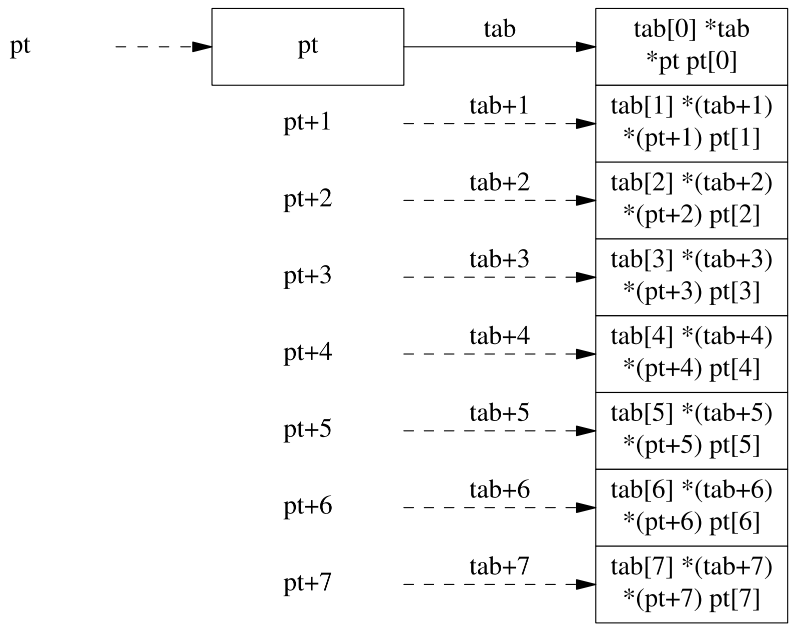

FIG. 10.5 – Pointeur et tableau

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 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]` ;
- `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 2ème 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 2ème élément du tableau.

Nous pouvons déduire de cette arithmétique de pointeur que : `tab[i]` **est équivalent à** `*(tab + i)`.

La figure 10.5 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;`

10.5 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 10.6 est un exemple d'utilisation à partir des définitions de variables suivantes :

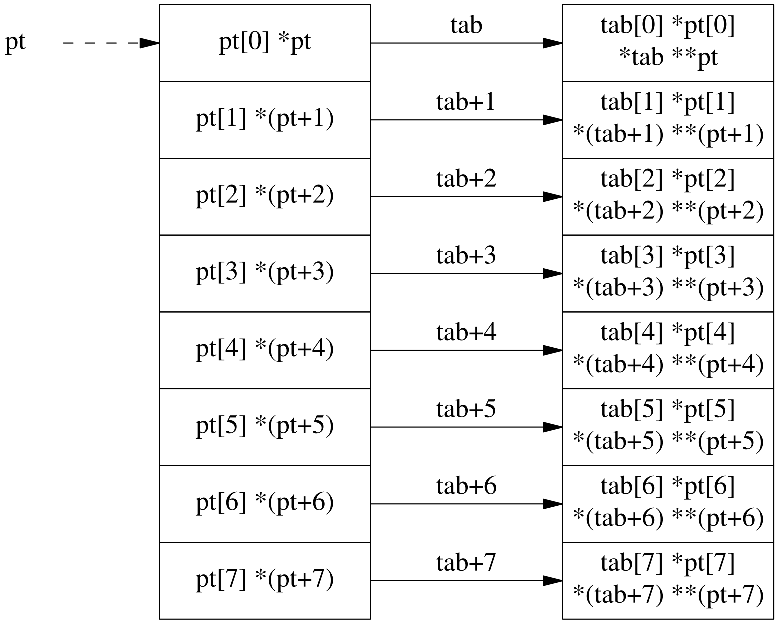

FIG. 10.6 – Tableau de pointeurs sur des variables dans un tableau

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

10.6 Pointeurs de tableau

Un pointeur peut contenir l'adresse d'un objet complexe comme un tableau dans sa globalité ou une structure dont nous parlerons dans le chapitre 11. Ce type de pointeur permet par exemple de manipuler les sous-tableaux d'un tableau à deux dimensions. Un pointeur sur un objet de type tableau se déclare par : `type (*nom)[taille] ;`

La figure 10.7 décrit les différentes façons d'accéder à des variables d'un tableau à deux dimensions à partir d'un tableau de pointeurs dont chaque pointeur contient l'adresse de début du sous-tableau de rang correspondant dans le tableau à deux dimensions.

```
int tab[6][5]; tableau de 6 fois 5 entiers.
int (*pt[6])[5]= {tab,tab+1,tab+2,tab+3,tab+4,tab+5};
```

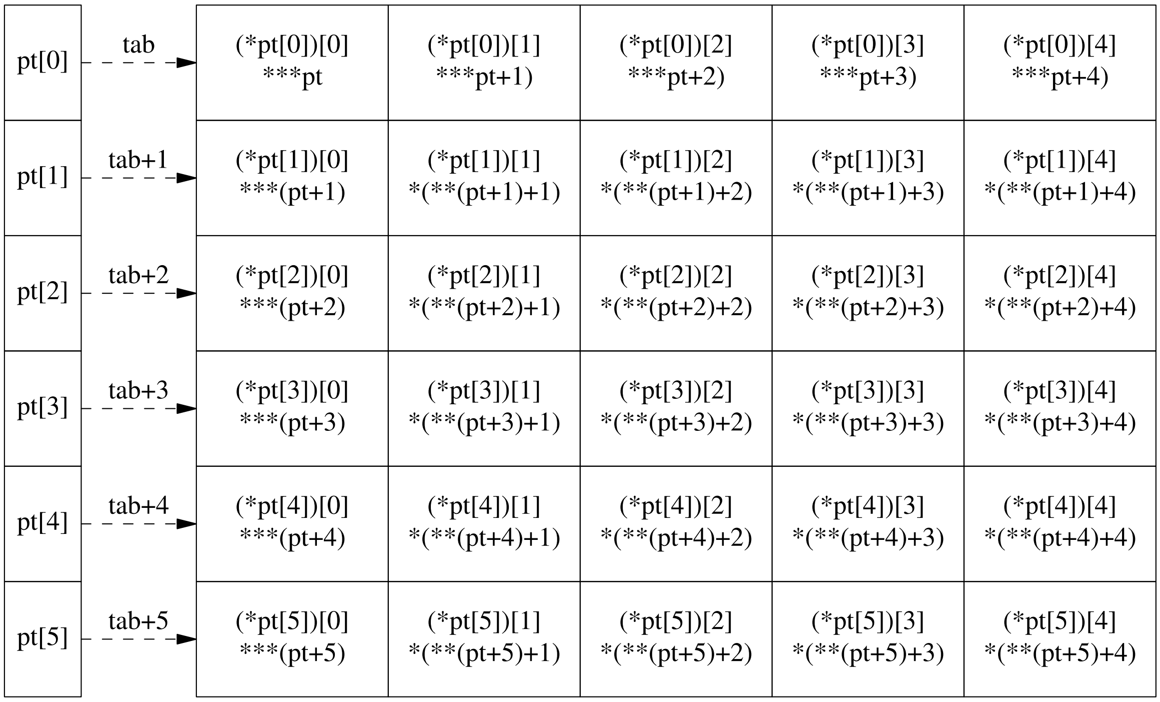

FIG. 10.7 – Tableau de pointeurs contenant des adresses de sous-tableaux

Chapitre 11

Structures

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

Par exemple une voiture peut être représentée par les renseigments suivants : la marque, la couleur, l'année, etc.

11.1 Définition de structures

Le langage C permet de définir des types de structures comme les autres langages évolués. Cela se fait selon la définition donnée par la table 11.1.

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 ;
} ;
```

```
struct   nom_de_structure    {
         type1 nom_champ1 ;
         type2 nom_champ2 ;
         type3 nom_champ3 ;
         type4 nom_champ4 ;
         type5 nom_champ5 ;
         type6 nom_champ6 ;
         etc.
         typeN nom_champ_N ;
}        variables ;
```

TAB. 11.1 – Syntaxe de la définition de structure

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

TAB. 11.2 – Adressage dans une structure

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 ;
```

Une fois ces définitions réalisées, nous pouvons utiliser les variables `obdate` et `ptdate` comme le montre le tableau 11.2.

Il est aussi possible de ne pas associer un nom de modèle à la définition de la structure. Les objets de ce type devront être déclarés immédiatement. Ce modèle de structure ne pourra plus être référencé par la suite, puisqu'il n'a pas de nom. Il est dit "anonyme".

```
struct {
        int jour ;
        int mois ;
        char annee[20] ;
} ob1, *pt1;
```

11.2 Utilisation de structures

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

11.2.1 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 11.2, donne les différentes possibilités d'affectation pour chaque champ.

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

Nous remarquerons l'équivalence 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.

11.2.2 Opérations sur la variable dans son ensemble

La normalisation du langage C a autorisé une pratique qui était déjà courante dans de nombreux compilateurs 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) ;
int resul ;
```

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 par exemple `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. `resul = checkdate(obdate2)`.

11.3 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 dit dans le II.4.a) que l'incrémentation d'un pointeur le faisait passer 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 structure 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++`; `ptnoeud` pointe sur `tab[1]`.

11.4 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 tableau 11.3.

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  nom_de_structure      {
        unsigned nom_champ1     : longueur1 ;
        unsigned nom_champ2     : longueur2 ;
        unsigned nom_champ3     : longueur3 ;
        unsigned nom_champ4     : longueur4 ;
        unsigned nom_champ5     : longueur5 ;
        unsigned nom_champ6     : longueur6 ;
        etc.
        unsigned nom_champ_N    : longueurN ;
}                             objets ;
```

TAB. 11.3 – Définition d'un modèle de structure champs de bits

```
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  ;
}   ;
```

Chapitre 12

Unions

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

12.1 Définition d'union

La définition d'une union respecte une syntaxe proche de celle d'une structure et qui est donnée dans la table 12.1.

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`.

```
union   nom_de_union        {
        type1 nom_champ1 ;
        type2 nom_champ2 ;
        type3 nom_champ3 ;
        type4 nom_champ4 ;
        type5 nom_champ5 ;
        type6 nom_champ6 ;
        etc.
        typeN nom_champ_N ;
}       variables ;
```

TAB. 12.1 – Syntaxe de la définition d'une union

12.2 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 11.2.1).

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 étés inventées pour utiliser un même espace mémoire avec des types de données différents dans des étapes différentes d'un même programme. Elles sont trés utilisées dans les compilateurs.

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

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

Programme 12.1 Utilisation du'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);
}
```

Dans l'exemple 12.1, 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`. 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 12.1 donne le résultat suivant :

```
Valeur de lmot.ent : -1
Valeur de lmot.ent1 : 65535
```

Chapitre 13

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 (voir chap. 15).

13.1 Définition

La définition d'une énumération respecte la syntaxe donnée dans la table 13.1.

```
enum   nom_de_énumération  {
       énumérateur1,
       énumérateur2,
       énumérateur3,
       énumérateur4,
       énumérateur5,
       etc.
       énumérateurN
}      variables;
```

TAB. 13.1 – Syntaxe de la définition d'une énumération

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.

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 1 pour l'énumérateur suivant.

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

Ce premier exemple d'énumération définit les constantes symboliques :

1. `rouge` qui correspond à la valeur 0 ;

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

Ce premier exemple utilise les possibilités standard des énumérations.

Nous pouvons aussi fixer une valeur à l'un des énumérateurs :

```
        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.

Voici un dernier exemple qui montre une utilisation des énumérations pour différencier les types de fichiers dans les inodes d'un système de fichiers de type UNIX.

```
enum typefic {
 normal = 0100000,
 repertoire = 0040000,
 special_bloc = 0060000,
 special_car = 0020000,
 sock_unix = 0140000,
 tube_nomme = 0010000,
 lien_symb = 0120000
};
```

13.2 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.

13.3 Limites des énumérations

Les énumérations sont des types synonymes (voir chap. 14) 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.

Chapitre 14

Types synonymes et objets complexes

Ce chapitre aborde la définition de type complexes.

14.1 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 prédéfini `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 10 `int`. La variable `tt` est un tableau de 10 entiers.

`typedef` est très semblable à une directive du préprocesseur (chap. 15) 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 ;
    int mois ;
    int annee ;
} date ;
```

et d'utiliser directement ce type. Ceci évite les répétitions fastidieuses du mot réservé `struct`. Cette définition est implicite en C++.

```
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 initialisé.

14.2 Types complexes

Nous allons nous intéresser aux types complexes sous l'angle du lecteur de programme écrit par un autre rédacteur. Ainsi notre premier objectif est de pouvoir comprendre une ligne de déclaration ou de définition de variable qui semble à première vue complexe. Pour réaliser cette lecture correctement nous appliquerons la règle suivante :

- il faut partir du nom de l'objet ;
- et ensuite il faut appliquer les règles de précédence des opérateurs telles que données dans le tableau 5.6.

Les parties importantes des règles de précédence sont :

- () et [] sont plus prioritaires avec une associativité de gauche à droite ;
- * est moins prioritaire avec une associativité de droite à gauche.

Le tableau 14.1 donne un ensemble de définitions de variables et décrit le type de chaque variable.

14.3 Fonctions et tableaux

Comme le montre les exemples précédents, il est possible de définir des tableaux de pointeurs sur des fonctions.

```
int (*t1[10])();
int *(*t2[10])();
```

Le premier exemple définit un tableau de 10 pointeurs qui peuvent recevoir des adresses de fonctions qui retournent un entier. Le second exemple définit un tableau de 10 pointeurs de fonctions qui peuvent recevoir des adresses de fonctions qui retournent un pointeur sur un entier.

En langage C, le nom d'une fonction est considéré comme une adresse et l'appel d'une fonction consiste à déréférencer l'adresse pour provoquer un saut du pointeur de programme courant à l'adresse correspondant au nom de la fonction.

Déclaration	Objet
`int *tab[10] ;`	tableau de 10 pointeurs sur des entiers
`int **ta[10] ;`	tableau de 10 pointeurs sur des pointeurs sur des entiers
`int *(*tb)[10] ;`	pointeur sur un tableau de 10 pointeurs sur des entiers
`int *fp() ;`	fonction qui retourne un pointeur sur un entier
`int (*fp)() ;`	pointeur sur une fonction qui retourne un entier

`union u {`
`  struct t1 u1 ;`
`  struct t2 u2 ;`
`}`
	Modèle d'union de deux structures de type
	t1 et t2

`int tt()[10] ;`	définition illégale fonction qui retourne un tableau.
`int tt[10]() ;`	définition illégale tableau de fonctions
`int (*t1[10])() ;`	Tableau de 10 adresses de fonctions qui retournent un entier
`int *(*t2[10])() ;`	Tableau de 10 adresses de fonctions qui retournent un pointeur d'entier
`int t3[] ;`	tableau de taille inconnue la définition de l'objet est externe
`int *t4[] ;`	tableau de pointeur externe
`int t5[][] ;`	déclaration illégale il faut connaitre la taille de la 2ème dimension
`int *t6[] ;`	Tableau de pointeur sur des entiers.
`int **t7 ;`	Pointeur sur un pointeur d'entier (Souvent utilisé pour décrire un tableau de pointeur.)
`float (*pf[5])()= {sin,cos}`	Tableau de pointeurs sur des fonctions qui retournent des float Les deux premières entrées du tableau sont initialisées par les adresses de cos et sin

TAB. 14.1 – Exemples d'objets complexes

Chapitre 15

Préprocesseur

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`
- définition de variables de préprocessing :
  - `#define NOM valeur`
  - `#undef NOM`
- définition de macrofonction ou macro-expression : `#define m(x) (128*(x)+342*(x)*(x))`
- sélection du code en fonction des variables du préprocesseur :
  - `#if`
  - `#ifdef`
  - `#ifndef`
  - `#else`
  - `#endif`

15.1 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.

La figure 15.1 montre le retrait des commentaires dans un fichier source avant l'introduction d'un nouveau commentaire.

La figure 15.2 montre le retrait des commentaires dans un fichier source après l'introduction du nouveau commentaire ce qui provoque une erreur de syntaxe sur la ligne `a = 2 ;` car le compilateur voit `*/ a = 2 ;`.

15.2 Inclusion de fichiers

L'inclusion de fichiers par le préprocesseur est déclenchée par la rencontre de la directive :

`#include nom_de_fichier`

Avant CPP	Après CPP
`int a,b,c ;`	`int a,b,c ;`
`a = 1 ;`	`a = 1 ;`
`b = 1 ;`	`b = 1 ;`
`c = 1 ;`	`c = 1 ;`
`/* maintenant ajoute a+b a c */`	
`c += a + b ;`	`c += a + b ;`
`a = 2 ;`	`a = 2 ;`

TAB. 15.1 – Traitement des commentaires

avant CPP	après CPP	Erreur
`int a,b,c ;`	`int a,b,c ;`	
`a = 1 ;`	`a = 1 ;`	
`b = 1 ;`	`b = 1 ;`	
`/* ceci ne nous interesse plus`		
`c = 1 ;`		
`/* maintenant ajoute a+b a c */`		
`c += a + b ;`	`c += a + b ;`	
`*/`	`*/`	erreur
`a = 2 ;`	`a = 2 ;`	

TAB. 15.2 – Erreur dûe au traitement des commentaires

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. par son chemin absolu, ce qui se traduit dans un système de type UNIX par le fait que le nom de chemin correspondant au fichier commence par "/" ;
   - `#include "/users/chris/essai/header.h"`
   - `#include` $< /users/chris/essai/header.h >$
2. à partir du catalogue courant, si le nom de fichier est entouré par des guillemets ;
   - `#include "header.h"`
   - `#include "h/mem.h"`
   - `#include "../h/uucp.h"`
3. 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. Ce catalogue est de manière standard `/usr/include` dans les systèmes UNIX.
   - `#include` $< stdio.h >$
   - `#include` $< sys/dir.h >$

Il est possible de demander au préprocesseur d'ajouter d'autres catalogues à sa recherche, en utilisant une option de compilation à l'appel de l'enchaîneur de passes. Cette option est dans un enchaîneur de passe UNIX `-Inom_du_catalogue`. Cette option peut être utilisée plusieurs fois de manière à spécifier plusieurs catalogues de recherche. Si le nom du fichier est entre guillemets, le préprocesseur cherche le fichier :

1. dans le catalogue courant,
2. puis dans les catalogues spécifiés par les options `-I`.

Si le nom du fichier est entre "<" et ">", le préprocesseur cherche le fichier :

1. dans les catalogues spécifiés par les options `-I`,
2. puis dans le catalogue par défaut du compilateur.

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++)

TAB. 15.3 – Utilisation d'une constante de compilation

Avant CPP	Après CPP
#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 ;
printf("Valeur de PI %d",PI) ;	printf("Valeur de PI %d",3.14159) ;
(PI) LG PI LGPI	(3.14159) 20 3.14159 LGPI

TAB. 15.4 – Interprétaion des variables par le préprocesseur

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.

15.3 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.

15.3.1 Définition de constantes de compilation

Comme le montre la table 15.3, 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
```

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, CPP 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 15.4 est un exemple d'utilisation des variables de précompilation.

15.3.2 Définition destinée à la sélection

CPP 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 précompilateur teste si elles sont définies ou non. 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`

15.3.3 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.

15.3.4 Définition à l'appel de l'enchaîneur de passes

Il existe deux possibilités pour définir les variables de précompilation à l'appel de l'enchaîneur de passes :

- en utilisant l'option `-Dnom` avec le nom de la variable de précompilation. Le fichier sera alors traité par le préprocesseur comme si la variable de précompilation `nom` était définie, par la directive `#define nom`.
- en utilisant l'option `-Dnom=valeur` avec le nom de la constante de précompilation suivi du signe "=" et de la valeur à associer à cette constante de précompilation. Le préprocesseur changera les occurrences de la variable `nom` par la valeur `valeur`, comme si la constante de précompilation `nom` était définie, par la directive `#define nom valeur`.

Il existe des variables de précompilation qui sont prédéfinies. Elles sont associées au type de la machine, au compilateur et à l'environnement.

Par exemple, sur une **SPS7**, les variables `sm90` et `68000` sont prédéfinies. De même, sur un **sun3**, les variables `sun` et `68020` sont prédéfinies. Ou encore, sur un **sun4**, les variables `sun` et `sparc` sont prédéfinies. Pour finir, les compilateurs à la norme ANSI doivent avoir une variable de préprocesseur prédéfinie appelée `__STDC__`.

Il est possible d'effacer ces définitions au niveau de l'appel à l'enchaîneur de passes, en faisant : `cc -Unom`.

Ceci ne peut pas inhiber les définitions qui apparaissent dans le code par des `#define`.

15.4 Définition de macro-expressions

CPP permet la définition de macro-expressions encore appelées macrofonctions. Ces macro-expressions sont très semblables à celles de macro-assembleurs. 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 attention à l'ordre d'évaluation pour des macro-expressions complexes, et la technique la plus simple est d'encadrer le nom des pseudo-variables dans l'expression de la macro. Comme le montre l'exemple 15.5, le parenthésage garantit que l'expression résultante est correctement interprétée.

15.5 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. Le principe de base consiste à passer ou à supprimer

Avant CPP	Après CPP	Bon/Mauvais

`#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;`
`int e,f,g;`
`a = b = c = 1;`
`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);`
		mauvais
		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);`
		mauvais
		
		bon

TAB. 15.5 – Evaluation de macros par CPP

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.

15.5.1 Sélection avec `#if`

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.

En plus des expressions du langage C, le préprocesseur permet de tester l'existence d'une variable en utilisant le mot `defined` dans la ligne de test.

```
#define DEBUG
#if defined DEBUG
  code passe
```

Avant CPP	Aprés CPP
`#define vrai 1` `#define faux 0`	

`#if vrai`
vrai_est_vrai
`#else`
vrai_est_faux
`#endif`
	vrai_est_vrai

`#if faux`
faux_est_vrai
`#else`
faux_est_faux
`#endif`
	faux_est_faux

`#if vrai || faux`
vrai_ou_faux_est_vrai
`#endif`
	vrai_ou_faux_est_vrai

`#if !faux`
not_faux_est_vrai
`#endif`
	not_faux_est_vrai

`#if vrai && !faux`
vrai_et_not_faux_est_vrai
`#endif`
	vrai_et_not_faux_est_vrai

`#if vrai + faux`
vrai_plus_faux_est_vrai
`#endif`
	vrai_plus_faux_est_vrai

`#if vrai * faux`
vrai_mult_faux_est_vrai
`#else`
vrai_mult_faux_est_faux
`#endif`
	vrai_mult_faux_est_faux

TAB. 15.6 – Exemple d'utilisation du #if

```
#endif
#if defined(DEBUG)
  code passe
#endif
```

15.5.2 Sélection avec #ifdef et #ifndef

La sélection avec `#ifdef` et `#ifndef` est semblable à celle avec `#if defined` ou `#if !defined`. Dans l'exemple 15.1, le test de compilation avec LOG permet de déterminer si des informations sont envoyées au système de trace du système d'exploitation. De plus le type de retour de la fonction `xalloc()` dépend du fait que le compilateur soit du type standard ANSI ou non. Comme

Programme 15.1 Exemples de sélection de code par `#ifdef`

```
#ifdef LOG
       if (LogLevel > 20)
               syslog(LOG_DEBUG, "%s: unlink %s",
                      CurEnv->e_id, f);
#endif /* LOG */
#ifdef __STDC__
void *
#else /* !__STDC__ */
char *
#endif /* __STDC__ */
xalloc(sz)
      register int sz;
{
#ifdef __STDC__
      register void *p;
#else /* !__STDC__ */
      register char *p;
#endif /* __STDC__ */
```

le montre l'exemple 15.2, les sélections peuvent être imbriquées.

Programme 15.2 Sélection de code par `#ifdef` imbriqués

```
#ifdef _PATH_SENDMAILPID
void
WritePid()
{
    extern char *PidFile;
    FILE *f;

    (void) unlink(PidFile); /* try to be safe :-) */
    if ((f = dfopen(PidFile, "w")) != NULL)
    {
            fprintf(f, "%d\\n", getpid());
            (void) chmod(PidFile, 0444);
            (void) fclose(f);
    }
# ifdef LOG
    else
            syslog(LOG_NOTICE, "Could not log daemon "
                             "pid %d to file %s: %m",
                    getpid(), PidFile);
# endif /* LOG */
}
#endif  /* _PATH_SENDMAILPID */
```

Chapitre 16

Entrée-sorties de la bibliothèque standard

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 la réalisation des entrée-sorties.

La bibliothèque standard contient :

1. les fonctions permettant la gestion des entrée-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.

L'ensemble des fonctions qui permettent de faire des entrée-sorties standard de bas niveau est appellé BIO (Basic Input Output).

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

Le fichier `stdio.h` contient :
- les définitions de macro-expressions ;
- les prototypes des fonctions ;
- les définitions de constantes : EOF, stdin, stdout, etc. ;
- la définition du tableau des fichiers ouverts.

Le langage C donne une vision "*UNIXienne*" des entrée-sorties. La même interface d'accès aux fichiers peut être utilisée sur les périphériques. 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.

16.1 Entrée-sorties standard

Conformément à la vision UNIXienne des choses, trois pseudo-fichiers sont ouverts par l'environnement de programmation lorsque le programme commence à s'exécuter. Ces trois pseudo-fichiers permettent l'accès au terminal de l'utilisateur.

Ils ont pour nom :

stdin pour fichier d'entrée standard ; ce fichier est le plus souvent associé au clavier ;

stdout pour fichier de sortie standard ; ce fichier désigne le plus souvent l'écran et les fonctions d'accès associées utilisent une technique de tampon[1] pour les écritures qui lui sont associées ;

stderr pour fichier d'erreur standard, ce fichier correspond lui aussi le plus souvent à l'écran mais les écritures sont non tamponnées.

Ces fichiers peuvent être redirigés au niveau de l'interprète de commandes [Bou78] par l'utilisation des symboles > et < à l'appel du programme. Ils sont automatiquement redirigés lorsque les commandes sont enchaînées par des tubes. Par exemple :
`prog >fichier` : lorsque prog écrira, ce sera redirigé vers fichier
`prog <fichier` : prog fera ses lectures dans fichier
`prog 2 >fichier` : prog écrira ses messages d'erreur dans fichier.
`prog1 | prog2` : prog1 a sa sortie standard redirigée dans l'entrée standard de prog2.

A ces fichiers standard sont associées des fonctions prédéfinies qui permettent de réaliser les opérations suivantes :
- lecture et écriture caractère par caractère ;
- lecture et écriture ligne par ligne ;
- lecture et écriture formatées.

16.1.1 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 15) ; nous les considérerons comme des fonctions pour simplifier la présentation.

int getchar(void) ;

synopsis : cette fonction permet de lire un caractère sur `stdin` 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 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 stdout.

argument : Elle est définie comme recevant un entier[2] 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.

Voici un programme C qui réalise la lecture de son fichier standard d'entrée caractère par caractère et qui reproduit les caractères de son fichier standard d'entrée sur son fichier standard de sortie. Son rôle est similaire à celui de l'utilitaire `cat(1)` du système UNIX lorsqu'il est utilisé comme filtre.

[1] Souvent nommé par l'anglicisme buffer, ce tampon permet de ne pas trop solliciter le système lors des écritures. Sa taille la plus courante est de un kilo-octets.

[2] 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.

Programme 16.1 Lecture et écriture caractère par caractère sur les fichiers standard

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

16.1.2 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 stdin ; 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 un 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 (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 stdout.
- **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.

Voici une nouvelle version du programme précédent de lecture et réécriture entre le fichier standard d'entrée et le fichier standard de sortie 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.

16.1.3 Échanges avec formats

Nous avons déjà parlé des entrée-sorties formatées dans le chapitre sur les éléments de base permettant de débuter la programmation (chapitre 4). Les fonctions permettant de faire des entrée-sorties formatées sur les fichiers standard d'entrée et de sortie sont les suivantes :

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

Programme 16.2 Lecture ligne par ligne sur les fichiers standard

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

synopsis : lecture formatée sur stdin.

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 stdout

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 4.2. Les possibilités en matière de format sont données de manière exhaustive dans la partie sur les entrée-sorties formatées (voir section 16.5).

16.2 Ouverture d'un fichier

A part les trois pseudo-fichiers dont nous avons parlé dans le paragraphe précédent, 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 ((void *)0) 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.

Programme 16.3 Ouverture d'un fichier

```
#include <stdio.h>
main(){
    FILE * MyFich;
    MyFich = fopen("/etc/passwd","r");
    ...
}
```

16.3 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 16.7).

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 la figure 16.4.

Programme 16.4 Ouverture et fermeture d'un fichier

```
#include <stdio.h>
main(){
    FILE * MyFich;
    MyFich = fopen("/etc/passwd","r");
    ...
    fclose(MyFich);
}
```

16.4 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.

16.4.1 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 16.8).

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 16.8).

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 (comportement identique à la commande `cat(1)` du système UNIX avec comme argument un seul nom de fichier). Le fichier est ouvert en mode lecture et il est considéré comme étant en mode texte.

16.4.2 Accès par ligne

Comme pour les entrée-sorties sur les pseudo-fichiers standard, 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 un caractère nul `'\0'` de manière à ce que la ligne une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.

Programme 16.5 Lecture caractère par caractère d'un fichier après ouverture

```
#include <stdio.h>
main(int argc, char *argv[]){
    FILE * MyFich;
    int TheCar;
    if(argc != 2) return 1;
    MyFic = fopen(argv[1],"r");
    if(MyFic == NULL) {
            printf("Cant open %s \n",argv[1]);
            return 2;
    }
    while((TheCar=fgetc(MyFic))!=EOF)
             fputc(TheCar,stdout);
    fclose(MyFic);
}
```

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 16.8.

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 un caractère nul). Elle doit contenir un retour chariot pour obtebir 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 16.8.

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 :

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 pseudo-fichiers stdin et stdout qui sont le plus souvent des terminaux (écran + clavier).

Les différences sont les suivantes :
– la fonction `gets()` :

Programme 16.6 Lecture ligne à ligne d'un fichier après ouverture

```
#include <stdio.h>
main(int argc, char *argv[])
{
    FILE * TheFic;
    char BigBuf[256];

    if(argc!=2) return 1;
    TheFic = fopen(argv[1],"r");
    if(TheFic == NULL) {
        printf("Cant open %s \n",argv[1]);
        return 2;
    }
    while(fgets(BigBuf,sizeof BigBuf, TheFic)!=NULL)
        fputs(BigBuf,stdout);
    fclose(TheFic);
}
```

   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.

16.4.3 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 16.7 ligne 13). 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.

Ces fonctions 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.

Programme 16.7 Lecture d'enregistrements dans un fichier

```
 1. #include <stdio.h>
 2. #include <stddef.h>
 3. struct automobile {
 4.     int age;
 5.     char couleur[20], numero[10], type[10], marque[10];
 6.   } ParcAuto[20];
 7.
 8. main(int argc, char *argv[])
 9. {
10.     FILE * TheFic;
11.     int i;
12.     size_t fait;
13.     TheFic = fopen("FicParcAuto","rb+");
14.     if(TheFic == NULL) {
15.          printf("Cant open FicParcAuto \n");
16.          return 1;
17.     }
18.     for(i = 0; i<20;i++){
19.          fait = fread(&ParcAuto[i],sizeof(struct automobile),1,TheFic);
20.          if(fait != 1){
21.              printf("Erreur lecture fichier parcauto \n");
22.              return 2;
23.         }
24.     }
25.     fclose(TheFic);
26. }
```

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 automobile),20,TheFic);`

ou bien encore par :

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

16.5 Entrée-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 [3] pour chaque variable simple ou pour un tableau de caractères [4].

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.

16.5.1 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** ou **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** ou **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`).
   (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 la table 16.1.

La **spécification** entre les crochets définit un alphabet[5] de caractères. La donnée lue doit être conforme à cette spécification. La lecture avec un format de spécification retourne une chaîne de caractères.

---

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

[4] Des formats particuliers permettent la lecture de plusieurs caractères à l'intérieur d'un tableau de caractères.

[5] 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. 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.

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 ou f ou g	nombre avec partie décimale en notation point décimal, ou notation exponentielle
c	caractère
s	mots ou chaîne de caractères sans blanc
[spécification]	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

TAB. 16.1 – Code de conversion pour scanf

16.5.2 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. 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** ou 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, ou 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, ou 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 la table 16.2.

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

code	conversion réalisée
%	écrit 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, 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, 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

TAB. 16.2 – Codes de conversion 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

TAB. 16.3 – Modificateurs de format pour printf

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 la table 16.3.

16.5.3 Conversion sur les entrée-sorties standard

Nous avons déjà exploré dans la section 16.1.3 les deux fonctions d'entrée-sorties standard formatées qui sont :

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

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

16.5.4 Conversion en mémoire

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

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.

16.5.5 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 16.8 ouvre le fichier `/etc/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).

Programme 16.8 Lecture avec format dans un fichier texte

```
#include <stdio.h>
main(int argc, char *argv[])
{
    FILE * TheFic;
    int  i,res;
    char Nom[10], Passwd[16],Gcos[128],Rep[255],Shell[255];
    int  Uid, Gid;

    TheFic = fopen("/etc/passwd","r");
    if(TheFic == NULL) {
        printf("Cant open %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 %s %s %s\n",Nom, Passwd, Uid, Gid, Gcos,
               Rep ,Shell);
    }
    fclose(TheFic);
}
```

16.6 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'entrée-sortie). Nous allons voir les fonctions qui permettent de connaitre 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 systeme 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 repositionner 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 16.9 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.

Cette modification est réalisée par le programme 16.9 selon les instructions C suivantes :

- la ligne 19 correspond à une lecture d'un enregistrement du fichier dans la zone mémoire `UneAuto` du type `struct automobile`.
- la ligne 24 modifie la valeur du champ age dans la structure en mémoire ;
- la ligne 25 modifie la position courante du fichier pour positionner le pointeur courant à l'adress de début de l'enregistrement qui est en mémoire.
- la ligne 28 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 la figure 16.10. La différence majeure entre ces deux exemples vient du fait que dans la version 16.10 le programme conserve l'information permettant de repositionner le pointeur courant dans le fichier, alors que dans le programme 16.9 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.

16.7 Gestion des tampons

Les entrée-sorties sont en général bufferisées[6] (sauf `stderr`).

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 entrée-sorties 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'entrée-sortie associé au fichier ouvert, cette zone doit avoir une taille prédéfinie dont la valeur est BUFSIZE. Elle peut être égale au pointeur NULL, ce qui rend les entrée-sorties du fichier non bufferisées.

[6] 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.

Programme 16.9 Modifications par déplacement dans un fichier

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

main(int argc, char *argv[])
{
    FILE * TheFic;
    int i;
    size_t fait;
    TheFic = fopen("FicParcAuto","r+b");
    if(TheFic == NULL) {
         printf("Cant open FicParcAuto \n");
         return 1;
    }
    for(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, (long) -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 fichier parcauto fait = %d \n",fait);
             return 4;
        }
    }
    fclose(TheFic);
}
```

Programme 16.10 Déplacements dans un fichier avec fgetpos()

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

main(int argc, char *argv[])
{
    FILE * TheFic;
    int i;
    size_t fait;
    fpos_t CurPos;
    TheFic = fopen("FicParcAuto","r+b");
    if(TheFic == NULL) {
         printf("Cant open FicParcAuto \n");
         return 1;
    }
    for(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 automobile),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 automobile),1,TheFic);
        if(fait != 1){
             printf("Erreur ecriture fichier parcauto fait = %d \n",fait
);.
             return 5;
        }
    }
    fclose(TheFic);
}
```

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'entrée-sortie 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 entrée-sorties de ce fichier seront totalement bufferisées (par exemple les écritures n'auront lieu que lorsque le tampon sera plein).

   _IOLBF signifie que les entrée-sorties seront bufferisées ligne par ligne (i.e. dans le cas de l'écriture un retour chariot provoque l'appel système).

   _IONBF les entrée-sorties sur le fichier sont non bufferisées.
4. la taille de la zone mémoire (buffer).

int fflush(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 entrée-sorties sur les terminaux sont bufferisées ligne par ligne. La fonction `fflush()` permet de forcer l'écriture des dernières informations.

16.8 Gestion des erreurs

Les erreurs des fonctions d'entrée-sorties 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'entrée-sorties 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.

Programme 16.11 Gestion des cas d'erreurs pendant la manipulation d'un fichier

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

main()
{
    FILE * TheFic;
    size_t fait;
    TheFic = fopen("FicParcAuto","r+b");
    if(TheFic == NULL) {
        perror("Cant open FicParcAuto");
        return 1;
    }
    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, (long) -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 FicParcAuto fait = %d \n",fait);
            break;
        }
        fflush(TheFic);
    }
    clearerr(TheFic);
    fclose(TheFic);
}
```

Chapitre 17

Autres fonctions de la bibliothèque standard

17.1 Fonctions de manipulation de chaînes de caractères

La bibliothèque standard fournit des fonctions de manipulation de chaînes de caractères. Il est nécessaire d'inclure le fichier $< string.h >$ pour avoir la définition des fonctions décrites dans la table 17.1.

17.2 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)** vrai si c est une lettre (alphabétique)
- **isupper(c)** vrai si c est une majuscule (upper case)
- **islower(c)** vrai si c est une minuscule (lower case)
- **isdigit(c)** vrai si c est un chiffre
- **isspace(c)** vrai si c est un blanc, interligne ou tab
- **ispunct(c)** vrai si c est caractère de ponctuation
- **isalnum(c)** vrai si c est alphabétique ou numérique
- **isprint(c)** vrai si c est écrivable de 040 à 0176
- **isgraph(c)** vrai si c est graphique de 041 à 0176
- **iscntrl(c)** vrai si c est del (0177) ou un caractère de contrôle (<040)
- **isascii(c)** vrai si c est ASCII (<0200)

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.

Déclaration	Travail	Retour
char *s1, *s2 ; int n ; char *strcat (s1, s2) char *strncat (s1, s2, n)	 ajoute la chaîne s2 derrière la chaîne s1 ajoute au plus n caractères	 pointeur sur s1 pointeur sur s1
int strcmp (s1, s2) int strncmp (s1, s2, n)	compare s1 et s2 sur au plus n caractères	nombre positif, nul, négatif s1>s2 , s1==s2 , s1<s2
char *strcpy (s1, s2) char *strncpy (s1, s2, n)	copie s2 dans s1 au plus n caractères	
char *s ; int strlen (s)		 taille de s
int c ; char *strchr (s, c) char *strrchr (s, c) char c ; char *index(s, c) char *rindex(s, c)	 cherche caractère c dans s idem que strchr idem que strrchr	 pointeur sur première occurrence pointeur sur dernière occurrence
char *strpbrk (s1, s2)	cherche 1 car de s2 dans s1	pointeur sur première occurrence
int strspn (s1, s2) int strcspn (s1, s2)	1^{er} motif dans s1 ne contenant que des car de s2 1^{er} motif dans s1 ne contenant aucun car de s2	longueur longueur

TAB. 17.1 – Fonctions de manipulation de chaînes

17.3 Quelques fonctions générales

17.3.1 system()

Il est possible de demander au système l'exécution d'un utilitaire. La fonction **system(s)** provoque l'exécution de l'utilitaire correspondant à la chaîne de caractère s.

Exemple : system("date") ;

Nous avons peu parlé d'implantation de données en mémoire. Il est possible de gérer une partie de l'espace dynamiquement par les fonctions `calloc()` et `cfree()`.

– **calloc(n, sizeof (objet))** alloue de la place pour n objets et retourne un pointeur sur la zone.
– **cfree(p)** restitue l'espace pointé par p où p a été obtenu par un calloc.

17.3.2 exit()

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 le cas du programme 17.1 qui doit lire un fichier. Si ce fichier est absent, la fonction qui est chargée de l'ouvrir peut décider l'arrêt du programme. Nous allons écrire cette fonction.

Programme 17.1 Utilisation de l'appel système exit()

```
int ouvre(const char *nom_fichier)
{
    int o ;
    o = open(nom_fichier, 0) ;
    if( o == -1 )
    {
        printf("impossible d'ouvrir %s\n", nom_fichier) ;
        exit(1) ;
    }
    return o ;
}
```

Table des Exemples

Table des figures

Liste des tableaux

Bibliographie

[BK78] D.M. Ritchie B.W. Kernighan. *The C Programming Language.* Prentice Hall Inc., Englewood Cliffs, New Jersey, Mars 1978.

[BK88] D.M. Ritchie B.W. Kernighan. *The C Programming Language Second Edition.* Prentice Hall Inc., Englewood Cliffs, New Jersey, 1988.

[Bou78] S. Bourne. The UNIX Shell. *Bell System Technical Journal*, 57(6) :1971–1990, July/August 1978.

[Dax92] P. Dax. *Langage C 7eme Edition.* Eyrolles, 61, Boulevard Saint-Germain, 75005 Paris, 1992.

[Dij65] E.W. Dijkstra. Programming Considered as a Human Activity. In *Proc. IFIP Congress*, pages 213–217, 1965.

[Dij68] E.W. Dijkstra. GO TO Statements Considered Harmful. *Communications of the ACM*, 11(3) :147–148, Mars 1968.

[DR78] S.C. Johnson D.M. Ritchie. Portabilty of C Programs and the UNIX Operating System. *Bell System Technical Journal*, 57(6) :2021–2048, July/August 1978.

[ISO90] ISO/IEC, editor. *Programming Language C.* ISO/IEC, 1990.

[SJ78] M.E. Lesk S.C. Johnson. Language Development Tools. *Bell System Technical Journal*, 57(6) :2155–2176, July/August 1978.

[Str86] B. Stroustrup. *The C++ Programming Language.* Prentice Hall Inc., Englewood Cliffs, New Jersey, 1986.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4) :221–227, Avril 1971.

[Wir74] N. Wirth. On the Composition of Well Structured Programs. *ACM Computing Survey*, 6(4) :247–259, Decembre 1974.

Index