Support de Cours de Langage C

Christian Bac

2 juillet 2003

©Christian Bac 1985-2003

Table des matières

Avant-propos **xi**

1 Historique et présentation **1**

1.1 Historique 1

1.2 Présentation du langage 2

1.3 Idées fondamentales 3

1.4 En-chaîneur de passes 5

1.4.1 Pré-processeur 5

1.4.2 Compilateur 5

1.4.3 Optimiseur de code 7

1.4.4 Assembleur 7

1.4.5 Éditeur de liens 7

1.4.6 Quelques options de cc 7

2 Généralités sur la syntaxe **9**

2.1 Mise en page 9

2.1.1 Identifiant 9

2.1.2 Espaces lexicaux 9

2.2 Mots réservés 10

2.3 Constantes 10

2.4 Instructions 10

3 Types et variables **11**

3.1 Types de base 11

3.1.1 Types entiers 12

3.1.2 Types avec parties décimales 12

3.1.3 Tailles des types 12

3.2 Constantes associées aux types de base 13

3.2.1 Constantes de type entier 13

3.2.2 Constantes avec partie décimale 14

3.2.3 Constantes de type caractère 14

3.2.4 Chaînes de caractères 15
3.3 Qualificatifs 15
3.4 Taille et normalisation 15
3.5 Définition de variables 16
3.6 Types dérivés des types de base 18
3.6.1 Tableaux et structures 18
3.6.2 Pointeurs 19
3.7 Initialisation de variables 20
3.8 Conversion de type 22
3.8.1 Conversions implicites 22
3.9 Déclaration et définition 23
3.10 Exercices sur les types et variables 23
3.10.1 Exercice 1 23
3.10.2 Exercice 2 24
3.10.3 Exercice 3 25
3.10.4 Exercice 4 25

4 Eléments de base 27
4.1 Bonjour 27
4.2 Lire et écrire 28
4.3 Quelques opérations 29
4.4 Plus sur `printf()` et `scanf()` 29
4.5 Exercices sur `printf()` et `scanf()` 31
4.5.1 Exercice 1 31
4.5.2 Exercice 2 31
4.5.3 Exercice 3 31
4.5.4 Exercice 4 32

5 Opérateurs et expressions 37
5.1 Opérateurs un-aires 37
5.1.1 Opérateur de référencement 38
5.1.2 Opérateur de déréférencement ou indirection 38
5.1.3 Utilisation des & et * 38
5.1.4 Opérateurs d'incrémentation et de décrémentation 39
5.1.5 Opérateur de taille 40
5.1.6 Opérateur de négation logique 40
5.1.7 Plus et moins unaires 40
5.1.8 Complément à un 40
5.2 Opérateurs binaires 40
5.2.1 Opérateurs arithmétiques 41

©Christian Bac 1985-2003

5.2.2 Opérateurs manipulant les bits . . . 41
5.2.3 Opérateurs de décalage . . . 42
5.2.4 Opérateurs de relation . . . 43
5.2.5 Opérateur binaire d'affectation et de succession . . . 44
5.2.6 Opérateurs d'affectation . . . 44
5.2.7 Opérateur ternaire . . . 45
5.2.8 Précédence des opérateurs . . . 45
5.3 Expressions . . . 47
5.4 Instructions . . . 47
5.5 Exercices sur les opérateurs et les expressions . . . 47
5.5.1 Exercice 1 . . . 47
5.5.2 Exercice 2 . . . 48
5.5.3 Exercice 3 . . . 48
5.5.4 Exercice 4 . . . 48
5.5.5 Exercice 5 : Operateur ternaire . . . 49
5.5.6 Exercice 6 . . . 49

6 Instructions de contrôle **57**
6.1 Instructions conditionnelles . . . 57
6.1.1 Test . . . 57
6.1.2 Table de branchement . . . 58
6.2 Instructions itératives . . . 60
6.2.1 `while` . . . 60
6.2.2 `for` . . . 61
6.2.3 `do while` . . . 62
6.2.4 Remarques sur les instructions itératives . . . 63
6.3 Ruptures de séquence . . . 63
6.3.1 `continue` . . . 63
6.3.2 `break` . . . 63
6.3.3 goto . . . 64
6.3.4 `return` . . . 65
6.4 Exercices sur les instructions de contrôle . . . 65
6.4.1 Exercice 1 . . . 65
6.4.2 Exercice 2 . . . 66
6.4.3 Exercice 3 . . . 66
6.4.4 Exercice 4 . . . 66
6.5 Exercices sur les ruptures de séquence . . . 66
6.5.1 Exercice 5 . . . 66
6.5.2 Exercice 6 . . . 66

©Christian Bac 1985-2003

7 Programmation structurée **83**
7.1 Historique . . . 83
7.2 Idées fondamentales . . . 83
7.3 Langage C et programmation structurée . . . 84
7.3.1 Ambitions du langage C . . . 84
7.3.2 C et structures fondamentales . . . 84
7.4 Quelques exemples . . . 84
7.4.1 Exemple avec des tests . . . 85
7.4.2 Exemple avec une boucle . . . 86

8 Fonctions **95**
8.1 Définition d'une fonction . . . 95
8.2 Retour de fonction . . . 96
8.3 Passage des paramètres . . . 97
8.3.1 Passage de constantes . . . 97
8.3.2 Passage de variables . . . 99
8.4 Utilisation de pointeurs en paramètres . . . 99
8.5 Conversion de type lors des appels . . . 102
8.6 Récursivité . . . 102
8.7 Arguments de la fonction main() . . . 103
8.8 Pointeur de fonction . . . 103
8.9 Étapes d'un appel de fonction . . . 103
8.10 Exercices sur les fonctions . . . 106
8.10.1 Exercice 1 . . . 106
8.10.2 Exercice 2 . . . 106
8.10.3 Exercice 3 . . . 106

9 Compilations séparées **111**
9.1 Programme . . . 111
9.2 Fichier source . . . 111
9.3 Visibilité . . . 113
9.3.1 Espaces de nommage et visibilité . . . 114
9.3.2 Extension de la visibilité . . . 114
9.4 Prototypes des fonctions . . . 114
9.5 Fonctions externes . . . 115
9.6 Fonctions définies ultérieurement . . . 116
9.7 Vérification des prototypes . . . 116
9.8 Multiples déclarations et définitions . . . 117
9.8.1 Fichiers d'inclusion . . . 117
9.8.2 Réduction de la visibilité . . . 118

©Christian Bac 1985-2003

9.8.3 Variables locales rémanentes . . . 119
9.8.4 Travailler en groupe . . . 120
9.9 Exercices sur les fonctions et la visibilité des variables . . . 120
9.9.1 Exercice 1 : simulation d'un ascenseur . . . 120
9.9.2 Exercice 2 : racines d'une équation du deuxième degré . . . 121
9.9.3 Exercice 3 : utilisation des fichiers d'inclusion . . . 121

10 Pointeurs et tableaux 129
10.1 Tableaux à une dimension . . . 129
10.2 Arithmétique d'adresse et tableaux . . . 130
10.3 Tableaux multidimensionnels . . . 131
10.4 Pointeurs et tableaux . . . 132
10.5 Tableau de pointeurs . . . 133
10.6 Pointeurs vers un tableau . . . 133
10.7 Exercices sur les tableaux et les pointeurs . . . 135
10.7.1 Exercice 1 : tri de tableaux d'entiers . . . 135

11 Structures 143
11.1 Définition . . . 143
11.2 Utilisation . . . 144
11.2.1 Opérations sur les champs . . . 144
11.2.2 Opérations sur la variable dans son ensemble . . . 144
11.3 Structures et listes chaînées . . . 145
11.4 Champs de bits . . . 145
11.5 Exercices sur les structures . . . 146
11.5.1 Exercice 1 . . . 146
11.5.2 Exercice 2 . . . 147

12 Unions 149
12.1 Définition . . . 149
12.2 Accès aux champs . . . 150
12.3 Exercices sur les unions et les champs de bits . . . 150
12.3.1 Exercice 1 . . . 150
12.3.2 Exercice 2 . . . 151

13 Énumérations 153
13.1 Définition . . . 153
13.2 Utilisation . . . 154
13.3 Limites des énumérations . . . 155

14 Types synonymes et complexes 157

©Christian Bac 1985-2003

14.1 Types synonymes 157
14.2 Types complexes 158
14.3 Fonctions et tableaux 158
14.4 Exercices sur les déclarations complexes 160
14.4.1 Exercice 1 160
14.4.2 Exercice 2 160
14.4.3 Exercice 3 161

15 Préprocesseur 165
15.1 Commentaires 165
15.2 Inclusion de fichiers 165
15.3 Variables de pré-compilation 168
15.3.1 Définition de constantes de compilation 168
15.3.2 Définition destinée à la sélection 168
15.4 Définition de macro-expressions 168
15.5 Effacement d'une définition 169
15.6 Définition à l'appel de l'enchaîneur de passes 169
15.7 Sélection de code 170
15.7.1 Sélection avec `#if` 171
15.7.2 Sélection avec `#ifdef` et `#ifndef` 171
15.8 Exercices sur le préprocesseur 171
15.8.1 Exercice 1 171
15.8.2 Exercice 2 171

16 Entrées-sorties de la bibliothèque 179
16.1 Entrée-sorties standards 179
16.1.1 Échanges caractère par caractère 180
16.1.2 Échanges ligne par ligne 181
16.1.3 Échanges avec formats 182
16.2 Ouverture d'un fichier 182
16.3 Fermeture d'un fichier 184
16.4 Accès au contenu du fichier 184
16.4.1 Accès caractère par caractère 185
16.4.2 Accès ligne par ligne 186
16.4.3 Accès enregistrement par enregistrement 187
16.5 Entrée-sorties formatées 189
16.5.1 Formats : cas de la lecture 190
16.5.2 Formats : cas de l'écriture 191
16.5.3 Conversion sur les entrée-sorties standards 192
16.5.4 Conversion en mémoire 192

©Christian Bac 1985-2003

16.5.5 Conversion dans les fichiers 193
16.6 Déplacement dans le fichier 194
16.7 Gestion des tampons 195
16.8 Gestion des erreurs 196

17 Autres fonctions de la bibliothèque **203**
17.1 Fonctions de manipulation de chaînes de caractères 203
17.2 Types de caractères 203
17.3 Quelques fonctions générales 204
17.3.1 `system()` 204
17.3.2 `exit()` 204

GNU Free Documentation License **207**
17.4 Applicability and Definitions 207
17.5 Verbatim Copying 208
17.6 Copying in Quantity 208
17.7 Modifications 209
17.8 Combining Documents 210
17.9 Collections of Documents 210
17.10 Aggregation With Independent Works 210
17.11 Translation 211
17.12 Termination 211
17.13 Future Revisions of This License 211

Liste des programmes exemples **211**

Liste des figures **215**

Liste des tableaux **218**

Bibliographie **220**

Index **221**

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Avant-propos

Notice de copyright

NOTICE de COPYRIGHT ©

Ce support de cours correspond au cours de langage C de Christian Bac (ci-après l'auteur, je ou moi) de l'Institut National de Télécommunications, 9 rue Charles Fourier 91011, Évry, France.

Ce document est disponible aux formats latex, PostScript ou HTML, sur le web à l'URL :
`http ://picolibre.int-evry.fr/projects/coursc/`.

Il est fourni tel quel, l'auteur ayant fait de son mieux pour supprimer les erreurs, sans garantie sur son utilisabilité, ni sur l'exactitude des informations qu'il contient.

Vous avez le droit de copier, distribuer et/ou modifier ce document selon les termes de la licence de documentation libre, version 1.1 ou toute version postérieure publiée par la Free Software Foundation ; avec une section invariante le chapitre **Avant-propos** qui contient cette licence, sans texte spécifique en première et en quatrième de couverture. Une copie de la licence est inclue dans le chapitre 17.3.2 intitulé "GNU Free Documentation License".

Toute remarque ou erreur peut être notifiée à l'auteur à l'adresse électronique suivante :

`Christian.Bac AT int-evry.fr`

Fin de la NOTICE de COPYRIGHT ©

Conventions d'écriture

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* et pour les locutions latines comme *versus* ;
- 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 non des variables.

Vocabulaire courant

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 seulement par le haut. Ainsi, les derniers éléments qui ont été ajoutés sur une pile sont les plus facilement accessible, une pile représente le modèle LIFO (*Last In First Out*) les derniers éléments qui y sont ajoutés sont les premiers à pouvoir être retirés. 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).

Un peu d'histoire

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 tapée avec les accents français 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.

©Christian Bac 1985-2003

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 évoluer de manière indépendante.

LaTeX 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` et il constitue une sorte de référence lorsque je veux rédiger un nouveau document ;
2. grâce à des outils comme `LaTeX2HTML`, je peux permettre aux autres de consulter ce support 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. Conan, 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`.

GnuFDL En 2003, je suis convaincu des vertus du partage au travers d'Internet, non seulement des codes mais aussi du savoir, j'ai placé ce cours sous licence GNU Free Documentation Licence pour lui permettre de continuer à vivre et pour faciliter son utilisation par mes collègues enseignants francophones. Pour l'occasion, j'en ai profitté pour associer à ce support quelques exercices et leurs corrigés qui étaient sur un support séparé.

Je tiens enfin à remercier les internautes qui m'envoient des correctifs ou des remarques qui permettent à ce cours de s'améliorer.

©Christian Bac 1985-2003

©Christian Bac 1985-2003

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.

Ces deux langages prédécesseurs du C avaient la particularité d'être sans type. Ils ne travaillaient que sur des données décrites par un mot machine ce qui leur donnait 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 manipuler un octet alors que son mot mémoire est de 2 octets, il est nécessaire pour utiliser les fonctionnalités du PDP11 introduire un type de donnée char et un type int. Ces notions de type n'étant pas prises en compte par le langage B, elles le seront dans son successeur le C.
- 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 [ISO89] ;
- 1999 première révision de la norme ISO/IEC 9899 [ISO99].

[1] À l'époque UNIX était une marque déposée des laboratoires Bell. Aujourd'hui, c'est devenu un nom quasiment commun :-), bien que la marque soit toujours la propriété de *The Open Group* qui pratique des programmes de certification.

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 [ISO89] 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, pré-processeur). 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.

Le compilateur de langage C se limite aux fonctionnalités qui peuvent être traduites efficacement en instructions machine.

La règle de fonctionnement du compilateur C, édictée ci-dessus, permet de détecter ce qui est fait directement par le compilateur lui-même et ce qui ne peut l'être. 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(bits) ou des groupes d'éléments binaires. 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 décimales.
- Il permet de manipuler des caractères en considérant qu'un caractère est représenté par un octet à partir du code ASCII[2].

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

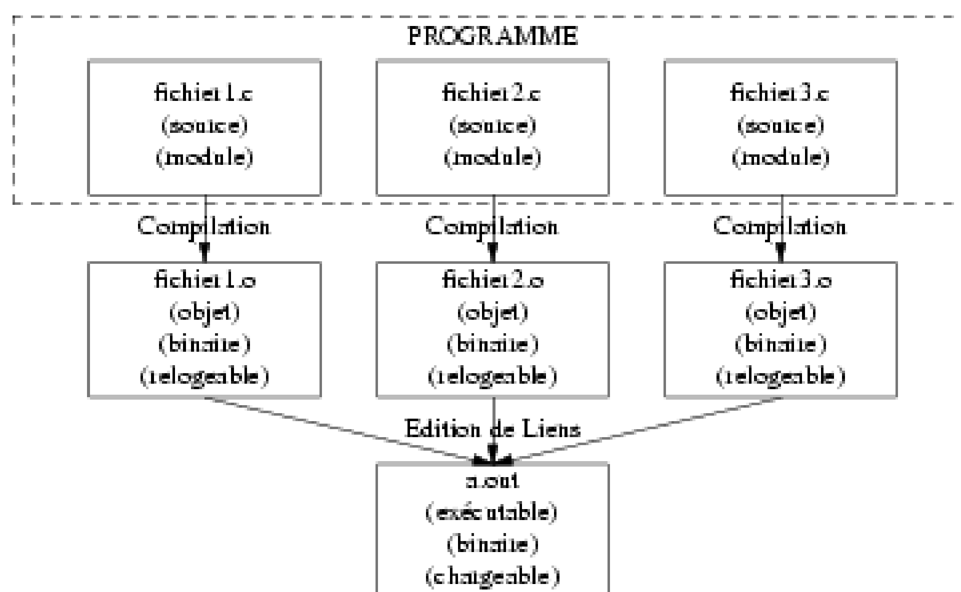

FIG. 1.1 – Structure d'un programme C

- 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.

Chacune des parties peut être regroupée dans un ou plusieurs fichiers de langage C que l'on appelle aussi module.

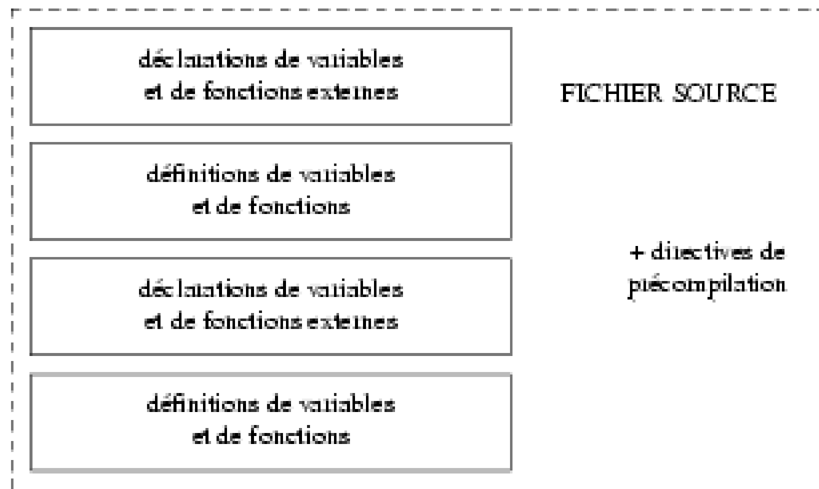

FIG. 1.2 – Fichier source

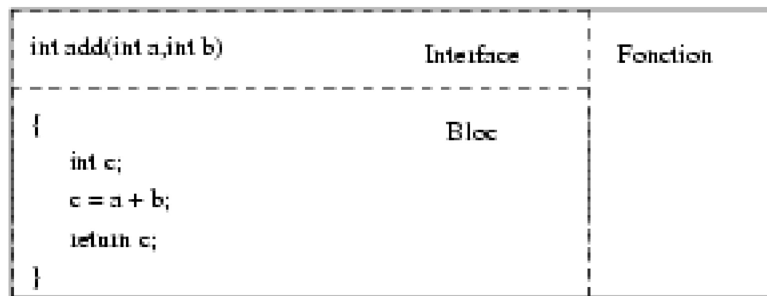

FIG. 1.3 – Structure d'une fonction C

Prenons pour exemple un programme qui enregistre une série de noms et notes d'élèves. Chaque nom est associé à une note. Une fois la saisie terminée le programme trie la liste des élèves par rang de note et affiche cette liste ainsi triée. Puis il trie la liste des élèves par ordre alphabétique à partir de leurs noms et stocke cette liste dans un fichier sur disque. Ce type d'application peut se découper en trois parties correspondant à la figure 1.1 :

– la partie interactive qui échange les informations avec l'utilisateur (`fichier1.c` ;
– la partie de calcul qui dans ce cas est un tri (par note ou par nom) (`fichier2.c`) ;
– la partie stockage des données sur disque une fois ces données triées (`fichier3.c`).

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 le prototype des fonctions utilisées dans ce module mais définies dans d'autres modules ;
– 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 pré-processeur).

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 de ses 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.

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

©Christian Bac 1985-2003

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 d'un point virgule (;) ;
- ou une instruction de contrôle de flot (test, boucle, rupture de séquence).

Nous parlerons en détail des possibilités du pré-processeur dans le chapitre 15.

1.4 En-chaîneur de passes

L'étude des différentes actions entreprises lors de ce que nous appelons de manière abusive la compilation, permet de mieux comprendre de quel outils proviennent les différentes caractéristiques de la programmation en langage C.

Nous prenons le cas des outils utilisés dans les systèmes de type UNIX 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 en-chaîneurs de passes ; nous allons voir l'usage, le fonctionnement, et les options de `cc`. Cet outils sert à appeler les différents utilitaires nécessaires à la transformation d'un programme C en un fichier exécutable. L'en-chaîneur de passes `cc` met en œuvre cinq utilitaires :
- le pré-processeur, que nous appellerons `cpp` ;
- 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`.

La figure1.4 donne un organigramme des différentes actions entreprises par un en-chaîneur de passes. Les fichiers temporaires de 1 à 3 décrits dans cette figure sont utilisés par les outils pour stocker les données intermédiaires nécessaires à la traduction des différents codes.

1.4.1 Pré-processeur

Le pré-processeur ou pré-compilateur 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 macro-expressions ;
- sélection de parties de texte.

1.4.2 Compilateur

Le compilateur lit ce que génère le pré-processeur 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

[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.

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

©Christian Bac 1985-2003

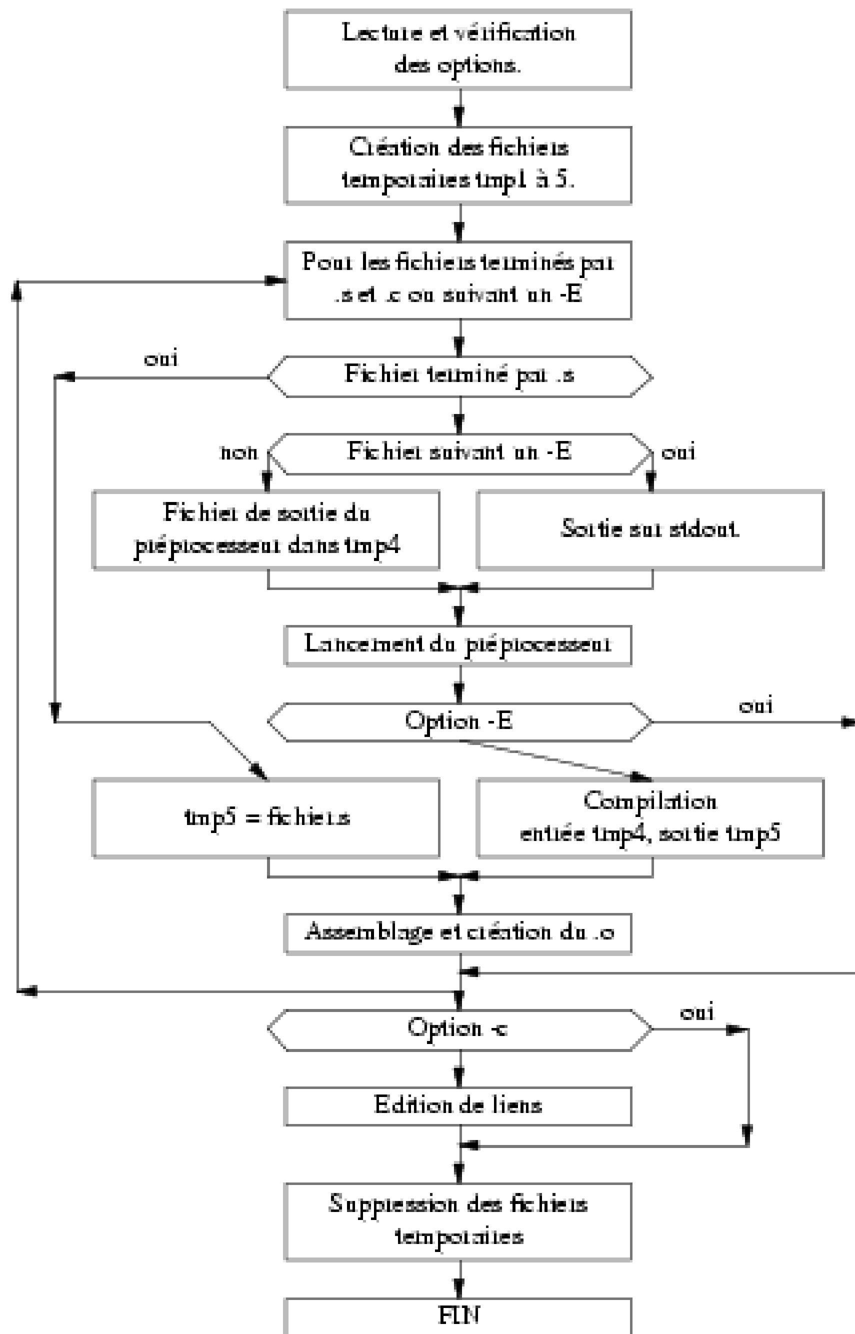

FIG. 1.4 – Schéma de fonctionnement de cc

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

©Christian Bac 1985-2003

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 Éditeur 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`.

1.4.6 Quelques options de cc

La figure1.4 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 pré-processeur 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 pré-processeur 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`

-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 ;

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

©Christian Bac 1985-2003

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

-Wall cette option augmente le nombre de messages d'alerte générés par le compilateur lorsqu'il rencontre des constructions dangereuses.

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

©Christian Bac 1985-2003

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 règles 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 pré-compilation (voir chapitre 15) commencent par un `#`.

2.1.1 Identifiant

Les identifiants sont utilisés pour donner des noms aux différentes entités utilisés dans le langage. Un identifiant 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 le souligné "_".

2.1.2 Espaces lexicaux

Les différents identifiants d'un programme en langage C sont classés dans des espaces lexicaux qui permettent d'isoler les noms selon leur signification. Les espaces lexicaux utilisés par un compilateur C sont les suivants :

- le premier espace contient les identifiants relatifs aux types synonymes, aux variables et aux 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'énumé-rations ;
- 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.

Ces espaces sont isolés ce qui permet d'avoir une étiquette qui porte le même nom qu'une variable mais jamais une fonction qui porte le même nom qu'une variable.

©Christian Bac 1985-2003

2.2 Mots réservés

Ce sont les mots prédéfinis du langage C. Ils ne peuvent pas être réutilisés pour des identifiants. Ils sont relatifs aux différents concepts du langage :

type des 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 anonymes. 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 : `{}`.

©Christian Bac 1985-2003

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. L'initialisation est traitée dans la section3.7.

3.1 Types de base

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

`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 qualificatifs pour préciser sa taille (`long` ou `short`), et le fait qu'il soit uniquement positif (`unsigned`) ou positif et négatif (`signed`) [1]. Le qualificatif `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. 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.

`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.

[1] Dans le cas le plus courant une variable du type entier peut contenir une valeur positive ou négative.

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 qualificatif `signed` permet d'insister sur le fait que la variable peut prendre des valeurs positives ou négatives. Il fait pendant au qualificatif `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 ce type de plus grande précision permet 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[2] et trente-deux bits sur les machines du type station de travail.

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 suivantes en nombre de bits : 16 (DEC PDP11, Intel 486), 32 (SUN Sparc, Intel Pentium) et 64 (DEC Alpha). Vous remarquerez que malgré son architecture interne de 64 bits, le compilateur pour alpha utilise des

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

©Christian Bac 1985-2003

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

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

entiers sur 32 bits. Il est aussi le seul processeur capable de différencier les `double` des `long doubles`.

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 identifiant.

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 (octal) lorsque le premier caractère est un `0` ou en base seize lorsque les deux premiers caractères sont `0X` (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 :

©Christian Bac 1985-2003

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 "`+`" un-aire.

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 de type caractère

Les constantes du type caractère simple sont toujours entourées d'apostrophes (*single quote*). 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 simplement entouré d'apostrophes, 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.
   '$\backslash 0$' : octet du nul, il sert à délimiter les fins de chaînes de caractères.
   '$\backslash 012$' : saut de ligne (*Line Feed*, LF) ;
   '$\backslash 015$' : retour chariot (*Carriage Return*, CR) ;
   '$\backslash 011$' : tabulation horizontale (*Horizontal Tabulation*, HT) ;
3. de même, un caractère peut être représenté par sa notation en base seize.
   '$\backslash x0$' : caractère nul ;
   '$\backslash xA$' : saut de ligne (LF) ;
   '$\backslash xC$' : retour chariot (CR) ;
   '$\backslash x9$' : tabulation horizontale (HT) ;
4. un certain nombre d'abréviations est aussi disponible :
   '$\backslash a$' : alert (sonnerie, BEL) ;
   '$\backslash b$' : backspace (BS) ;
   '$\backslash f$' : saut de page (FF) ;
   '$\backslash n$' : saut de ligne (LF) ;
   '$\backslash r$' : retour chariot (CR) ;
   '$\backslash t$' : tabulation horizontale (HT) ;
   '$\backslash v$' : 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".

FIG. 3.1 – Chaîne de caractères constante

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 d'autres qualificatifs qui ont été spécifiés par la norme. Il s'agit de `const`, `volatile`, `static` et `register`.

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.

Les qualificatifs `static` et `register` sont décrits dans la section 3.5.

3.4 Taille et normalisation

Les tailles 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 pré-compilateur et donnent les tailles et valeurs maximales des types de base et des types non signés.

Le programme 3.1 est un exemple de fichier <limits.h> et le programme 3.2 est un exemple de fichier <float.h>.

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>.

©Christian Bac 1985-2003

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

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
```

3.5 Définition de variables

Nous appellerons **identifiant** soit un nom de fonction, soit un nom de variable. Pour compléter ce que nous avons dit dans l'introduction de ce chapitre, voici comment les différents besoins associés à la **définition** de variables sont couverts par le langage C :

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

réservation de l'espace mémoire nécessaire au support de la variable lors de l'exécution ;
$\Longrightarrow$ grâce au type et à la classe mémoire.

initialisation de la variable à l'aide d'une constante dont le type correspond à celui de la variable ;
$\Longrightarrow$ en faisant suivre le nom par un symbole d'affectation = et une valeur compatible avec la variable.

association d'une durée de vie à la variable qui permet l'utilisation dans certaines parties du programme (règles de visibilité).
$\Longrightarrow$ grâce à la classe mémoire et au lieu de définition.

©Christian Bac 1985-2003

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
```

Programme 3.3 Exemple de fichier `stddef.h`

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

Une définition de variable est l'association d'un identifiant 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 possibilités associées aux classes mémoire dans le chapitre 9 sur la visibilité. Les **classes mémoire** sont :

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

`local` ou `auto` : cette classe comprend l'ensemble des variables définies dans un bloc. C'est le cas de toute variable définie 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 qualificatif 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).

©Christian Bac 1985-2003

Déclaration/définition	Classe mémoire
`int a;`	définition d'une variable globale
`int`	
`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
`int`	(rien à voir la variable b 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

`extern` ce qualificatif permet de spécifier que la ligne correspondante n'est pas une tentative de définition mais une **déclaration** (voir 3.9). 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 qualificatif 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 qualifiées `register`.

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

3.6 Types dérivés des types de base

Un type dérivé est créé à partir des types de base vus précédemment 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.

©Christian Bac 1985-2003

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'étoile (symbole de la multiplication `*`) qui précéde son nom dans sa définition.

Tout 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[3] valeurs qui sont des adresses de variables du type entier (`int`).

De même, `ptchar` est une variable du type pointeur sur un caractère. Elle peut donc contenir des valeurs qui sont des adresses de variables de type caractère (`char`).

Le compilateur C vérifie le type des adresses mises dans un pointeur. Le type du pointeur conditionne les opérations arithmétiques (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 définit les variables de la manière suivante :

```
int  in;
int  tabint[10];
char car;
int  *ptint;
char *ptchar;
```

Un pointeur peut être affecté avec l'adresse d'une variable ayant un type qui correspond à celui associé au pointeur. Comme nous le verrons dans la partie sur les opérateurs, le é-commercial `&` donne l'adresse du

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

©Christian Bac 1985-2003

nom de variable qui le suit et les noms de tableau correspondent à l'adresse du premier élément du tableau. Les pointeurs précédents peuvent donc être affectés de la manière suivante :

```
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 minuscule" dans l'entier `car`.

Il est possible de réaliser ces opérations en utilisant le pointeur pour accéder aux éléments du tableau. Ainsi les lignes :

```
ptint=tab;
*ptint=4;
```

affectent le pointeur `ptint` avec l'adresse du premier élément du tableau `tabint` équivalent (comme nous le reverrons dans le chapitre sur pointeurs et tableaux 10) à `&tabint[0]` ; puis le premier élément du tableau (`tabint[0]`) est affecté avec la valeur 4.

3.7 Initialisation de variables

L'initialisation se définit comme l'affectation d'une valeur lors de la définition de la variable.

Toute modification de valeur d'une variable postérieure à sa définition n'est pas une initialisation mais une affectation.

Par défaut, les variables de type global (définies en dehors de toute fonction) 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 initialiser pour vous prémunir contre le risque d'utiliser une variable sans en connaître la valeur initiale. Sur les compilateurs anciens, pour les variables de type tableau ou structure, seules les variables globales pouvaient être initialisées.

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

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. Les caractères de la chaîne sont considérés comme constants.

Le tableau 3.3 donne des exemples d'initialisation.

©Christian Bac 1985-2003

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.
const char *p3 = "Bonjour les";	Pointeur sur un caractère initialisé à l’adresse de la chaîne de caractères constante. La chaîne peut être mise dans une zone de mémoire accessible en lecture seulement.

TAB. 3.3 – Exemples d’initialisations

©Christian Bac 1985-2003

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 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 l'encart 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.

La notion de richesse d'un type est précisée dans la norme [ISO89]. 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`.

3.8.1 Conversions implicites

Le C selon la rédaction de Kernighan et Ritchie décrivait la façon avec laquelle les variables étaient promues lors des appels de fonction. Cette règle porte d'une part sur les nombres à virgule flottante qui sont transfomés en `double`, et d'autre part sur tous les types entier dont la taille est plus petite que l'entier naturel. Les variables ou les constantes des types suivants sont utilisées dans une expression, 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 tableau3.4.

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 :

(type) expression

Prenons pour exemple l'expression : `i = (int) f + (int) d ;`

©Christian Bac 1985-2003

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

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 et 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 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, les noms des arguments peuvent être omis),

– **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".

3.10 Exercices sur les types et variables

3.10.1 Exercice 1

Dans un fichier appelé `exo1.c`, déclarer les variables suivantes :
– chaîne de 10 caractères globale et statique ;
– tableau de 10 entiers global ;
– pointeur sur caractère local en registre ;
– flottant local ;
– caractère local statique ;
– entier nommé ex externe.
Dans la fonction `main()`, faire écrire les adresses des variables.

Compiler avec : `gcc -c exo1.c`

Dans un deuxième fichier appelé `exo1bis.c` mettre la définition de l'entier `ex` que vous initialiserez avec la valeur 20.

Compiler ce deuxième fichier de la même manière : `gcc -c exo1bis.c`

Faire l'édition de liens par : `gcc -o exo1 exo1.o exo1bis.o`

©Christian Bac 1985-2003

Programme 3.4 Suggestion de corrigé chapitre 3 exercice 1

```
 1 static char chaine[10];
 2 int tableau[10];
 3 extern int ex;
 4 int
 5 main (int argc, char *argv[], char **envp){
 6   register char *pointeur;
 7   float local;
 8   static char car;
 9   return 0;
10 }
```

Programme 3.5 Suggestion de corrigé chapitre 3 exercice 1 second fichier

```
 1
 2 int ex = 20;
```

3.10.2 Exercice 2

Dans un fichier appelé `exo2.c`, déclarer les variables suivantes :

- un entier global initialisé à 10 ;
- un tableau de 10 caractères global initialisé à `"bonjour"` ;
- un pointeur sur caractère initialisé à l'adresse du premier élément du tableau.

Compiler, essayer de mettre l'adresse de l'entier dans le pointeur de caractère et regardez les messages d'erreurs.

Une fois que vous aurez lu le chapitre 4 sur les éléments de bases pour réaliser les premiers programmes, faire écrire le tableau à partir de son nom et avec le pointeur.

PROGRAMME 3.6 SUGGESTION DE CORRIGÉ CHAPITRE 3 EXERCICE 2

```
 1 /* declaration des variables globales */
 2 int global = 10;
 3 char tableau[10] = "bonjour";
 4 int
 5 main (int argc, char *argv[], char **envp){
 6   char *pt = tableau;   /* ou pt = & tableau[0] */
 7   pt = (char *) &global;  /* adresse de l entier dans le pointeur de char */
 8   pt = tableau;           /* adresse du tableau dans le pointeur */
 9   printf("%s\n", tableau);
10   printf("%s\n", pt);
11   return 0;
12 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
bonjour
bonjour
```

©Christian Bac 1985-2003

3.10.3 Exercice 3

Déclarer :
- un entier `i` ;
- un flottant `f` ;
- un pointeur sur caractère `ptc` ;
- un pointeur sur entier `pti` ;
- un tableau d'entier `ti` ;
- un tableau de caractères `tc`.

Réaliser les opérations suivantes :
- `i = f;`
- `f = i;`
- `pti = 0;`
- `ptc = pti;`
- `pti = tc;`
- `ptc = ti`.

Compiler.

Faire les "casts" nécessaires à une compilation sans erreurs.

Programme 3.7 Suggestion de corrigé chapitre 3 exercice 3

```
 1 int
 2 main (int argc, char *argv[], char **envp){
 3   /* declarations des variables */
 4   int i;
 5   float f;
 6   char *ptc;
 7   int *pti;
 8   int ti[10];
 9   char tc[10];
10   /* affectations des variables */
11   i = f;
12   f = i;
13   pti = 0;
14   ptc = (char *) pti;
15   pti = (int *) tc;
16   ptc = (char *) ti;
17   return 0;
18 }
```

3.10.4 Exercice 4

Définir les variables :
- `i` entier
- `f` flottant
- `l` long
- `c` caractère
- `tc` tableau de caractères.

en les initialisant de la manière suivante :
- `i` à la valeur hexadécimale 50 ;
- `f` a 3.14 ;
- `l` à la valeur octale 40 ;

– c à “z”;
– tc à "qwertyuiop".

Programme 3.8 Suggestion de corrigé chapitre 3 exercice 4

```
 1 int
 2 main (int argc, char *argv[], char **envp){
 3   /* declaration et initialisation des variables */
 4   int i = 50;
 5   float f = 3.1415927;
 6   long l = 040L;
 7   char c = 'z';
 8   static char tc[] = "qwertyuiop";
 9   return 0;
10 }
```

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.

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` ? Ce programme est donné dans la figure 4.1.

Il est constitué de la manière suivante :

– la première ligne est une directive pour le pré-processeur 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

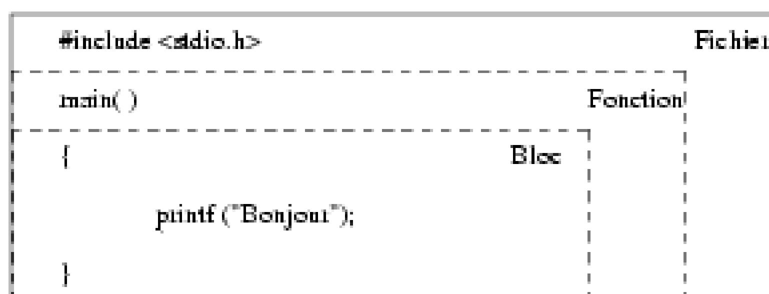

FIG. 4.1 – Programme qui écrit Bonjour

©Christian Bac 1985-2003

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 ;
– la fonction `main()` est terminée par une accolade fermante.

La fonction `printf()`[1] est une fonction[2] qui reçoit un nombre d'arguments variable. Ces arguments sont transformés en une chaîne de caractères. Cette transformation fait appel à une notion de format. Comme nous le verrons plus loin, le format définit le type de la donnée en mémoire et le mode de représentation de la donnée lorsqu'elle est affichée.

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()

```
 1 #include <stdio.h>
 2 char tt[80];       /* Tableau de 80 caracteres */
 3 int
 4 main (int argc, char *argv[])
 5 {
 6   printf ("ecrivez une chaine de caracteres : ");
 7   scanf ("%s", tt);
 8   printf ("\nLa chaine entree est : %s\n", tt);
 9   return 0;
10 }
```

DONNÉES EN ENTRÉE

```
bonjour
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
ecrivez une chaine de caracteres :
La chaine entree est : bonjour
```

Ce programme ne décrit toujours pas les arguments de `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 locale. Les seules instructions sont les appels aux fonctions de lecture et d'écriture (`scanf()` et `printf()`).

[1]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).

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

©Christian Bac 1985-2003

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

TAB. 4.1 – Conversions usuelles de printf et scanf

4.3 Quelques opérations

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

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

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 caractères 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 table4.1.

Dans une première approche de scanf(), nous considérons qu'il ne faut mettre que des types de conversion 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 parlons dans la section 5.1. Cet opérateur doit être mis devant le nom de la variable, dans les formats destinés à la fonction scanf, comme le montre le tableau 4.2, pour les variables dont nous avons déjà parlé sauf pour les variables du type tableau de caractères.

Les exemples de lecture d'une chaîne de caractères montrent que l'adresse du premier élément d'un tableau correspond au nom de tableau, nous en reparlerons dans le chapitre 10.

Le programme 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.

©Christian Bac 1985-2003

déclaration	**lecture**	**écriture**	**format externe**
`int i;`	`scanf("%d",&i);`	`printf("%d",i);`	décimal
`int i;`	`scanf("%o",&i);`	`printf("%o",i);`	octal
`int i;`	`scanf("%x",&i);`	`printf("%x",i);`	hexadécimal
`unsigned int i;`	`scanf("%u",&i);`	`printf("%u",i);`	décimal
`short j;`	`scanf("%hd",&j);`	`printf("%d",j);`	décimal
`short j;`	`scanf("%ho",&j);`	`printf("%o",j);`	octal
`short j;`	`scanf("%hx",&j);`	`printf("%x",j);`	hexadécimal
`unsigned short j;`	`scanf("%hu",&j);`	`printf("%u",j);`	décimal
`long k;`	`scanf("%ld",&k);`	`printf("%ld",k);`	décimal
`long k;`	`scanf("%lo",&k);`	`printf("%lo",k);`	octal
`long k;`	`scanf("%lx",&k);`	`printf("%lx",k);`	hexadécimal
`unsigned long k;`	`scanf("%lu",&k);`	`printf("%lu",k);`	décimal
`float l;`	`scanf("%f",&l);`	`printf("%f",l);`	point décimal
`float l;`	`scanf("%e",&l);`	`printf("%e",l);`	exponentielle
`float l;`		`printf("%g",l);`	la 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);`	`printf("%s",p);`	chaîne de caractères
	`scanf("%s",&p[0]);`		

TAB. 4.2 – Exemples de `printf` et `scanf`

PROGRAMME 4.2 LECTURES MULTIPLES AVEC SCANF()

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i = 10;
 6   float l = 3.14159;
 7   char p[50] = "Bonjour";
 8   printf ("%d bonjour %f %s\n", i, l, p);
 9   scanf ("%d%f%s", &i, &l, p);
10   printf ("Apres lecture au clavier : %d %f %s\n", i, l, p);
11   return 0;
12 }
```

DONNÉES EN ENTRÉE

```
23 6.55957 salut
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
10 bonjour 3.141590 Bonjour
Apres lecture au clavier : 23 6.559570 salut
```

4.5 Exercices sur `printf()` et `scanf()`

4.5.1 Exercice 1

Réaliser un programme dans un fichier contenant une fonction `main()` qui réalise les écritures suivantes :

- écrire le caractère 'a' ;
- écrire la chaîne de caractères "bonjour" ;
- écrire l'entier 32567 dans les formats :
  - décimal ;
  - hexadécimal ;
  - octal ;
  - non signé ;
- écrire le flottant 3.1415927 dans les formats suivants :
  - notation exponentielle ;
  - notation avec point décimal ;
  - variable (g).

4.5.2 Exercice 2

Reprendre l'exercice 1 en séparant chaque impression par un retour chariot.

4.5.3 Exercice 3

Déclarer des variables des types suivants :

- entier ;
- caractère ;
- flottant ;
- chaîne de caractères ;

©Christian Bac 1985-2003

PROGRAMME 4.3 SUGGESTION DE CORRIGÉ CHAPITRE 4 EXERCICE 1

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   /* ecriture de a bonjour 32567 32567 hexa 32567 octal 32567 non signe */
 5   printf("%c", 'a');
 6   printf("%s", "bonjour");
 7   printf("%d", 32567);
 8   printf("%x", 32567);
 9   printf("%o", 32567);
10   printf("%d", (unsigned) 32567);
11   /* ecriture de pi format e f g */
12   printf("%e", 3.1415927);
13   printf("%9.7f", 3.1415927);
14   printf("%g", 3.1415927);
15   return 0;
16 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
abonjour325677f3777467325673.141593e+003.14159273.14159
```

puis réaliser des opérations de lecture afin d'affecter ces variables.

4.5.4 Exercice 4

Lire et réécrire les éléments de l'exercice 3.

PROGRAMME 4.4 SUGGESTION DE CORRIGÉ CHAPITRE 4 EXERCICE 2

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   /* ecriture de a bonjour 32567 32567 hexa 32567 octal 32567 non signe */
 5   printf("%c\n", 'a');
 6   printf("%s\n", "bonjour");
 7   printf("%d\n", 32567);
 8   printf("%x\n", 32567);
 9   printf("%o\n", 32567);
10   printf("%d\n", (unsigned) 32567);
11   /* ecriture de pi au format e f g */
12   printf("%e\n", 3.1415927);
13   printf("%9.7f\n", 3.1415927);
14   printf("%g\n", 3.1415927);
15   return 0;
16 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
a
bonjour
32567
7f37
77467
32567
3.141593e+00
3.1415927
3.14159
```

©Christian Bac 1985-2003

PROGRAMME 4.5 SUGGESTION DE CORRIGÉ CHAPITRE 4 EXERCICE 3

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   /* declaration des variables */
 5   int evry;
 6   char dassaut;
 7   float ille;
 8   char io[100];
 9   /* saisie du nombre entier */
10   printf("entrer un entier\n");
11   scanf("%d", &evry);
12   /* saisie du caractere */
13   /* en effacant le caractere blanc ou \r ou \n precedent genant pour %c */
14   printf("entrer un caractere\n");
15   scanf(" %c", &dassaut);
16   /* saisie du nombre reel */
17   printf("entrer un reel\n");
18   scanf("%f", &ille);
19   /* saisie de la chaine de caracteres */
20   printf("entrer une chaine de caracteres\n");
21   scanf("%s", io);
22   return 0;
23 }
```

DONNÉES EN ENTRÉE

```
126 f 655957e-1 unseulmot possible
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer un entier
entrer un caractere
entrer un reel
entrer une chaine de caracteres
```

©Christian Bac 1985-2003

PROGRAMME 4.6 SUGGESTION DE CORRIGÉ CHAPITRE 4 EXERCICE 4

```
 1 #include <stdio.h>
 2 int
 3 main(int argc, char *argv[], char **envp){
 4   /* declaration des variables */
 5   int evry;
 6   char dassaut;
 7   float ille;
 8   char io[100];
 9   /* saisie du nombre entier */
10   printf("entrer un entier\n");
11   scanf("%d", &evry);
12   /* saisie du caractere */
13   /* en effacant le caractere blanc (\r ou \n) precedent genant pour %c */
14   printf("entrer un caractere\n");
15   scanf(" %c", &dassaut);
16   /* saisie du nombre reel */
17   printf("entrer un reel\n");
18   scanf("%f", &ille);
19   /* saisie de la chaine de caracteres */
20   printf("entrer une chaine de caracteres\n");
21   scanf("%s", io);
22   /* impression des resultats */
23   printf("%d\n%c\n%f\n%s\n", evry, dassaut, ille, io);
24   return 0;
25 }
```

DONNÉES EN ENTRÉE

```
126 f 655957e-1 unseulmot possible
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer un entier
entrer un caractere
entrer un reel
entrer une chaine de caracteres
126
f
65595.703125
unseulmot
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

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 les langages qui l'ont précédé (par exemple FORTRAN, ADA) 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 classification 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 un-aires

Un opérateur un-aire agit sur un opérande qui peut être une constante, une variable, ou une expression. Ainsi, l'opérateur un-aire `-` 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 un-aires.

Opérateur	Utilisation
`&`	opérateur d'adresse appelé aussi de référencement
`*`	opérateur d'indirection ou de déréférencement sur une adresse
`--`	opérateur de décrémentation
`++`	opérateur d'incrémentation
`sizeof`	opérateur donnant la taille en octet
`!`	négation logique
`-`	moins unaire, inversion du signe
`+`	plus unaire
`~`	complément à un

TAB. 5.1 – Liste des opérateurs unaires

©Christian Bac 1985-2003

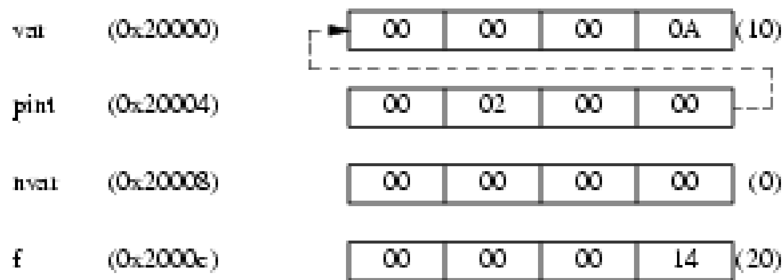

FIG. 5.1 – Exemple de relation entre pointeur et variable

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

Programme 5.1 Définitions de variables et d'un pointeur

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

5.1.1 Opérateur de référencement

Le `&` est l'opérateur de référencement, il retourne l'adresse en mémoir (référence) de la variable dont le nom le suit. `&var` donne l'adresse en mémoire 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). Comme le montre l'extrait de l'exemple : `pint = &var ;`

5.1.2 Opérateur de déréférencement ou indirection

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

5.1.3 Utilisation des & et *

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'instruction `pint = &nvar ;` est traduite dans la figure5.2.

Une fois cette instruction 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` $\leftarrow$ `@ 0x20000`, soit `R0` $\leftarrow$ `10`);
2. soustraire 2 à cette valeur (`R0` $\leftarrow$ `R0 -2`,
   soit `R0` $\leftarrow$ `8`);

[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.

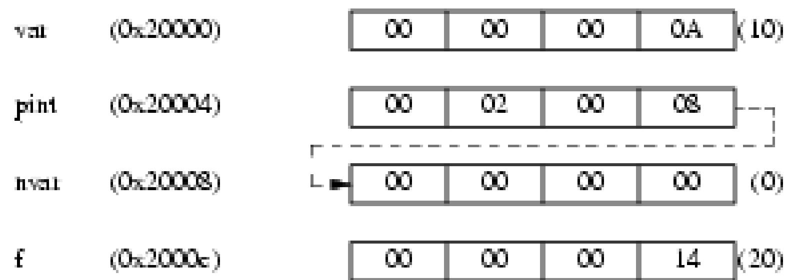

FIG. 5.2 – Mise en relation d'un pointeur et d'une variable

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 Opérateurs d'incrémentation et de décrémentation

Les opérateurs `--` et `++` permettent de décrémenter et d'incrémenter des variables de type entier. Dans une première approche, nous pouvons considérer que :

`var-` est équivalent à `var = var - 1`, ce qui implique en partant d'une variable `var` ayant une valeur de `8` que cette variable contient la valeur `7` une fois l'expression calculée à l'exécution ;

`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` à la fin de l'exécution de l'expression.

Il est possible d'utiliser les opérateurs un-aires d'incrémentation et de décrémentation derrière la variable (postfixé) ou devant celle-ci variable (préfixé). Ce qui permet de post-incrémenter, de pré-incrémenter, de post-décrémenter ou de pré-décrémenter.

Lorsque l'opérateur est préfixé, l'opération est appliquée avant que la valeur correspondant à l'opération ne soit calculée. Dans le cas où l'opération est post-fixée, la valeur de la variable avant l'opération est utilisée pour les autres calculs et ensuite l'opération est appliquée.

Prenons comme exemple, deux entiers `i` et `j`, et initialisons ces deux variables avec la valeur `0`.

```
int i=0, j=0;
```

Si nous écrivons `j = ++i`, c'est une pré-incrémentation de la variable `i`. Cela signifie incrémenter `i` de `1` puis mettre la valeur de `i` dans `j`. À la fin de cette opération `i` vaut `1` et `j` vaut `1`. En anticipant sur l'opérateur de succession 5.2.5, nous pouvons considérer cette instruction 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 instruction `i` vaut `1` et `j` vaut `0`. Cette opération est équivalente à :

```
j=i,
i = i+1,
j
```

5.1.5 Opérateur de taille

L'opérateur `sizeof` donne la taille en octets de la variable dont le nom suit. En gardant les définitions de variables du programme 5.1 : `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 (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 synonyme, comme décrit dans le chapitre 14, il est défini dans le fichier `<stddef.h>`.

5.1.6 Opérateur de négation logique

La négation logique (non logique, logical not) sert à inverser une condition en la faisant passer 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 un-aire inverse le signe de l'expression qui le suit. Le plus un-aire a été introduit par la norme ; il n'existait pas dans les versions C 72. Il sert à ne pas changer 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 l'opérande associé.

Si nous gardons l'exemple de notre variable de type entier `var` contenant la valeur `10`. En considérant qu'un entier est représenté sur 16 bits, cette valeur est équivalente en binaire à :
`0000 0000 0000 0000 0000 0000 0000 1010`
et la valeur `~var` est équivalente en binaire à :
`1111 1111 1111 1111 1111 1111 1111 0101`
soit la valeur décimale `-11`.

Comme son nom, complément à 1, 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 en référence au complément à 1, l'inversion de signe est aussi appelée complément à 2 puisque `-var` est équivalent à `~var +1`, soit le complément à 1 de la valeur de var plus 1.

5.2 Opérateurs binaires

Le tableau 5.2 donne la liste des opérateurs binaires.

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

©Christian Bac 1985-2003

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

5.2.1 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 un-aires, 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 entiers qui lui sont associés. Par exemple, l'expression `14 % 3` donne la valeur `2`.

5.2.2 Opérateurs manipulant les bits

Les opérateurs de masquage et de décalages sont utilisés pour manipuler les bitsdes variables entières. Ils 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 de niveau bit 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.

Les opérateurs qui permettent de manipuler des expressions au niveau bit sont de deux types ceux qui permettent d'extraire une partie des informations contenue dans un opérande (opérateurs de masquage) et ceux qui modifient l'information en déplaçant les bits vers la droite ou vers la gauche (opérateurs de décalage).

[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.

©Christian Bac 1985-2003

Opérateurs de masquage

Les trois premiers opérateurs binaires de niveau bit \& | ^) servent à sélectionner une partie d'une valeur ou bien à forcer certains bits à un.

Pour reprendre l'exemple des droits d'accès sur un fichier dans le système UNIX, considérons que le type de fichier 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 le programme 5.2, le mode initial est fixé à `0100751` sachant que la partie haute (à gauche dans la valeur) 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 :

1. fichier normal `010` dans la partie gauche de la valeur ;
2. pas de traitement particulier (bits sst à 0) ;
3. lecture écriture et exécutable pour le propriétaire (`7` ou `111`) ;
4. lecture et exécutable pour le groupe (`5` ou `101`) ;
5. exécutable pour les autres(`1` ou `001`).

Programme 5.2 Utilisation des opérateurs de masquage

```
1 int var=10, *pint=&var, nvar=0;
2 long f=20L;
3 char tabc[4]="abc";
```

Voici l'explication des différentes lignes du programme 5.2 :

– la ligne 3 met dans `res` la partie de `mode` qui spécifie le type de fichier, après cette instruction la variable `res` contient la valeur `0100000` ;
– la ligne 4 met dans `res` la valeur courante de `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.

5.2.3 Opérateurs de décalage

Les opérateurs de décalage (`<<` et `>>`) servent à manipuler les valeurs contenues dans les variables de type entier en poussant les bits vers la gauche ou la droite. 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'implantation (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.

Voici quelques explication sur les définitions et instructions du programme 5.3 :

©Christian Bac 1985-2003

PROGRAMME 5.3 UTILISATION DES OPÉRATEURS DE DÉCALAGE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   unsigned short int mode = 0100751;
 6   int res = mode << 4;
 7   printf ("res = %o\n", res);
 8   res = res >> 4;
 9   printf ("res = %o\n", res);
10   res = (mode >> 12) << 12;
11   printf ("res = %o\n", res);
12   return 0;
13 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
res = 2017220
res = 100751
res = 100000
```

- la ligne 2 met dans `res` la valeur correspondant à `mode` décalé à gauche de quatre positions, supprimant ainsi la partie qui spécifie le type de fichier. Le décalage vers la gauche a été réalisé en insérant des bits à la valeur zéro sur la droite de la valeur. La variable `res` contient à la fin de cette opération la valeur `017220` ;
- la ligne 3 remet dans `res` sa valeur courante décalée de quatre positions vers la droite, en ne conservant que les droits d'accès du fichier. Cette ligne met dans la variable `res` la valeur `000751`. Vous constatez que nous avons ainsi effacé la partie de la valeur qui correspond au type du fichier ;
- réciproquement la ligne 4 efface les bits correspondant aux droits d'accès en mettant 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`.

5.2.4 Opérateurs 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 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

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

TAB. 5.3 – Liste des opérateurs binaires d'affectation

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

bornes non strictes `12` et `143` il faut écrire : `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'`.

5.2.5 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.

5.2.6 Opérateurs 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 (=).

L'association d'un opérateur binaire avec l'opérateur d'affectation 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 que le type entier est représenté sur 32 bits, et que les deux variables `i` et `j` sont définies de la manière suivante : `int i = 100,j = 5;`.

©Christian Bac 1985-2003

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

TAB. 5.5 – Opérateur ternaire

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 et rend le programme difficile à lire. Ces possibilités complexes offertes en matière d'expressions sont de ce fait peu utilisées ou alors soigneusement encadrées par des parenthèses qui favorisent la compréhension.

5.2.7 Opérateur ternaire

L'opérateur ternaire met en jeu trois expressions (ex1,ex2,ex3) et permet de contruire une test avec retour de valeur.

La première expression (`ex1`) joue le rôle de condition. Si la condition est vérifiée, 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 `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 l'expression donne la valeur de `a` ;
2. sinon l'expression donne la valeur de `b` ;

Ainsi l'expression `max = a >= b ? a : b` permet de mémoriser dans la variable `max` la valeur maximum des deux valeurs contenues dans les variables `a` et `b`.

5.2.8 Précédence des opérateurs

Dans l'évaluation d'une ligne contenant des opérateurs et des opérandes, le compilateur utilise des ordres de priorité que nous appelons relations de **précédence**. Cette précédence détermine l'ordre d'association des opérateurs et opérande par le compilateur.

Ces régles de précédences sont représentées par le tableau 5.6, l'encart suivant donne la grille de lecture de ce tableau.

©Christian Bac 1985-2003

<table>
<tr><th>Classe d'opérateur</th><th>Opérateur(s)</th><th>Associativité</th></tr>
<tr><td>Parenthésage</td><td>( )</td><td>de gauche à droite</td></tr>
<tr><td>Appel de fonction
Suffixes ou</td><td>( )
[ ] -> .
++ --</td><td>de gauche à droite</td></tr>
<tr><td>Un-aires
préfixes</td><td>& * + - ~ !
++ -- sizeof sizeof( )</td><td>de droite à gauche</td></tr>
<tr><td>Changement de type</td><td>(type)</td><td>de droite à gauche</td></tr>
<tr><td>Multiplicatifs</td><td>* / %</td><td>de gauche à droite</td></tr>
<tr><td>Additifs</td><td>+ -</td><td>de gauche à droite</td></tr>
<tr><td>Décalages</td><td><< >></td><td>de gauche à droite</td></tr>
<tr><td>Comparaisons</td><td>< <= > >=</td><td>de gauche à droite</td></tr>
<tr><td>Égalités</td><td>== !=</td><td>de gauche à droite</td></tr>
<tr><td>et bit à bit</td><td>&</td><td>de gauche à droite</td></tr>
<tr><td>ou exclusif bit à bit</td><td>^</td><td>de gauche à droite</td></tr>
<tr><td>ou bit à bit</td><td>|</td><td>de gauche à droite</td></tr>
<tr><td>et logique</td><td>&&</td><td>de gauche à droite</td></tr>
<tr><td>ou logique</td><td>||</td><td>de gauche à droite</td></tr>
<tr><td>Condition</td><td>?:</td><td>de droite à gauche</td></tr>
<tr><td>Affectations</td><td>= += -= *= /= %=
&= |= ^=
<<= >>=</td><td>de droite à gauche</td></tr>
<tr><td>Succession</td><td>,</td><td>de gauche à droite</td></tr>
</table>

TAB. 5.6 – Précédence des opérateurs

RÈGLE 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 case 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), qui détermine dans quel ordre le compilateur associe, les opérateurs et les opérandes (en commençant par la droite ou la gauche).

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 :
- parenthésage ;
- opérateurs d'accès, appel de fonction et post incrémentation ou décrémentation ;
- opérateurs un-aires (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) ;
- opérateurs binaires d'affectation (associativité de droite à gauche) ;
- succession.

©Christian Bac 1985-2003

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

5.3 Expressions

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

Maintenant que nous avons vu l'ensemble des opérateurs du langage C, nous pouvons définir de manière rigoureuse dans l'encart ci dessus la notion d'expression.

Une expression est donc une suite syntaxiquement cohérente d'opérateurs et d'opérandes. Cette suite doit donc avoir des opérandes cohérents en nombre et en type avec les opérateurs.

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.

La table 5.3 décrit les relations entre expression et utilisation d'une expression dans un test.

Prenons la variable `i` qui est définie `int i=10;` l'expression `i` donne la valeur `10` . Cette valeur peut être utilisée dans un test et est considérée comme vraie dans ce cas.

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 (voir chapitre suivant),
- soit une expression suivie de " ;".

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

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

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

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

5.5 Exercices sur les opérateurs et les expressions

5.5.1 Exercice 1

Déclarer un entier, un pointeur sur entier et un tableau de 10 entiers.

©Christian Bac 1985-2003

1. Faire écrire l'adresse de l'entier.
2. Affecter la valeur 1 à l'entier .
3. Faire écrire la valeur de l'entier + 1.
4. Faire écrire la valeur de l'entier.

Mettre 1, 2, 3, 4 dans le tableau et affecter la valeur correspondant à l'adresse de début du tableau au pointeur.

1. En utilisant l'opérateur de post-incrémentation, faire écrire le contenu du pointeur et son incrément, ainsi que le contenu des entiers pointés par le pointeur et son incrément.
2. Faire écrire la taille de l'entier, du pointeur et du tableau.
3. Faire écrire la négation de ce qui est pointé par le pointeur.

5.5.2 Exercice 2

Définir deux entiers `i` et `j` initialisés avec les valeurs 10 et 3 respectivement.

Faire écrire les résultats de :
- `i + j`
- `i - j`
- `i * j`
- `i / j`
- `i % j`

Trouver une façon différente de calculer `i % j`. Affecter les valeurs 0x0FF et 0xF0F respectivement à `i et j`.

Faire écrire en hexadécimal les résultats de :
- `i & j`
- `i | j`
- `i ^ j`
- `i << 2`
- `j >> 2`

5.5.3 Exercice 3

Définir deux entiers `i` et `j` initialisés respectivement avec les valeurs 1 et 2.

Faire écrire les résultats des différentes opérations logiques entre `i` et `j` (égalité, inégalité, et, ou, etc.).

Noter les differences eventuelles entre les resultats de :
- `i && j` et `i & j`
- `i || j` et `i | j`

5.5.4 Exercice 4

En faisant écrire les résultats intermédiaires, réaliser les opérations suivantes sur un entier initialisé à 10 :
- division par 2
- addition avec 3
- multiplication par 2
- affectation du reste de sa propre division par 3
- ou avec 10
- décalage de 2 bits vers la gauche
- et avec 19
- ou exclusif avec 7.

NB : Il est possible de réaliser ces opérations de deux facons différentes.

©Christian Bac 1985-2003

5.5.5 Exercice 5 : Operateur ternaire

En utilisant 4 entiers `i`, `j`, `k` et `l`, avec `k` initialisé à 12 et `l` à 8, écrire le programme qui :
- lit les valeurs de `i` et `j`
- écrit la valeur de `k` si `i` est nul ;
- écrit la valeur de `i + l` `i` est non nul et `j` est nul
- écrit la valeur de `i + j` dans les autres cas.

5.5.6 Exercice 6

En utilisant quatre variables entières `i`, `j`, `k` et `l` :
- en une seule expression affecter la valeur 1 à `i`, `j` et `k`.
- ajouter 1 à `i` de quatre manières différentes.
- écrire l'expression qui affecte à `k` le resultat de`i + j +1` et incrémente `i` et `j` de 1 (quatre solutions au moins).
- écrire l'expression qui met :
  - `3 * i` dans `i`
  - `3 * i + j` dans `j`
  - divise `k` par 2 logiquement
  - divise la nouvelle valeur de `j` par celle de `k` et met dans `l` la puissance de 2 correspondant à la valeur obtenue.

©Christian Bac 1985-2003

PROGRAMME 5.4 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 1

```
 1 #include <stdio.h>
 2
 3 int i, *pti, tab[10];
 4 int
 5 main (int argc, char *argv[], char **envp){
 6   /* ecriture de l'adresse de l'entier */
 7   printf(" Adresse de l'entier : %x \n", &i);
 8   /* affectation de l'entier a 1 */
 9   i = 1;
10   /* ecriture de la valeur de l'entier + 1 */
11   printf(" Valeur de l'entier + 1 : %d\n", i + 1);
12   /* ecriture de a valeur de l'entier */
13   printf(" Valeur de l'entier : %d\n", i);
14   /* remplissage du tableau */
15   tab[0] = 1;  tab[1] = 2;
16   tab[2] = 3;  tab[3] = 4;
17   /* Affectation de l adresse de debut du tableau au pointeur */
18   pti = tab;
19   /* ecriture du contenu du pointeur et de son increment. */
20   printf(" Contenu du pointeur : %x \n", pti);
21   printf(" et de son increment : %x \n", pti + 1);
22   /* ecriture du contenu des objets pointes */
23   printf(" contenu de l'objet pointe par pt : %d \n", *pti);
24   printf(" et par l'increment de pt        : %d \n", *++pti);
25   /* ecriture des tailles de l'entier du pointeur et du tableau */
26   printf(" Taille de l'entier : %d octets \n", sizeof (i));
27   printf(" Taille du pointeur : %d octets \n", sizeof (pti));
28   printf(" Taille du tableau  : %d octets \n", sizeof (tab));
29   /* ecriture de la negation de ce qui est pointe par pt */
30   printf(" Negation de pt  : %d \n", !*pti);
31   return 0;
32 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Adresse de l'entier : 8049808
 Valeur de l'entier + 1 : 2
 Valeur de l'entier : 1
 Contenu du pointeur : 80497e0
 et de son increment : 80497e4
 contenu de l'objet pointe par pt : 1
 et par l'increment de pt         : 2
 Taille de l'entier : 4 octets
 Taille du pointeur : 4 octets
 Taille du tableau  : 40 octets
 Negation de pt  : 0
```

©Christian Bac 1985-2003

PROGRAMME 5.5 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 2

```
 1 #include <stdio.h>
 2 
 3 int
 4 main(int argc, char *argv[], char **envp){
 5   /* Definition des deux entiers initialises a 10 et 3 */
 6   int i = 10, j = 3;
 7   /* ecriture du resultat de differentes operations arithmetiques */
 8   printf(" Le resultat de i + j est %d \n", i + j);
 9   printf(" Le resultat de i - j est %d \n", i - j);
10   printf(" Le resultat de i * j est %d \n", i * j);
11   printf(" Le resultat de i / j est %d \n", i / j);
12   printf(" Le resultat de i %% j est %d \n", i % j);
13   /* facon differente de calculer i % j */
14   printf(" calcul different de i %% j : %d \n", i - (j * (i / j)));
15   /* nouvelle affectation de i et de j */
16   i = 0x0FF;
17   j = 0xF0F;
18   /* ecriture du resultat de differentes operations logiques */
19   printf(" Le resultat de i & j est %x \n", i & j);
20   printf(" Le resultat de i | j est %x \n", i | j);
21   printf(" Le resultat de i ^ j est %x \n", i ^ j);
22   printf(" Le resultat de i << 2 est %x \n", i << 2);
23   printf(" Le resultat de j >> 2 est %x \n", j >> 2);
24   return 0;
25 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Le resultat de i + j est 13
 Le resultat de i - j est 7
 Le resultat de i * j est 30
 Le resultat de i / j est 3
 Le resultat de i % j est 1
 calcul different de i % j : 1
 Le resultat de i & j est f
 Le resultat de i | j est fff
 Le resultat de i ^ j est ff0
 Le resultat de i << 2 est 3fc
 Le resultat de j >> 2 est 3c3
```

©Christian Bac 1985-2003

PROGRAMME 5.6 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 3

```
 1 #include <stdio.h>
 2 
 3 int i, *pti, tab[10];
 4 int
 5 main (int argc, char *argv[], char **envp){
 6   int i = 1, j = 2; /* initialisation de deux entiers */
 7   /* ecriture du resultat de differentes operations logiques */
 8   printf(" Resultat de diverses operations logiques avec i = 1 et j = 2\n");
 9   printf(" Le resultat de i & j est %x \n", i & j);
10   printf(" Le resultat de i | j est %x \n", i | j);
11   printf(" Le resultat de i ^ j est %x \n", i ^ j);
12   printf(" Le resultat de ~i est %x \n", ~i);
13   printf(" Le resultat de ~j  est %x \n", ~j);
14   printf(" Le resultat de i && j est %x \n", i && j);
15   printf(" Le resultat de i || j est %x \n", i || j);
16   printf(" Le resultat de !i  est %x \n", !i);
17   printf(" Le resultat de !j  est %x \n", !j);
18   return 0;
19 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Resultat de diverses operations logiques avec i = 1 et j = 2
Le resultat de i & j est 0
Le resultat de i | j est 3
Le resultat de i ^ j est 3
Le resultat de ~i est fffffffe
Le resultat de ~j  est fffffffd
Le resultat de i && j est 1
Le resultat de i || j est 1
Le resultat de !i  est 0
Le resultat de !j  est 0
```

©Christian Bac 1985-2003

PROGRAMME 5.7 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 4

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   int i = 10;
 5   printf (" Resultat de diverses operations avec i = %d\n\n", i);
 6   /* ecriture du resultat de differentes operations */
 7   printf(" Division par 2   %d\n", i /= 2);
 8   printf(" Addition avec 3  %d\n", i += 3);
 9   printf(" Multiplication par 2 %d\n", i *= 2);
10   printf(" Reste de la division par 3 %d\n", i %= 3);
11   printf(" OU logique avec 10 %d\n", i |= 10);
12   printf(" Shift de 2 a gauche  %d\n", i <<= 2);
13   printf(" ET logique avec 19 %d\n", i &= 19);
14   printf(" OU exclusif avec 7 %d\n", i ^= 7);
15   return 0;
16 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Resultat de diverses operations avec i = 10

Division par 2 5
Addition avec 3 8
Multiplication par 2 16
Reste de la division par 3 1
OU logique avec 10 11
Shift de 2 a gauche 44
ET logique avec 19 0
OU exclusif avec 7 7
```

PROGRAMME 5.8 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 5

```
 1 #include <stdio.h>
 2 int
 3 main(int argc, char *argv[], char **envp){
 4   int i, j, k = 12, l = 8;  /* definition de 4 entiers */
 5   /* lecture des valeurs de i et de j */
 6   printf("\n Entrer la valeur de i :");
 7   scanf("%d", &i);
 8   printf("\n Entrer la valeur de j :");
 9   scanf("%d", &j);
10   /* ecriture du resultat selon les valeurs de i et de j
11    *    si i est nul, impression de la valeur de k
12    *    si j est nul, impression de la valeur de i + l
13    *    impression de la valeur de i +j dans les autres cas
14    */
15   printf("\n resultat : %d\n", (!i ? k : (!j ? i + l : i + j)));
16   return 0;
17 }
```

DONNÉES EN ENTRÉE

```
126 0
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer la valeur de i :
 Entrer la valeur de j :
 resultat : 134
```

©Christian Bac 1985-2003

PROGRAMME 5.9 SUGGESTION DE CORRIGÉ CHAPITRE 5 EXERCICE 6

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   int i, j, k, l;   /* definition des 4 entiers */
 5   /* affectation des entiers */
 6   i = j = k = 1;
 7   /* incrementation de i de 4 manieres differentes */
 8   printf(" premiere solution i++ = %d \n", i++);
 9   printf(" deuxieme solution ++i = %d \n", ++i);
10   printf(" troisieme solution i = i + 1 = %d \n", i = i + 1);
11   printf(" quatrieme solution i += 1 = %d \n", i += 1);
12   /* affectation de i + j + 1 a k et incrementation de i et de j */
13   printf(" resultat apres affectation et incrementation\n");
14   k = i++ + j++ + 1;    /* ou i++ + ++j ou ++i+j++ ou ++i+ ++j -1 */
15   printf(" valeur de k : %d \n", k);
16   printf(" valeur de i : %d \n", i);
17   printf(" valeur de j : %d \n", j);
18   /* ecriture d'une expression qui realise les operations suivantes :
19    *  3 * i dans i et 3 * i + j dans j
20    *  et divise k par 2 logiquement et divise j par k
21    *  et affecte a l la puissance 2 correspondant à la valeur precedente
22    */
23   l = l << (j += (i *= 3)) / (k >>= l = 1);
24   printf(" Resultat apres calcul de l'expression \n");
25   printf(" valeur de i ......... %d \n", i);
26   printf(" valeur de j ......... %d \n", j);
27   printf(" valeur de k ......... %d \n", k);
28   printf(" valeur de l ......... %d \n", l);
29   return 0;
30 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
premiere solution i++ = 1
deuxieme solution ++i = 3
troisieme solution i = i + 1 = 4
quatrieme solution i += 1 = 5
resultat apres affectation et incrementation
valeur de k : 7
valeur de i : 6
valeur de j : 2
Resultat apres calcul de l'expression
valeur de i ......... 18
valeur de j ......... 20
valeur de k ......... 3
valeur de l ......... 64
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 6

Instructions de contrôle

Les instructions de contrôle servent à controler le déroulement de l'enchaînement des instructions à l'intérieur d'un programme, ces instructions peuvent être des instructions conditionnelles ou itératives.

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 l'encart suivant :

```
if( expression ) instruction

if( expression ) instruction1
else instruction2
```

L'expression n'est pas forcément un test qui retourne la valeur 0 ou +1. Ce peut être un calcul ou une affectation, car, comme nous l'avons déja dit dans la section5.1.6 , il n'y a pas de type booléen, et 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 (a == b)`	usuel comme dans tous les autres langages
`int b = 1;`	
`if (a = b)`	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 pour tester si une variable contient une valeur. Par exemple l'ensemble de tests décrit dans le programme 6.1 permet de tester si un nombre est pair ou impair et si il est supérieur à 5 ou non.

©Christian Bac 1985-2003

PROGRAMME 6.1 EXEMPLE DE TESTS IMBRIQUÉS

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i;
 6   scanf ("%d", &i);
 7   if (i == 6 || i == 8)
 8     {
 9       printf ("le nombre est superieur a 5\n");
10       printf ("le nombre est pair\n");
11     }
12   else if (i == 4 || i == 2 || i == 0)
13     printf ("le nombre est pair\n");
14   else if (i == 7 || i == 9)
15     {
16       printf ("le nombre est superieur a 5\n");
17       printf ("le nombre est impair\n");
18     }
19   else if (i == 5 || i == 3 || i == 1)
20     printf ("le nombre est impair\n");
21   else
22     printf ("ceci n est pas un nombre\n");
23   return 0;
24 }
```

DONNÉES EN ENTRÉE

```
5
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
le nombre est impair
```

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 l'encart suivant :

©Christian Bac 1985-2003

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
	}	

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

1. l'expression est évaluée comme une valeur entière ;
2. les valeurs des `case` sont évaluées comme des constantes entières ;
3. l'exécution se fait à partir du `case` dont la valeur correspond à l'expression. Elle s'exécute en séquence jusqu'à la rencontre d'une instruction `break;` ;
4. les instructions qui suivent la condition `default` sont exécutées lorsqu'aucune constante des **case** n'est égale à la valeur retournée par l'expression ;
5. l'ordre des `case` et du `default` n'est 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 6.2, 6.3 et 6.4 sont des exemples d'utilisation de tables de sauts.

Le premier exemple 6.2 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.

L'exemple 6.3 reprend l'exemple précédent 6.1 réalisé avec des tests imbriqués. L'éxécution se déroule donc de la même façon, en utilisant la valeur dans l'expression `switch`, c'est-à-dire la valeur de `i` :

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`.

Dans l'exemple 6.4, la sélection incrémente les variables `nb_chiffres` ou `nb_lettres` selon que le caractère lu au clavier est un chiffre, ou une lettre. Si le caractère lu n'est ni un chiffre ni une lettre, la variable `nb_others` est incrémentée.

©Christian Bac 1985-2003

while(expression) instruction

TAB. 6.1 – Syntaxe du `while`

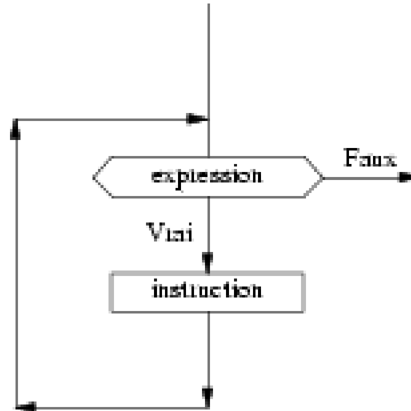

FIG. 6.1 – Organigramme du `while`

6.2 Instructions itératives

Les instructions itératives 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 6.1.

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

Dans cet exemple 6.5, 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. 6.7) que cet exemple peut être réécrit à l'aide d'une boucle `for` de manière à ne pas déborder du tableau.

L'exemple 6.6 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 fonctionne correctement car le compilateur a mis un octet nul (`'\0'`) à la fin de la chaîne `tab` (qui sans cela n'en serait pas une).

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

TAB. 6.2 – Comparaison du `for` et du `while`

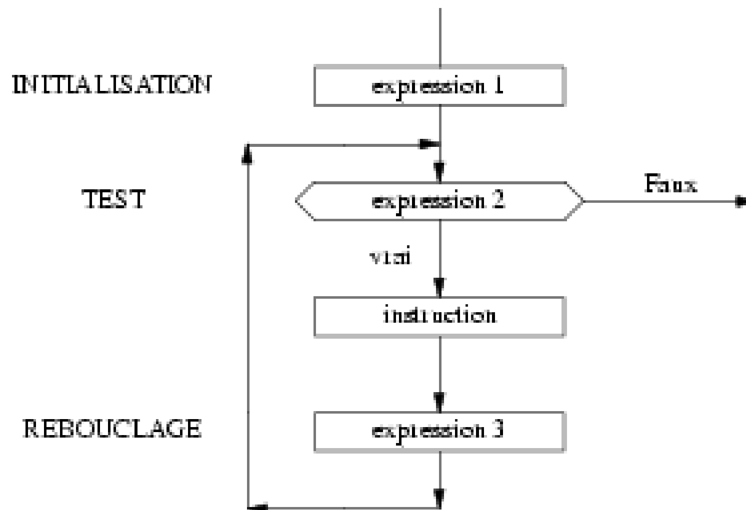

FIG. 6.2 – Organigramme du `for`

6.2.2 **`for`**

La syntaxe du `for` est donnée dans l'encart suivant.

for(exp1 ; exp2 ; exp3) instruction

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

1. l'expression `expr1` est réalisée une seule fois lors de l'entrée dans la boucle, nous l'appellerons expression d'initialisation ;
2. l'expression `expr2` est la condition d'exécution de l'instruction. Elle est testée à chaque itération, y compris la première. Si l'expression `expr2` prend la valeur vraie l'instruction controlée par le `for` est exécutée, sinon la boucle se termine ;
3. l'expression `expr3` contrôle l'avancement de la boucle. Elle permet de manière générale de calculer la prochaine valeur avec laquelle la condition de passage va être retestée, elle est exécutée après l'instruction à chaque itération avant le nouveau test de passage.

Le `for` est équivalent à un `while` plus une instruction d'initialisation et une instruction de contrôle comme il est montré dans le tableau 6.2.

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 < 10 ; i++)`
2. `for (i = 0 , j = 10 ; i < j ; i++ , j--)`

©Christian Bac 1985-2003

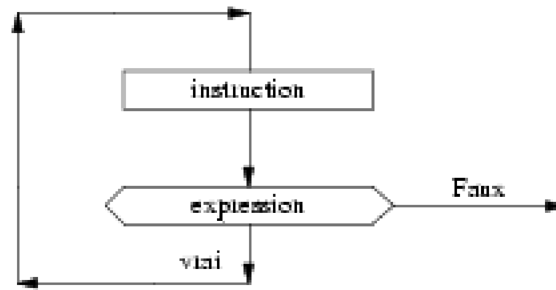

FIG. 6.3 – Organigramme du `do while`

3. `for ( ; ; ) inst`

Le premier exemple correspond à une boucle de parcours classique d'un tableau de taille 10. Dans cet exemple, l'expression d'initialisation met la valeur 0 dans la variable qui joue le rôle d'indice pour parcourrir le tableau ; la condition d'exécution se fait en testant si l'indice courant est strictement inférieur à la taille du tableau (10), et la progression d'indice se fait par pas de 1 en utilisant l'opérateur d'incrémentation (++).

Le deuxième exemple montre comment avoir plusieurs initialisations et plusieurs expressions dans l'expression d'avancement de la boucle, en utilisant l'opérateur de succession `","`.

Le troisième exemple est une convention pour écrire une boucle infinie. Ce type de boucle infinie est utilisé lorsque l'instruction qu'elle contrôle 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).

L'exemple 6.7 montre l'utilisation d'une boucle `for` pour la lecture d'un ensemble de caractères au clavier et l'affectation du tableau avec les caractères lus. L'expression de condition d'exécution réalise plusieurs actions. Elle commence par vérifier qu'il n'y a pas de risque de débordement du tableau (indice inférieur à la taille du tableau). Puis elle réalise la lecture d'un caractére au clavier qu'elle stocke dans la variable `c`. La valeur mise dans la variable `c` est alors testée pour vérifier que la fin de fichier n'a pas été atteinte (attention au parenthésage il est indispensable à la bonne exécution). Si les deux conditions sont réunies, la valeur de l'indice est inférieure à la taille du tableau et le caractère lu n'est `EOF`, le caractére lu est mis dans l'élément correspondant au rang contenu dans la variable `rang`.

6.2.3 `do while`

La syntaxe du `do while` est donnée dans l'encart suivant :

do instruction while(expression) ;

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 du **while** mais avec le test en fin plutôt qu'en début, ce qui assure que l'instruction est réalisée au moins une fois.

Dans l'exemple donné dans le programme 6.8, 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.

©Christian Bac 1985-2003

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 instructions itératives

Comme nous l'avons vu dans la section 5.4, 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 sont réalisées par quatre instructions 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`

L'instruction `continue` est utilisée en relation avec les boucles. Elle provoque le passage à l'itération suivante de la boucle en sautant à la fin du bloc. Ce faisant, elle provoque la non exécution des instructions qui la suivent à l'intérieur du bloc.

Prenons le programme 6.9, qui compte le nombre de caractères non blancs rentrés au clavier, et le nombre total de caractères. Les caractères sont considérés comme blancs s'ils sont égaux soit à l'espace, la tabulation horizontale, le saut de ligne ou le retour à la colonne de numéro zéro. À la fin de l'exécution :

1. i contient une valeur qui correspond au nombre total de caractères qui ont été tapés au clavier ;
2. j contient une valeur qui correspond au nombre de caractères non blancs ;
3. 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 d'un bloc d'instruction associé à une instruction répétitive ou alternative controlée par les instructions `if`, `for`, un `while` ou un `do while`. Il n'est pas aussi général qu'en ADA ou en JAVA puisqu'il ne permet de sortir que d'un niveau d'imbrication.

Dans l'exemple 6.10, nous reprenons l'exemple 6.9, de manière à créer une boucle qui compte le nombre de caractères jusqu'au retour chariot en utilisant l'instruction `break`. Lorsque le retour chariot

©Christian Bac 1985-2003

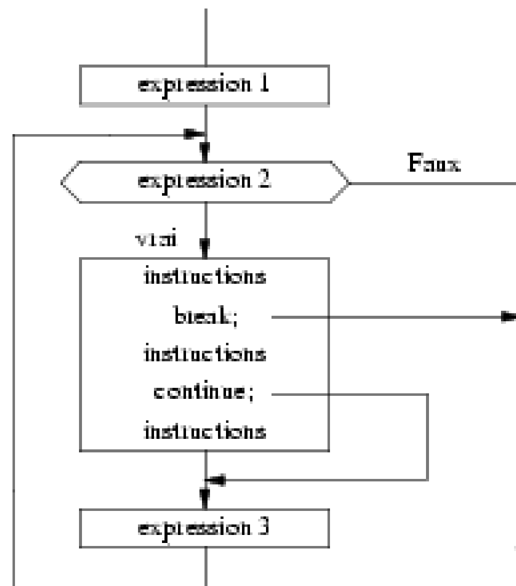

FIG. 6.4 – `break` et `continue` dans un `for`

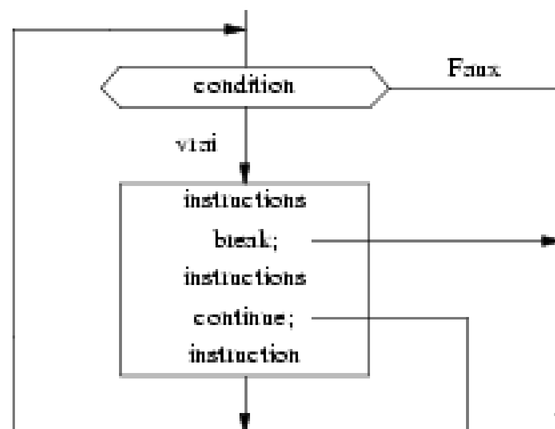

FIG. 6.5 – `break` et `continue` dans un `while`

('\r') est rencontré, le `break` provoque la sortie de la boucle `for`. Il en serait de même avec un `while` ou un `do while`.

Dans l'exemple 6.11, nous reprenons l'exemple 6.5, de manière à créer une boucle qui remplit le tableau en vérifiant qu'il n'y a pas de débordement de taille. Dans ce cas, nous utilisons le `break` pour sortir de la boucle lorsque la fin de fichier est rencontrée.

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 appelée label. Un label est une chaîne de caractères suivie du double point " : " .

©Christian Bac 1985-2003

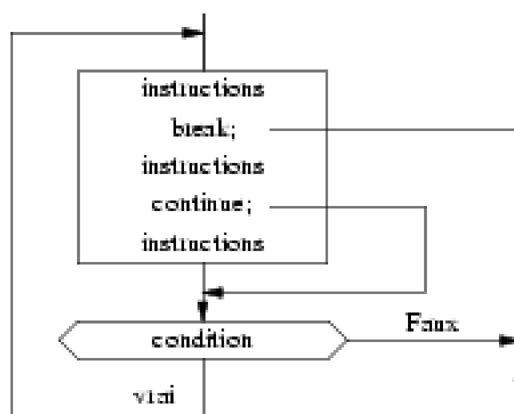

FIG. 6.6 – `break` et `continue` dans un `do while`

Le programme 6.12 est un extrait de la suite de tests du compilateur `gcc`. Dans ce programme :

`test_goto1()` doit retourner 2 si l'argument qui lui est passé est différent de 0 et 1 sinon.

`test_goto2()` doit retourner 8 si l'argument qui lui est passé est inférieur à 10 et 4 sinon.

`main()` provoque l'appel de la fonction `test_goto1()` successivement avec les valeurs 0 et 1 puis la fonction `test_goto2` avec une valeur inférieure à 10 (3) et une valeur supérieure à 10 (30). Les résultats obtenus par les retours d'appels de fonctions sont cumulés dans la variable globale `goto_val`.

6.3.4 `return`

Nous étudierons de manière plus détaillée les fonctions dans le chapitre 8. 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` permet de calculer une valeur correspondant au type de sa fonciton. Ce calcul se fait en évaluant l'expression qui suit le `return`. L'expression est évaluée et la valeur calculée est retournée à la fonction appelante.

Lorsque la fonction est de type `void`, le `return` est utilisé sans expression associée, dans le cas où la fonction est associé à un type qui n'est pas `void`, la valeur calculée par le return est celle que retourne la fonction contenant le `return`. Les formes possibles du **return** sont donc :
- `return ;`
- `return expression ;`

Il peut y avoir plusieurs `return` dans une fonction, le premier qui est exécuté dans le contexte de la fonction provoque : le calcul de la valeur à retrouner, la sortie de la fonction, la continuation de l'exécution de la fonction appelante.

Nous allons créer une fonction (voir prog. 6.13) 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.

6.4 Exercices sur les instructions de contrôle

6.4.1 Exercice 1

Lire 2 entiers `i` et `j`.

Écrire celui possédant la plus grande valeur absolue
- avec l'instruction `if`
- avec l'opérateur ternaire.

6.4.2 Exercice 2

Écrire un programme qui imprime le type de lettre(voyelle ou consonne) entrée au clavier.

Remarque : il peut être judicieux de transformer les majuscules en minuscules.

6.4.3 Exercice 3

A l'aide d'une boucle `for`, lire 10 entiers et les stocker dans un tableau.

Réécrire le contenu du tableau à l'aide d'une boucle `while`.

6.4.4 Exercice 4

En utilisant une boucle `do while`, transformer l'exercice 2 de façon à arréter le traitement lorsqu'un caractère autre qu'une lettre est entré au clavier.

6.5 Exercices sur les ruptures de séquence

6.5.1 Exercice 5

Lire caractère par caractère une ligne entrée au clavier (la fin de ligne est caracterisée par un retour chariot) et la stocker dans un tableau.

Transformer cette ligne en remplacant une suite d'espaces ou une tabulation par un espace unique, et en remplacant le retour chariot de fin de ligne par le caractère nul(`'\0'`).

Réécrire la ligne ainsi transformée par un `printf()` de chaîne de caractères.

6.5.2 Exercice 6

Cet exercice consiste en la lecture d'un entier, qui est le rang d'une lettre dans l'alphabet, et en l'écriture de la lettre correspondant à ce rang.

Pour cela il faudra :
- initialiser un tableau de caractères avec l'alphabet ;
- faire une boucle de lecture d'un entier ;
- sortir de la boucle lorsque l'entier lu est 100 ;
- imprimer la lettre correspondant à l'entier lorsque celui-ci est compris entre 1 et 26 ;
- dans les autres cas reboucler en lecture.

Programme 6.2 Exemple de table de sauts

```
 1 /* Extracted from GCC source code: this code is copyrighted see GCC source
 2  * code for copyright license
 3  */
 4 extern void ERROR (char *, ...);
 5 extern void warning (char *, ...);
 6 extern void pedwarn (char *, ...);
 7 extern int fflag, lflag, imag, pedantic;
 8 void
 9 exemple (char *p)
10 {
11   switch (*p++)
12     {
13     case 'f':
14     case 'F':
15       if (fflag)
16   ERROR ("more than one 'f' suffix on floating constant");
17       fflag = 1;
18       break;
19
20     case 'l':
21     case 'L':
22       if (lflag)
23   ERROR ("more than one 'l' suffix on floating constant");
24       lflag = 1;
25       break;
26
27     case 'i':
28     case 'I':
29     case 'j':
30     case 'J':
31       if (imag)
32   ERROR ("more than one 'i' or 'j' suffix on floating constant");
33       else if (pedantic)
34   pedwarn ("ISO C forbids imaginary numeric constants");
35       imag = 1;
36       break;
37
38     default:
39       ERROR ("invalid suffix on floating constant");
40     }
41 }
```

©Christian Bac 1985-2003

PROGRAMME 6.3 EXEMPLE DE TABLE DE SAUTS

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i;
 6   scanf ("%d", &i);
 7   switch (i)
 8     {
 9     case 6:
10     case 8:
11       printf ("le nombre est superieur a 5\n");
12     case 0:
13     case 2:
14     case 4:
15       printf ("le nombre est pair\n");
16       break;
17     default:
18       printf ("ceci n est pas un nombre\n");
19       break;
20     case 9:
21     case 7:
22       printf ("le nombre est superieur a 5\n");
23     case 5:
24     case 1:
25     case 3:
26       printf ("le nombre est impair\n");
27       break;
28     }
29   return 0;
30 }
```

DONNÉES EN ENTRÉE

```
5
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
le nombre est impair
```

©Christian Bac 1985-2003

PROGRAMME 6.4 EXEMPLE DE TABLE DE SAUTS

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int c, nb_chiffres = 0, nb_lettres = 0, nb_autres = 0;
 6   while ((c = getchar ()) != EOF)
 7     {
 8       switch (c)
 9         {
10         case '0': case '1': case '2': case '3': case '4':
11         case '5': case '6': case '7': case '8': case '9':
12           nb_chiffres++;
13           break;
14         case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
15         case 'g': case 'h': case 'i': case 'j': case 'k':
16         case 'l':         /* toutes les valeurs des */
17         case 'm':         /* lettres doivent etre */
18         case 'n':         /* presentes dans les case */
19         case 'o': case 'p': case 'q': case 'r': case 's': case 't':
20         case 'u': case 'v': case 'w': case 'x': case 'y': case 'z':
21           nb_lettres++;
22           break;
23         default:
24           nb_autres++;
25           break;
26         }
27     }
28   printf ("Chiffres lettres et autres sur le fichier d'entree : %d %d %d\n",
29           nb_chiffres, nb_lettres, nb_autres);
30   return 0;
31 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Chiffres lettres et autres sur le fichier d'entree : 10 35 13
```

©Christian Bac 1985-2003

PROGRAMME 6.5 LECTURE D'UNE LIGNE AVEC `while`

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   char tab[80];
 6   int c, rang = 0;
 7   while ((c = getchar ()) != EOF)
 8     tab[rang++] = c;
 9   tab[rang]='\0';
10   printf("Caracteres lus : %s\n",tab);
11   return 0;
12 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Caracteres lus : The quick brown fox jumped over the lazy dog, 1234567890.
```

PROGRAMME 6.6 RECOPIE D'UNE CHAÎNE AVEC UNE BOUCLE `while`

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   char tab[] = "Message initialise";
 6   char tab2[50];
 7   int i = 0;
 8   while (tab[i])
 9     {
10       tab2[i] = tab[i];
11       i++;
12     }
13   tab2[i] = '\0';
14   printf ("Resultat de la copie de tab : %s \n", tab);
15   printf ("dans tab2 : %s \n", tab2);
16   return 0;
17 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Resultat de la copie de tab : Message initialise
dans tab2 : Message initialise
```

©Christian Bac 1985-2003

PROGRAMME 6.7 LECTURE D'UNE LIGNE AVEC FOR

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   char tab[80];
 6   int rang, c;
 7   for (rang = 0; rang < 80 && (c = getchar ()) != EOF; rang++)
 8     tab[rang] = c;
 9   tab[rang]='\0';
10   printf ("Contenu du tableau tab apres execution :\n\t%s\n", tab);
11   return 0;
12 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Contenu du tableau tab apres execution :
The quick brown fox jumped over the lazy dog, 1234567890.
```

PROGRAMME 6.8 DÉCOMPOSITION DES PUISSANCES DE DIX D'UN NOMBRE AVEC UN DO WHILE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int s[10], i = 0, j=0, n = 5634;
 6   do
 7     s[i++] = n % 10;
 8   while ((n /= 10) > 10);
 9   printf("Valeurs des entiers dans s : ");
10   for(j=0;j<i;j++){
11     printf("%d ",s[j]);
12   }
13   printf("\n");
14   return 0;
15 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Valeurs des entiers dans s : 4 3 6
```

©Christian Bac 1985-2003

PROGRAMME 6.9 UTILISATION DU CONTINUE DANS UNE BOUCLE FOR

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j, c;
 6   for (i = 0, j = 0; (c = getchar ()) != EOF; i++)
 7     {
 8       if (c == ' ')
 9   continue;
10       if (c == '\t')
11   continue;
12       if (c == '\r')
13   continue;
14       if (c == '\n')
15   continue;
16       j++;
17     }
18   printf ("Caracteres lus : %d dont non blancs %d \n", i, j);
19   return 0;
20 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Caracteres lus : 58 dont non blancs 48
```

PROGRAMME 6.10 UTILISATION DES RUPTURES DE SÉQUENCE DANS UNE BOUCLE FOR

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j, c;
 6   for (i = j = 0; (c = getchar ()) != EOF; i++)
 7     {
 8       if (c == '\r')
 9   break;
10       if (c == ' ')
11   continue;
12       j++;
13     }
14   printf ("Caracteres lus : %d dont non blancs %d \n", i, j);
15   return 0;
16 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Caracteres lus : 58 dont non blancs 49
```

©Christian Bac 1985-2003

PROGRAMME 6.11 LECTURE D'UNE LIGNE AVEC FOR ET BREAK

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   char tab[80];
 6   int c, rang;
 7   for (rang = 0; rang < 80; rang++)
 8     {
 9       if ((c = getchar ()) != EOF)
10   tab[rang] = c;
11       else
12   break;
13     }
14   tab[rang]='\0';
15   printf("Ensemble de caracteres lus :\n %s\n",tab);
16   return 0;
17 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Ensemble de caracteres lus :
 The quick brown fox jumped over the lazy dog, 1234567890.
```

©Christian Bac 1985-2003

PROGRAMME 6.12 UTILISATION DE L'INFÂME GOTO

```
 1 /* From GCC TestSuite this code is copyrighted see GCC source code for
 2  * copyright license */
 3 /* Check the use of goto. */
 4
 5 int goto_val;
 6
 7 int
 8 test_goto1 (int f)
 9 {
10   if (f)      /* count(2) */
11     goto lab1;      /* count(1) */
12   return 1;     /* count(1) */
13 lab1:
14   return 2;     /* count(1) */
15 }
16
17 int
18 test_goto2 (int f)
19 {
20   int i;
21   for (i = 0; i < 10; i++)  /* count(15) */
22     if (i == f)
23       goto lab2;    /* count(14) */
24   return 4;     /* count(1) */
25 lab2:
26   return 8;     /* count(1) */
27 }
28 /* Added code for test not included in GCC test suite */
29 #include <stdio.h>
30 int
31 main(int argc, char * argv)
32 {
33   printf("Res1 : %d\n",goto_val += test_goto1 (0));
34   printf("Res2 : %d\n",goto_val += test_goto1 (1));
35   printf("Res3 : %d\n",goto_val += test_goto2 (3));
36   printf("Res4 : %d\n",goto_val += test_goto2 (30));
37   return 0;
38 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Res1 : 1
Res2 : 3
Res3 : 11
Res4 : 15
```

Programme 6.13 Utilisation de plusieurs `return`

```
 1 #include <stdio.h>
 2 int
 3 testc (int c)
 4 {
 5   switch (c)
 6     {
 7     case EOF:
 8       return -1;
 9     case 'A': case 'B': case 'C': case 'D': case 'E':
10     case 'F': case 'G': case 'H': case 'I': case 'J':
11     case 'K': case 'L': case 'M': case 'N': case 'O':
12     case 'P': case 'Q': case 'R': case 'S': case 'T':
13     case 'U': case 'V': case 'W': case 'X': case 'Y':
14     case 'Z':
15       return 0;
16     case '0': case '1': case '2': case '3': case '4':
17     case '5': case '6': case '7': case '8': case '9':
18       return 2;
19     case 'a': case 'b': case 'c': case 'd': case 'e':
20     case 'f': case 'g': case 'h': case 'i': case 'j':
21     case 'k': case 'l': case 'm': case 'n': case 'o':
22     case 'p': case 'q': case 'r': case 's': case 't':
23     case 'u': case 'v': case 'w': case 'x': case 'y':
24     case 'z':
25       return 1;
26     default:
27       break;
28     }
29   return -2;
30 }
```

©Christian Bac 1985-2003

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

PROGRAMME 6.14 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 1

```
 1 #include <stdio.h>
 2 int
 3 main(int argc, char *argv[], char **envp){
 4   int i, j;      /* definitions de deux entiers */
 5   printf(" Entrer les valeurs de i et de j ");
 6   scanf("%d%d", &i, &j);  /* lecture des deux entiers */
 7   /* ecriture de l'entier possedant la plus grande valeur
 8    * absolue a l'aide de l'instruction if
 9    */
10   if(i == j || i == -j){
11     printf("\nles deux entiers i (%d) et j (%d) ", i, j);
12     printf("sont egaux en valeur absolue\n");
13     return 0;
14   }
15   printf("\nle plus grand en valeur absolue de i (%d) et j (%d)", i, j);
16   printf(" est : ");
17   if(i < 0)
18     if(j < 0)
19       if(i > j) printf(" j : %d\n", j);
20       else      printf("i : %d\n", i);
21     else if(-i > j) printf("i : %d\n", i);
22     else            printf("j : %d\n", j);
23   else if(j < 0)
24     if(i > -j) printf("i : %d\n", i);
25     else       printf("j : %d\n", j);
26   else if(i > j) printf("i : %d\n", i);
27   else           printf("j : %d\n", j);
28   /* ecriture de l'entier possedant la plus grande valeur
29    * absolue a l'aide de l'operateur ternaire
30    */
31   printf("Avec l'operateur ternaire\n");
32   printf("le plus grand en valeur absolue de i (%d) et j (%d)\n", i, j);
33   printf("est : %c \n",
34    (((i < 0 ? -i : i) > (j < 0 ? -j : j)) ? 'i' : 'j'));
35   printf(" dont la valeur est : %d \n",
36    (((i < 0 ? -i : i) > (j < 0 ? -j : j)) ? i : j));
37   return 0;
38 }
```

DONNÉES EN ENTRÉE

```
-25 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer les valeurs de i et de j
le plus grand en valeur absolue de i (-25) et j (12) est : i : -25
Avec l'operateur ternaire
le plus grand en valeur absolue de i (-25) et j (12)
est : i
 dont la valeur est : -25
```

PROGRAMME 6.15 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 2

```
 1 #include <stdio.h>
 2 int
 3 main(int argc, char *argv[], char **envp){
 4   char c;          /* definition de la variable */
 5   /* saisie du caractere a analyser */
 6   printf(" Entrer une lettre : ");
 7   scanf(" %c", &c);
 8   /* transformation des majuscules en minuscules */
 9   if (c >= 'A' && c <= 'Z')
10     c += 'a' - 'A';
11   /* ecriture du type de lettre */
12   switch (c){
13   case 'a': case 'e': case 'i':
14   case 'o': case 'u': case 'y':
15     printf(" la lettre est une voyelle \n");
16     break;
17   case 'b': case 'c': case 'd': case 'f':
18   case 'g': case 'h': case 'j': case 'k':
19   case 'l': case 'm': case 'n': case 'p':
20   case 'q': case 'r': case 's': case 't':
21   case 'v': case 'w': case 'x': case 'z':
22     printf(" la lettre est une consonne \n\n");
23     break;
24   default:
25     printf(" pas une lettre !!!\n\n");
26     break;
27   }
28   return 0;
29 }
```

DONNÉES EN ENTRÉE

```
R
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer une lettre :  la lettre est une consonne
```

©Christian Bac 1985-2003

PROGRAMME 6.16 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 3

```
 1 #include <stdio.h>
 2 int tab[10];          /* definition du tableau */
 3 int
 4 main (int argc, char *argv[], char **envp){
 5   int i;
 6   int *pt;           /* pointeur pour acceder au tableau */
 7   int nbel = 5;
 8   /* premiere solution */
 9   /* remplissage du tableau a l'aide d une boucle for */
10   printf(" entrer les %d entiers : ", nbel);
11   for(i = 0; i < nbel; i++) scanf("%d", &tab[i]);
12   printf("\n");
13   /* ecriture du tableau avec une boucle while */
14   i = 0;
15   while(i < nbel)
16     printf(" l'element %d du tableau est %d \n", i, tab[i++]);
17
18   /* deuxieme solution a l'aide d'un pointeur */
19   /* remplissage du tableau a l'aide d'une boucle for */
20   pt = tab;
21   printf(" entrer les %d entiers : ", nbel);
22   for(i = 0; i < nbel; i++) scanf ("%d", pt++);
23   printf("\n");
24   /* ecriture du tableau */
25   i = 0;
26   pt = tab;
27   while(pt < &tab[nbel]){
28       printf(" l'element %d du tableau est %d \n", i, *pt++);
29       i++;
30   }
31   return 0;
32 }
```

DONNÉES EN ENTRÉE

```
1 2 3 4 5 6 7 8 9 10
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 entrer les 5 entiers :
 l'element 1 du tableau est 1
 l'element 2 du tableau est 2
 l'element 3 du tableau est 3
 l'element 4 du tableau est 4
 l'element 5 du tableau est 5
 entrer les 5 entiers :
 l'element 0 du tableau est 6
 l'element 1 du tableau est 7
 l'element 2 du tableau est 8
 l'element 3 du tableau est 9
 l'element 4 du tableau est 10
```

©Christian Bac 1985-2003

PROGRAMME 6.17 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 4

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[], char **envp){
 4   char c;
 5   int count = 0;
 6   /* saisie du caractere a analyser */
 7   printf(" Entrer une lettre : ");
 8   count = scanf(" %c", &c);
 9   if (count <= 0) return 0;
10   do{
11     /* transformation des majuscules en minuscules */
12     if (c >= 'A' && c <= 'Z')
13       c += 'a' - 'A';
14     /* ecriture du type de lettre */
15     switch(c){
16     case 'a': case 'e': case 'i':
17     case 'o': case 'u': case 'y':
18       printf(" la lettre est une voyelle \n");
19       break;
20     case 'b': case 'c': case 'd': case 'f':
21     case 'g': case 'h': case 'j': case 'k':
22     case 'l': case 'm': case 'n': case 'p':
23     case 'q': case 'r': case 's': case 't':
24     case 'v': case 'w': case 'x': case 'z':
25       printf(" la lettre est une consonne \n");
26       break;
27     default:
28       printf(" pas une lettre !!!\n\n");
29       break;
30     }
31     printf(" Entrer une lettre : ");
32     count = scanf(" %c", &c);
33   }while(count == 1);
34   return 0;
35 }
```

DONNÉES EN ENTRÉE

```
qwerty
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer une lettre :  la lettre est une consonne
 Entrer une lettre :  la lettre est une consonne
 Entrer une lettre :  la lettre est une voyelle
 Entrer une lettre :  la lettre est une consonne
 Entrer une lettre :  la lettre est une consonne
 Entrer une lettre :  la lettre est une voyelle
 Entrer une lettre :
```

©Christian Bac 1985-2003

PROGRAMME 6.18 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 5

```
 1 #include <stdio.h>
 2 char tab[100];  /* ligne initiale */
 3 char ntab[100]; /* ligne avec blancs compresses */
 4 int
 5 main (int argc, char *argv[], char **envp)
 6 {
 7   char c;
 8   int i = 0;  /* indice dans tab */
 9   int j = 0;  /* indice dans ntab */
10   int blk = 0;  /* 1 si blanc rencontre 0 sinon */
11   int s = 0;  /* marque de debut de ligne */
12   /* remplissage du tableau par lecture de la ligne */
13   printf(" Entrer une chaine avec des tabs et des blancs \n");
14   while((c = getchar ()) != '\n'){
15     if (i >= 99) break;
16     tab[i++] = c;
17   }
18   tab[i] = '\0';
19   /* parcours de tab et passage dans ntab en supprimant */
20   /* les caracteres inutiles */
21   i = 0;
22   while((c = tab[i++]) != '\0'){
23     /* elimination des tabs et des blancs de debut de ligne */
24     if((c == ' ' || c == '\t') && s == 0) continue;
25     s = 1;
26     /* remplacement de la tabulation par un blanc */
27     if(c == '\t') c = ' ';
28     /* remplacement de plusieurs blancs par un seul */
29     if(c == ' ')
30       if(blk != 0) continue;
31       else   blk = 1;
32     else   blk = 0;
33     ntab[j++] = c;
34   }
35   ntab[j] = '\0';          /* on assure la fin de chaine */
36   printf("\nAncienne ligne :\n %s\n", tab);
37   printf("Nouvelle ligne :\n %s\n", ntab);
38   return 0;
39 }
```

DONNÉES EN ENTRÉE

```
       parcours  de     tab        et     passage   dans ntab en supprimant
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer une chaine avec des tabs et des blancs

Ancienne ligne :
        parcours  de     tab        et     passage   dans ntab en supprimant
Nouvelle ligne :
 parcours de tab et passage dans ntab en supprimant
```

©Christian Bac 1985-2003

PROGRAMME 6.19 SUGGESTION DE CORRIGÉ CHAPITRE 6 EXERCICE 6

```
 1 #include <stdio.h>
 2
 3 /* Tableau contenant l'alphabet */
 4 char tab[] = "abcdefghijklmnopqrstuvwxyz";
 5 int
 6 main(int argc, char *argv[], char **envp){
 7   int i, count;
 8   /* boucle de lecture */
 9   while (1){
10     i = 0;
11     printf("entrer un nombre : ");
12     count = scanf("%d", &i);
13     if(count != 1) break;   /* on n'a pas lu un nombre */
14     if(i == 100)   break;   /* sortie de la boucle */
15     if(i < 1 || i > 26)  continue;  /* nombre n'est pas entre 1 et 26 */
16     /* ecriture de la lettre correspondante */
17     printf("\nle nombre %d correspond a la lettre %c \n", i, tab[i - 1]);
18   }
19   printf("\nFini\n");
20   return 0;
21 }
```

DONNÉES EN ENTRÉE

```
12 4 -2 6 35 26 100
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer un nombre :
le nombre 12 correspond a la lettre l
entrer un nombre :
le nombre 4 correspond a la lettre d
entrer un nombre : entrer un nombre :
le nombre 6 correspond a la lettre f
entrer un nombre : entrer un nombre :
le nombre 26 correspond a la lettre z
entrer un nombre :
Fini
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 7

Programmation structurée

La programmation structurée est un nom générique qui couvre un courant de pensées qui 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.
   - **if orif** : permettant de décrire un choix multiple.

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

©Christian Bac 1985-2003

   – **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 n'a cependant pas l'ambition d'obliger les programmeurs à respecter un quelconque style de programmation, ilest en effet 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 les logiciels de base de ce système (interpréteur de commande, compilateur, ...). Pour ce faire, il doit être capable de faire les mêmes choses que l'assembleur. Il est assez peu contraignant car ces concepteurs l'ont créé pour leur propre utilisation, ils ont donc préféré favoriser l'expressivité du langage (d'où la richesse des expressions) que d'éviter les erreurs de programmation en multipliant les tests à la compilation. Cette approche est différente 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 programme 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.

©Christian Bac 1985-2003

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

Nous allons réaliser cet exercice avec des tests et une boucles.

Sur les systèmes de type UNIX, les outils tels que `cb` ou `indent` peuvent être utilisés pour mettre en page du code mal présenté. Ils fournissent un bon point de départ pour avoir des idées de la mise en page correcte. L'éditeur de `emacs` réalise ce type d'assistance pour le langage C mais aussi pour d'autre langages.

7.4.1 Exemple avec des tests

La tâche réalisée par l'ensemble de tests que nous allons écrire de plusieurs manières différentes, 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 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 apparaître 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 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`.

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.

7.4.2 Exemple avec une boucle

Nous allons réaliser une fonciton qui calcule la longueur d'un chaîne de caractère (vue comme un tableau dont la fonction ne connait pas la taille), la boucle réalise le parcours du tableau de caractères (`tabc`) et continue tant que la caractère lu n'est pas le caractère `'\0'`.

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

Le programme 7.8 réalise ce parcours de tableau, il est écrit de manière structurée avec un `for` mais la mise en page laisse à désirer.

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

PROGRAMME 7.1 ENSEMBLE DE TESTS NON STRUCTURÉ

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j)
 9     goto T3;
10   if (i <= 0)
11     goto L1;
12   if (j <= 0)
13     goto L2;
14   if (i > j)
15     goto T2;
16   goto T1;
17 L2:if (i > -j)
18     goto T2;
19   goto T1;
20 L1:if (j <= 0)
21     goto L3;
22   if (-i > j)
23     goto T2;
24   goto T1;
25 L3:if (-i > -j)
26     goto T2;
27 T1:printf (" le plus grand est j : %d\n", j);
28   goto T4;
29 T2:printf (" le plus grand est i : %d\n", i);
30   goto T4;
31 T3:printf ("  i et j sont egaux\n");
32 T4:
33   return 0;
34 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres :  le plus grand est j : 12
```

©Christian Bac 1985-2003

PROGRAMME 7.2 ENSEMBLE DE TESTS STRUCTURÉ NON MIS EN PAGE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j) printf ("  i et j sont egaux\n");
 9   else if (i < 0) if (j < 0)
10   if (-i > -j) printf (" le plus grand est i : %d\n", i);
11   else printf (" le plus grand est j : %d\n", j);
12   else if (-i > j) printf (" le plus grand est i : %d\n", i);
13   else printf (" le plus grand est j : %d\n", j);
14   else if (j < 0) if (i > -j) printf (" le plus grand est i : %d\n", i);
15   else printf (" le plus grand est j : %d\n", j);
16   else if (i > j) printf (" le plus grand est i : %d\n", j);
17   else printf (" le plus grand est j : %d\n", j);
18   return 0;
19 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres :  le plus grand est j : 12
```

©Christian Bac 1985-2003

PROGRAMME 7.3 ENSEMBLE DE TESTS STRUCTURÉ ET MIS EN PAGE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j)
 9     printf ("  i et j sont egaux\n");
10   else if (i < 0)
11     if (j < 0)
12       if (-i > -j)
13   printf ("le plus grand est i : %d\n", i);
14       else
15   printf ("le plus grand est j : %d\n", j);
16     else if (-i > j)
17       printf ("le plus grand est i : %d\n", i);
18     else
19       printf ("le plus grand est j : %d\n", j);
20   else if (j < 0)
21     if (i > -j)
22       printf ("le plus grand est i : %d\n", i);
23     else
24       printf ("le plus grand est j : %d\n", j);
25   else if (i > j)
26     printf ("le plus grand est i : %d\n", i);
27   else
28     printf ("le plus grand est j : %d\n", j);
29   return 0;
30 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres : le plus grand est j : 12
```

©Christian Bac 1985-2003

PROGRAMME 7.4 ENSEMBLE DE TESTS STRUCTURÉ ET MIS EN PAGE AVEC COMMENTAIRES

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j) printf ("  i et j sont egaux\n");
 9   else{
10     if (i < 0){
11       if (j < 0)
12   if (-i > -j) printf ("le plus grand est i : %d\n", i);
13   else           printf ("le plus grand est j : %d\n", j);
14       else if (-i > j)
15   printf ("le plus grand est i : %d\n", i);
16       else
17   printf ("le plus grand est j : %d\n", j);
18     }else{         /* i >= 0 */
19       if (j < 0){
20   if (i > -j) printf ("le plus grand est i : %d\n", i);
21   else        printf ("le plus grand est j : %d\n", j);
22       }    /* fin j < 0 */
23       else{
24   if (i > j) printf ("le plus grand est i : %d\n", i);
25   else       printf ("le plus grand est j : %d\n", j);
26       }        /* fin j > 0 */
27     }          /* fin i > 0 */
28   }            /* fin i != j */
29   return 0;
30 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres : le plus grand est j : 12
```

©Christian Bac 1985-2003

PROGRAMME 7.5 ENSEMBLE DE TESTS STRUCTURÉ, MIS EN PAGE ET ÉTIQUETÉ

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j)
 9     printf ("  i et j sont egaux\n");
10   else
11     {                             /*  ELIF 1 */
12       if (i < 0)
13         {                        /* ELIF1.IF1 */
14           if (j < 0)
15             if (-i > -j)
16               printf (" le plus grand est i : %d\n", i);
17             else
18               printf (" le plus grand est j : %d\n", j);
19           else if (-i > j)
20             printf (" le plus grand est i : %d\n", i);
21           else
22             printf (" le plus grand est j : %d\n", j);
23         }                        /* fin ELIF 1.IF1 */
24       else
25         {                        /* ELIF 1.ELIF1 */
26           if (j < 0)
27             {
28               if (i > -j)
29                 printf (" le plus grand est i : %d\n", i);
30               else
31                 printf (" le plus grand est j : %d\n", j);
32             }                    /* fin j < 0 */
33           else
34             {
35               if (i > j)
36                 printf (" le plus grand est i : %d\n", i);
37               else
38                 printf (" le plus grand est j : %d\n", j);
39             }                    /* fin j > 0 */
40         }                        /* fin ELIF1.ELIF1 */
41     }                            /* fin ELIF 1 */
42   return 0;
43 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres :  le plus grand est j : 12
```

©Christian Bac 1985-2003

PROGRAMME 7.6 ENSEMBLE DE TESTS PAR TABLE DE BRANCHEMENT

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   int i, j;
 6   printf ("Veuillez entrer deux valeurs entieres : ");
 7   scanf ("%d%d", &i, &j);
 8   if (i == j)
 9     printf ("  i et j sont egaux\n");
10   else
11     {
12       /* l'expression (i<0) + 2*(j<0)) donne la valeur : */
13       /*      0  si i>0 et j>0,     1  si i<0 et j>0     */
14       /*      2  si i>0 et j<0      3  si i<0 et j<0     */
15       switch ((i < 0) + 2 * (j < 0))
16         {
17         case 0:
18           if (i > j)
19             printf ("le plus grand est i : %d\n", i);
20           else
21             printf ("le plus grand est j : %d\n", j);
22           break;
23         case 1:
24           if (-i > j)
25             printf ("le plus grand est i : %d\n", i);
26           else
27             printf ("le plus grand est j : %d\n", j);
28           break;
29         case 2:
30           if (i > -j)
31             printf ("le plus grand est i : %d\n", i);
32           else
33             printf ("le plus grand est j : %d\n", j);
34           break;
35         case 3:
36           if (-i > -j)
37             printf ("le plus grand est i : %d\n", i);
38           else
39             printf ("le plus grand est j : %d\n", j);
40         }           /* switch */
41     }               /* fin i != j */
42   return 0;
43 }
```

DONNÉES EN ENTRÉE

```
10 12
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Veuillez entrer deux valeurs entieres : le plus grand est j : 12
```

©Christian Bac 1985-2003

Programme 7.7 Boucle réalisée par des tests et sauts

```
 1 int
 2 mystrlen (char tabc[])
 3 {
 4   int i;
 5   i = 0;
 6 B1:
 7   if (tabc[i] == '\0')
 8     goto B2;
 9   i = i + 1;
10   if (i < 100)
11     goto B1;
12 B2:
13   return i;
14 }
```

Programme 7.8 Boucle réalisée par un for()

```
1 int
2 mystrlen (char tabc[])
3 {
4   int i;
5   for (i = 0; i < 100; i++) if (tabc[i] == '\0') break;
6   return i;
7 }
```

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

```
1 int
2 mystrlen (char tabc[])
3 {
4   int i;
5   for (i = 0; i < 100; i++)
6     if (tabc[i] == '\0')
7       break;
8   return i;
9 }
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 8

Fonctions

Les fonctions sont des parties de code source qui permettent de réaliser le même type de traitement plusieurs fois et/ou sur des variables différentes. 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 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 qui est en fait un bloc d'instructions.

L'interface complète d'une fonction contient :
– la déclaration du type de retour et du nom de la fonction ;
– une parenthèse ouvrante ;
– la déclaration des types et des noms des paramètres ;
– une parenthèse fermante.

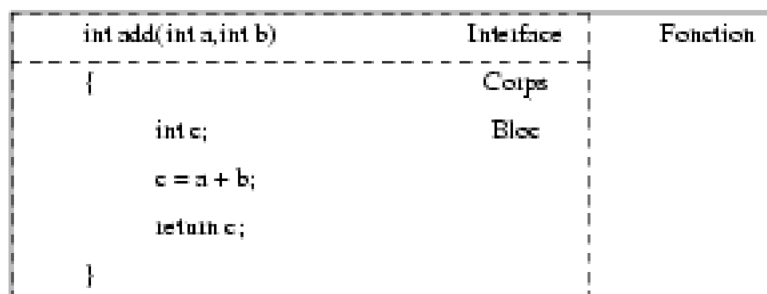

FIG. 8.1 – Structure d'une fonction

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 accepte 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

L'interface d'une fonction peut être incomplète si elle ne contient pas le nom des paramêtres comme nous le verrons dans le prochain chapitre, cette interface est aussi appelée signature de la fonction, et est utilisées lorsque des fonctions proviennent de différents fichiers sources ou de bibliothèques.

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 définitions de fonctions en C.

8.2 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 l'encart suivant.

```
return expression ;
```

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 `plus()` vue dans le tableau 8.1, 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 :

©Christian Bac 1985-2003

Définition	Appel

`int plus(int a,int b)`
`{`
`a = a + b;`
`return(a);`
`}`
	`int main(int argc, char *argv[])`
	`{`
	` int x=4,y=6,z;`
	` z = plus(1,23);`
	` z = plus(x,y);`
	`}`

TAB. 8.2 – Exemples d'appels de fonctions

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

8.3 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.

8.3.1 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 la fonction correspondant à la ligne `z = plus(1,23);`.

0 la pile contient les valeurs initiales de `x`, `y` et `z`, le sommet de pile est juste au dessus de `z`. Comme la variable `z` n'a pas été initialisée nous n'en connaissons pas la valeur ;

Étape 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 ;

Étape 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 connaît sous les noms "a" et "b". Ces deux variables sont les copies des valeurs dont nous avons parlé à l'étape précédente. Elles ont donc pour valeurs initiales respectivement 1 et 23 ;

Étape 3 la fonction `plus()` modifie la valeur de la variable "a" en exécutant la ligne `a=a+b;`, elle met donc la valeur 24 dans "a" ;

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

Étape 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).

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

La vision donnée par les schémas de la figure 8.2, de même que les autres dessins de ce chapitre, est simpliste en ce sens qu'elle considère 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 considère 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.

©Christian Bac 1985-2003

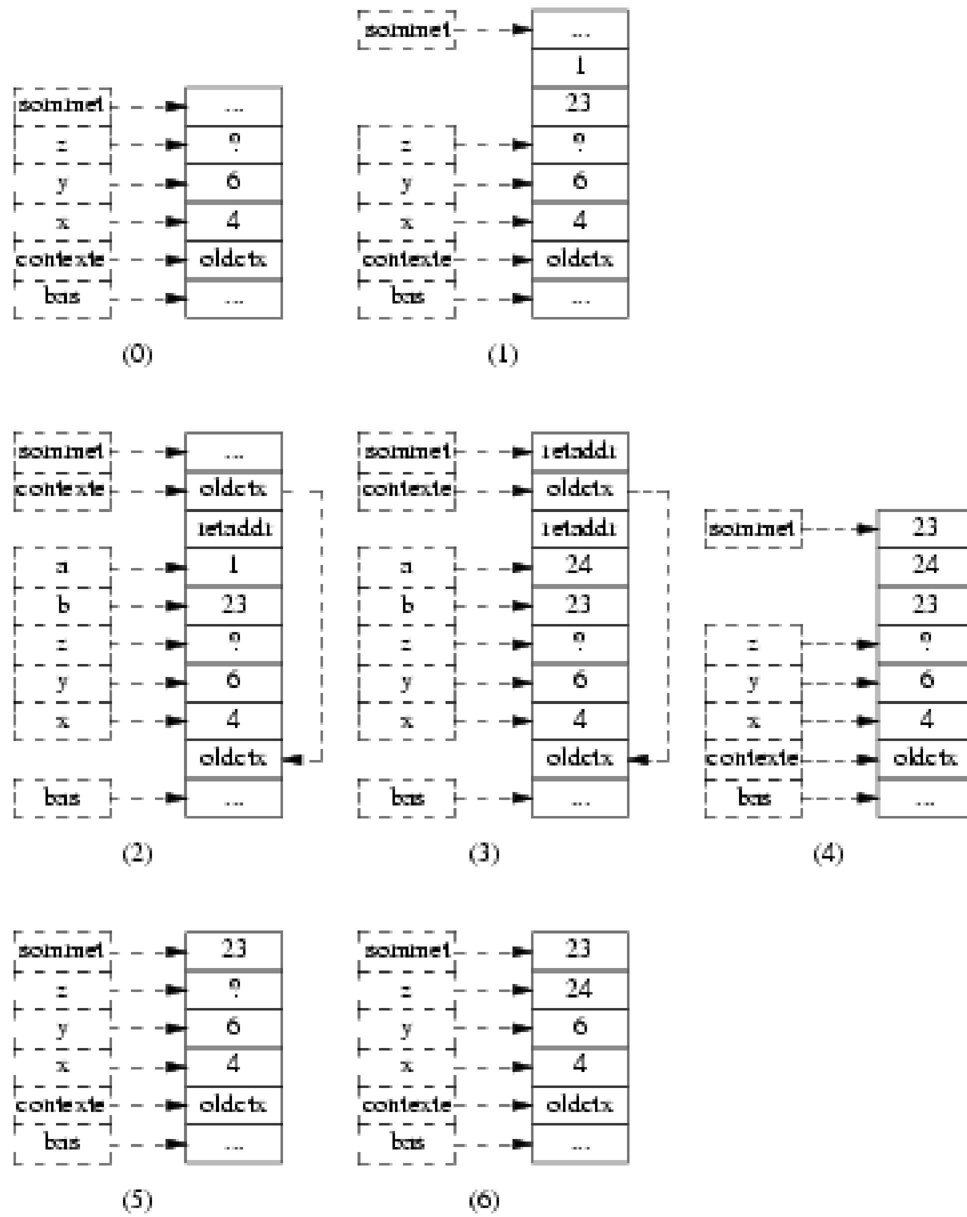

FIG. 8.2 – Pile et passage de constantes

©Christian Bac 1985-2003

Déclaration	Appel

`add(int a,int b,int *c)`
`{`
`    *c=a+b;`
`}`
	`int main(int argc, char *argv[])`
	`{ int x=5,y=7,z;`
	`add(x,y,&z);`
	`add(43,4,&x);`
	`}`

TAB. 8.3 – Pointeurs et appels de fonctions

8.3.2 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 :

Étape 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 ;

Étape 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 à l'étape précédente. Ces deux variables ont donc pour valeurs initiales 4 et 6 ;

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

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

Étape 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".

8.4 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].

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.

La figure 8.4, montre les différents états de la pile lors de l'exécution du premier appel `add(x,y,&z);` qui modifie la variable `z` en lui affectant la valeur 12 (5+7), dans les étapes 0 à 5. Lors du deuxième appel `add(43,4,&x);` la variable `x` est modifiée et elle prend la valeur 47 (43+4). Seules les étapes premières étapes de cet appel sont représentées sur la figure. Ces étapes de 6 à 8 correspondent à l'appel de fonction et l'exécution de l'affectation en utilisant le pointeur. Le retour de la fonction appelée à la fonction appelante n'est pas représenté. Il suffit de reprendre les étapes numérotés 4 et 5 en changeant la fléche qui va du pointeur à la variable associée (ici x au lieu de z) et la valeur de la case mémoire correspondante (x doit contenir 47).

[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.

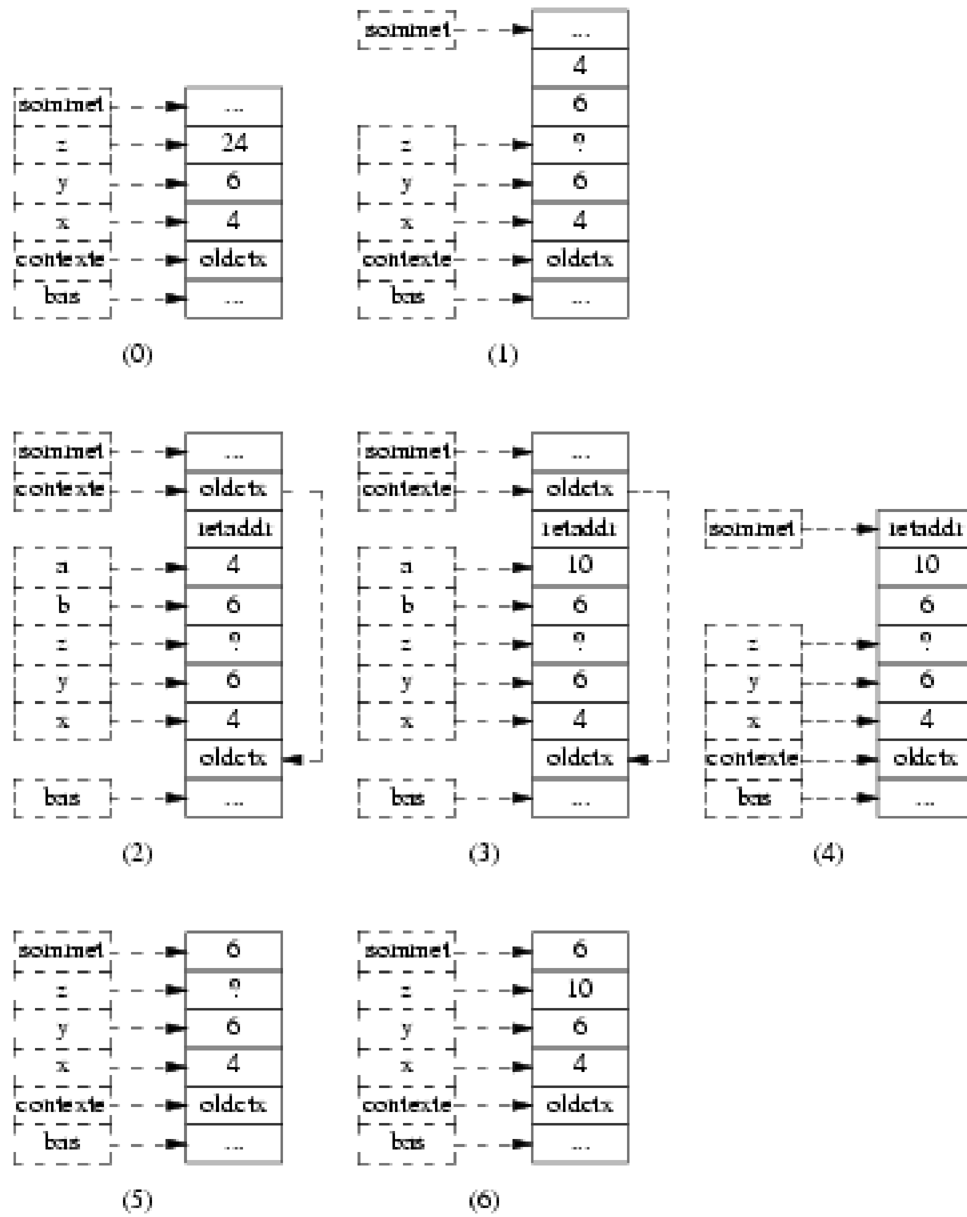

FIG. 8.3 – Pile et passage de variables

©Christian Bac 1985-2003

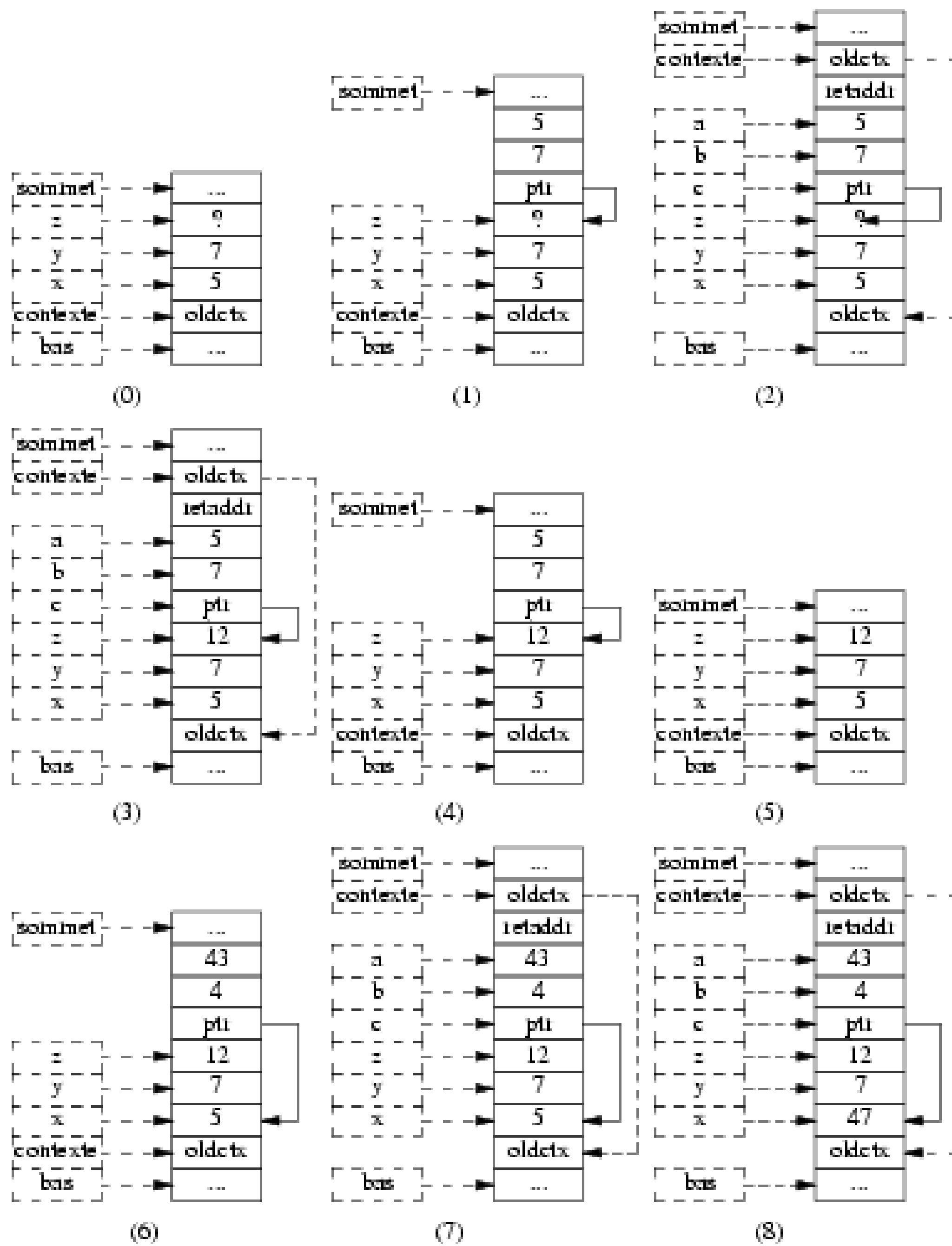

FIG. 8.4 – Pile et passage de variables avec référence

©Christian Bac 1985-2003

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

TAB. 8.4 – Conversions de type un-aire

8.5 Conversion de type lors des appels

Le C selon la description de Kernighan et Ritchie décrivait la façon avec laquelle les variables étaient promues lors des appels de fonction. Cette règle porte d'une part sur les nombres à virgule flottante qui sont transfomés en `double`, d'autre part sur tout les types entier dont la taille est plus petite que l'entier naturel. Lors de l'appel d'une fonction, les paramètres subissent les conversions de type un-aire telles que décrites dans le tableau 8.4. 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 ayant lieu lors de l'appel d'une fonction sont décrites dans le tableau 8.4. Les variables ou les constantes des types suivants sont utilisées dans une expression, 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.

La norme C99 maintient ce comportement pour des questions de compatibilités si les fonctions n'ont pas d'interface connue avant l'appel. Dans le cas normal (les fonctions ont une interface connue avant leur utilisation) les valeurs des variables sont passées en maintenant leur type.

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; }
```

©Christian Bac 1985-2003

8.7 Arguments 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)`. 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 pointer*.

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

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

argv 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`) ;
- la chaîne pointée par la première entrée du tableau `argv` contient le nom de la commande elle-même.

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

Les programmes 8.1 et 8.2 donnent deux exemples d'utilisation des arguments du `main` et des variables d'environnement, en utilisant l'impression de chaîne de caractéres par `printf()` en utilisant le format `%s` ou en imprimant chaque caractère un-à-un, en utilisant le format `%c`.

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 Étapes 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

©Christian Bac 1985-2003

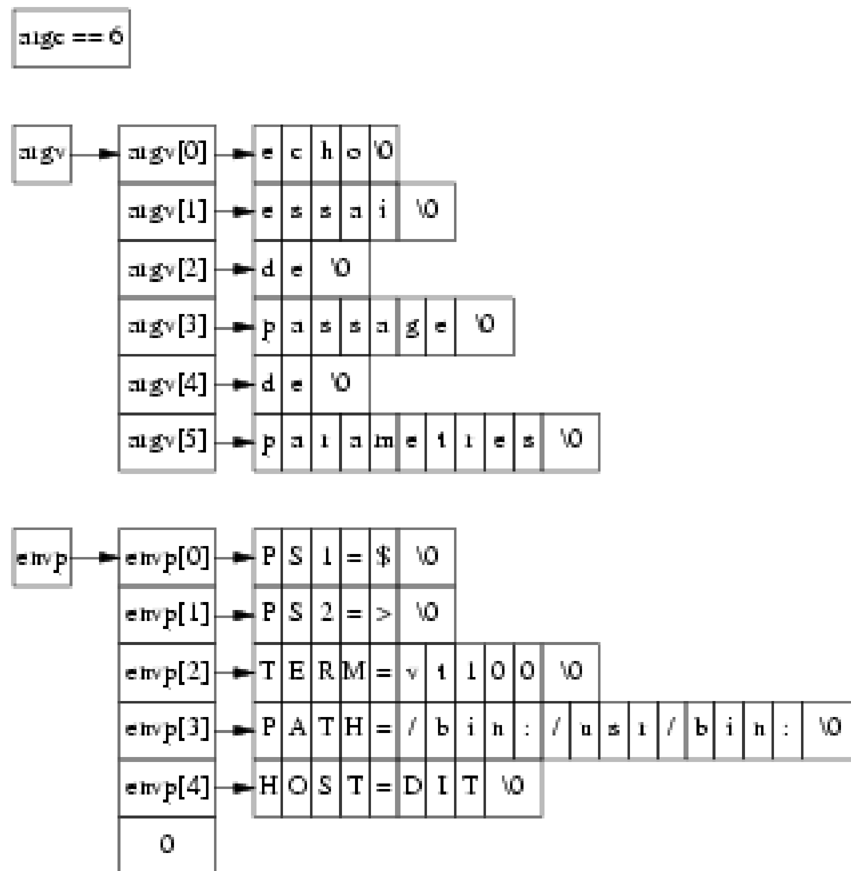

FIG. 8.5 – Arguments de `main()`

PROGRAMME 8.1 AFFICHAGE DES ARGUMENTS DE `MAIN()`

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[],char **envp)
 4 {
 5   int i;
 6   printf("Nous avons %d arguments : \n",argc);
 7   for(i=0;i<argc;i++){
 8     printf("argument %d == %s \n",i,argv[i]);
 9   }
10   printf("Nous affichons les 5 premières variables d'environnement \n");
11   printf("mais nous les comptons toutes. \n");
12   for(i=0; envp[i] ;i++){
13     if(i<5)
14       printf("Envp[%d] : %s \n",i,envp[i]);
15   }
16   printf("Il y a %d variables d'environnement : \n",i);
17   return 0;
18 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Nous avons 3 arguments :
argument 0 == c08e01
argument 1 == c08e01
argument 2 == c08e01.c
Nous affichons les 5 premières variables d'environnement
mais nous les comptons toutes.
Envp[0] : BIBINPUTS=.:/home/chris/Bib:/usr/share/texmf/bibtex/bib
Envp[1] : NNTPSERVER=news.int-evry.fr
Envp[2] : MANPATH=/usr/X11R6/man:/usr/local/man:/usr/share/man
Envp[3] : COURSC=/home/chris/Enseignement/COURS/C
Envp[4] : SSH_AGENT_PID=495
Il y a 65 variables d'environnement :
```

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 tout 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 tout 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`, ce `return` provoque le passage à l'épilogue dans la fonction appelée. Lorsque le `return` est associé à une valeur cette valeur est conservée dans un registre de calcul (qui sert aussi pour évaluer les expressions).

Épilogue 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[2]

[2]Ce sont les registres les plus utilisés, ils sont considérés par le compilateur comme utilisables pour l'évaluation de chaque ligne

qui contiennent la valeur de retour de la fonction). Pour cela, l'épilogue restaure les registres qu'il avait sauvegardés (correspondants aux registres demandés par les définitions de variables de type register), puis il restaure le contexte de pile en reprenant l'ancienne valeur dans la pile. Enfin, il replace le pointeur de sommet de pile à l'adresse qu'il avait avant le prologue. Finalement, il retourne à la fonction appelante en remettant dans le pointeur d'instruction la valeur qui avait été sauvegardée sur la pile lors de l'appel. L'éxécution continue alors dans 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.

8.10 Exercices sur les fonctions

8.10.1 Exercice 1

Définir trois variables `i`, `j`, `k` de type entier et de classe globale.

Écrire une fonction `globadd()` qui fait l'addition de `i` et `j` dans `k`.

Écrire la fonction `main()` qui réalise la saisie des variables `i` et `j`, fait appel à la fonction d'addition puis écrit le résultat contenu dans `k`.

8.10.2 Exercice 2

Même exercice que le précédent mais en utilisant le passage de paramètres et le retour de fonction.

Les trois variables `i`, `j`, `k` de type entier sont déclarées localement dans la fonction `main()`. La fonction d'addition est une fonction retournant un entier. Elle accepte deux paramètres entiers (`p1` et`p2`) et retourne la somme de ces deux paramètres.

La fonction `main()` saisit les deux variables locales `i` et `j` et appelle la fonction d'addition en récupérant le résultat de cette fonction dans la variable locale `k`. Elle écrit le contenu de `k` après l'appel de fonction.

8.10.3 Exercice 3

Même exercice que le précédent mais en utilisant le passage de paramètres et un pointeur pour modifier une variable dans la fonction appelante.

Les trois variables `i`, `j`, `k` de type entier sont declarées localement dans la fonction `main()`. La fonction d'addition `ptadd()` est une fonction sans retour. Elle accepte trois paramètres entiers (`p1` et`p2`) et un paramètre de type pointeur vers un entier qui sert pour affecter la variable dont la fonction appelante passe l'adresse avec la somme de ces deux premiers paramètres.

La fonction `main()` saisit les deux variables locales `i` et `j` et appelle la fonction d'addition en passant l'adresse de la variable locale `k`. Elle écrit le contenu de `k` après l'appel de fonction.

de code.

©Christian Bac 1985-2003

PROGRAMME 8.2 ARGUMENTS DE MAIN() CARACTÈRES UN-À-UN

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[],char **envp)
 4 {
 5   int i;
 6   register char *ptc;
 7   printf("Nous avons %d arguments : \n",argc);
 8   for(i=0;i<argc;i++){
 9     printf("argument %d == ",i);
10     for(ptc=argv[i];*ptc;ptc++)
11       printf("%c",*ptc);
12     printf("\n");
13   }
14   printf("Nous affichons les 5 premières variables d'environnement \n");
15   printf("mais nous les comptons toutes. \n");
16   for(i=0; envp[i] ;i++){
17     if(i<5){
18       printf("Environnement %d == ",i);
19       for(ptc=envp[i];*ptc;ptc++)
20   printf("%c",*ptc);
21       printf("\n");
22     }
23   }
24   printf("Il y a %d variables d'environnement : \n",i);
25   return 0;
26 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Nous avons 3 arguments :
argument 0 == c08e02
argument 1 == c08e02
argument 2 == c08e02.c
Nous affichons les 5 premières variables d'environnement
mais nous les comptons toutes.
Environnement 0 == BIBINPUTS=.:/home/chris/Bib:/usr/share/texmf/bibtex/bib
Environnement 1 == NNTPSERVER=news.int-evry.fr
Environnement 2 == MANPATH=/usr/X11R6/man:/usr/local/man:/usr/share/man
Environnement 3 == COURSC=/home/chris/Enseignement/COURS/C
Environnement 4 == SSH_AGENT_PID=495
Il y a 65 variables d'environnement :
```

©Christian Bac 1985-2003

PROGRAMME 8.3 SUGGESTION DE CORRIGÉ CHAPITRE 8 EXERCICE 1

```
 1 #include <stdio.h>
 2
 3 /* declaration des variables globales */
 4 int i;
 5 int j;
 6 int k;
 7
 8
 9 void
10 globadd(){     /* fonction addition */
11   k = i + j;
12 }
13
14 int
15 main(int argc, char *argv[], char **envp){
16   /* saisie des valeurs */
17   printf("entrer 1 entier\n");
18   scanf("%d", &i);
19   printf("entrer un autre entier\n");
20   scanf("%d", &j);
21   /* appel de add et impression du resultat */
22   globadd();
23   printf(" i + j = %d\n", k);
24   return 0;
25 }
```

DONNÉES EN ENTRÉE

```
24 67
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer 1 entier
entrer un autre entier
 i + j = 91
```

PROGRAMME 8.4 SUGGESTION DE CORRIGÉ CHAPITRE 8 EXERCICE 2

```
 1 #include <stdio.h>
 2
 3 /*  fonction addition  */
 4
 5 int
 6 add(int p1, int p2){
 7   return (p1 + p2);
 8 }
 9
10 int
11 main(int argc, char *argv[], char **envp){
12   /* declarations des variables locales */
13   int i, j, k;
14   /* saisie des valeurs */
15   printf("entrer 1 entier :");
16   scanf("%d", &i);
17   printf("entrer un autre entier :");
18   scanf("%d", &j);
19   /* appel de add */
20   k = add (i, j);
21   printf(" i + j = %d\n", k);
22   return 0;
23 }
```

DONNÉES EN ENTRÉE

```
24 67
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer 1 entier :entrer un autre entier : i + j = 91
```

PROGRAMME 8.5 SUGGESTION DE CORRIGÉ CHAPITRE 8 EXERCICE 3

```
 1 #include <stdio.h>
 2
 3 /*  fonction addition  */
 4
 5 void
 6 ptadd(int p1, int p2, int *pti)
 7 {
 8   *pti = p1 + p2;
 9   return;
10 }
11
12 int
13 main(int argc, char *argv[], char **envp)
14 {
15   /* declarations des variables locales */
16   int i, j, k;
17   /* saisie des valeurs */
18   printf("entrer 1 entier :");
19   scanf("%d", &i);
20   printf("entrer un autre entier :");
21   scanf("%d", &j);
22   /* appel de add */
23   ptadd(i, j, &k);
24   printf(" i + j = %d\n", k);
25   return 0;
26 }
```

DONNÉES EN ENTRÉE

```
24 67
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
entrer 1 entier :entrer un autre entier : i + j = 91
```

©Christian Bac 1985-2003

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.

À partir de maintenant, nous appellerons :
- **déclaration** : une association de type avec un nom de variable ou de fonction (dans ce cas la déclaration contient aussi le type des arguments de la fonction),
- **définition** : une déclaration et si c'est une variable, une demande d'allocation d'espace pour cette variable, si c'est une fonction la définition du corps de fonction contenant les instructions associées à cette fonction.

9.1 Programme

Comme nous l'avons déjà dit dans le chapitre 1, un programme en langage C est 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 schématiquement 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 (voir chapitre 14),
- des définitions de variables [1],
- des définitions de fonctions,
- des directives de pré-compilation et des commentaires.

Les directives de pré-compilation et les commentaires sont traités par le pré-processeur.

Le compilateur ne voit que les quatre premiers types d'objets.

Les fichiers inclus par le pré-processeur 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 transformées par l'éditeur de liens en des demandes de réservation mémoire à réaliser lors du démarrage du programme.

©Christian Bac 1985-2003

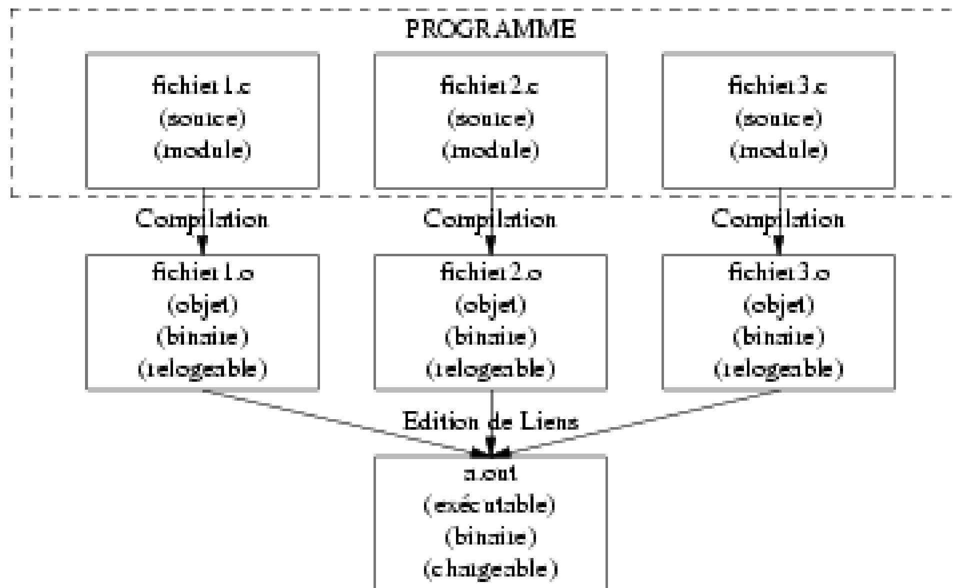

FIG. 9.1 – Du source à l'exécutable

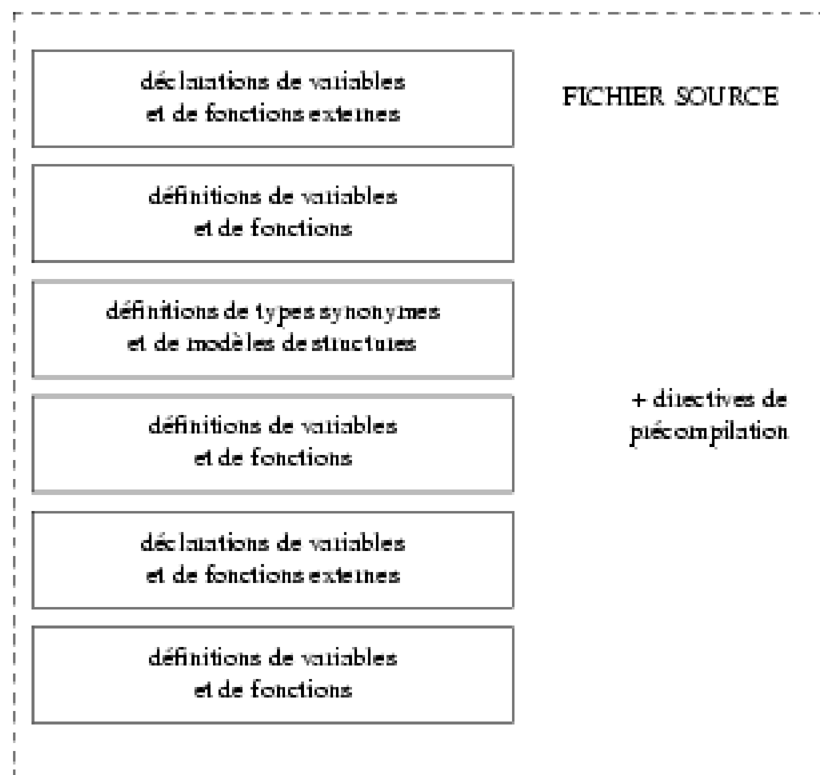

FIG. 9.2 – Survol d'un fichier source

©Christian Bac 1985-2003

Règle fondamentale :

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

TAB. 9.1 – Règle fondamentale de visibilité

```
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 */
}
void f4(int c)
{
    a = 10;
}
```

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 appellerons **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 jusqu'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 d'inférer 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 :

1. la fonction f4 peut appeler les fonctions f4, f3, f2, et f1 ;
2. la fonction f3 peut appeler les fonctions f3, f2, et f1 ;
3. la fonction f2 peut appeler la fonction f2, et f1 ;
4. la fonction f1 ne peut que s'appeler elle-même ;

©Christian Bac 1985-2003

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.

9.3.1 Espaces de nommage et visibilité

Nous avons dit dans le chapitre sur la généralité sur la syntaxe et dans la section2.1.2 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 figure9.3 :
- 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.

9.3.2 Extension de la visibilité

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

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

Bien entendu, une telle variable doit être définie dans un autre fichier du programme et tous les fichiers doivent être associés par une édition de liens, sinon cette dernière se termine avec des références non résolues et le fichier binaire exécutable n'est pas produit.

La figure 9.4 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.
- 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.

©Christian Bac 1985-2003

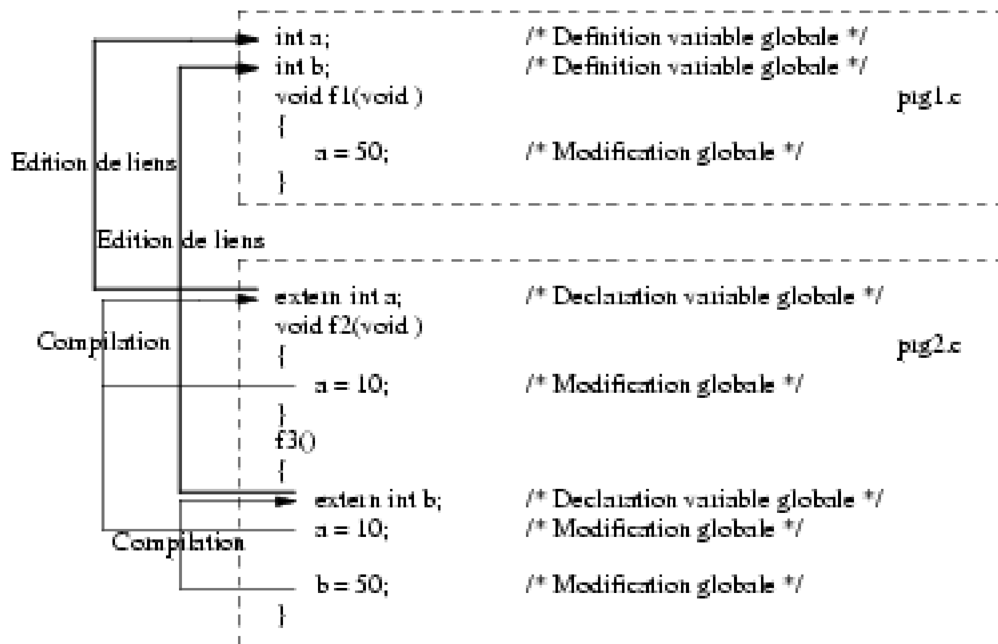

FIG. 9.4 – Visibilité des variables entre modules

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.

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 9.5, 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);`.

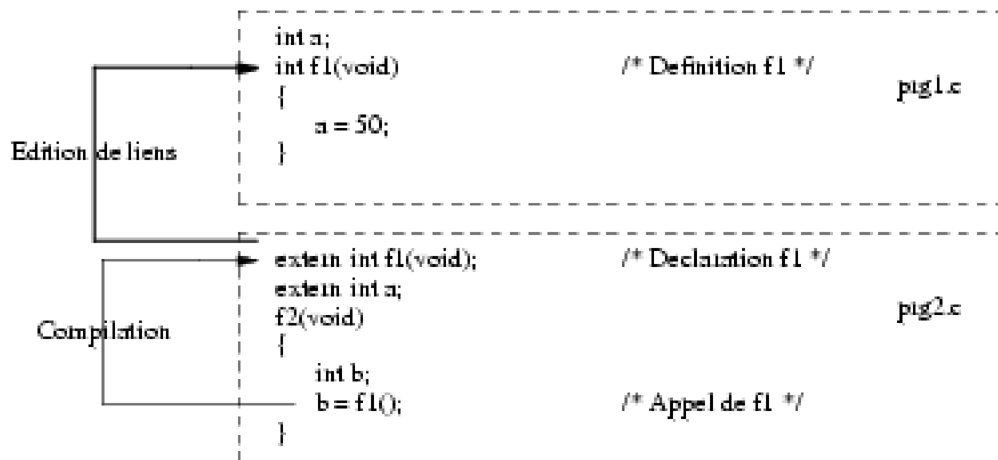

FIG. 9.5 – Visibilité des fonctions entre modules

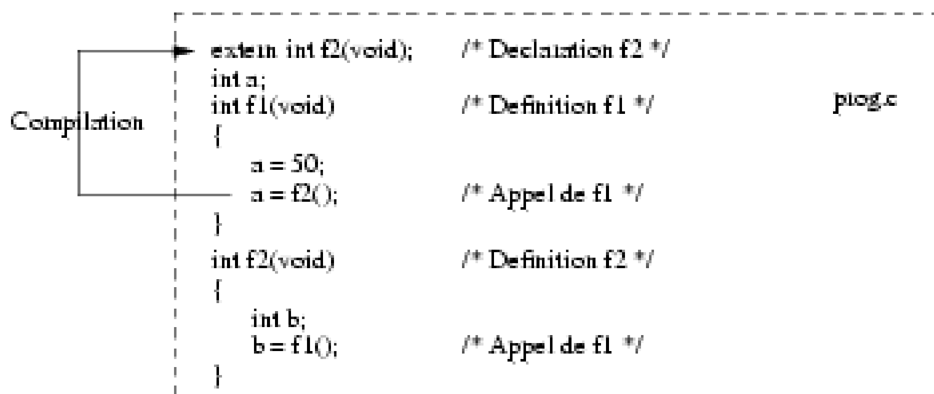

FIG. 9.6 – Visibilité des fonctions dans un module

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

9.7 Vérification 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 ce genre des vérifications. La normalisation du langage C a introduit de manière complète l'utilisation des prototypes de manière à garantir une bonne prise en compte des prototypes.

©Christian Bac 1985-2003

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 sûr n'y avoir qu'une seule initialisation qui transforme la 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.

L'exemple 9.2 montre l'utilisation de plusieurs déclarations candidates à devenir une définition d'une même variable. Dans cet exemple les deux fonctions `plusplus()` et `moinsmoins()` maninpulent la même variable `a`.

Programme 9.2 Déclarations candidates multiple d'une variable

```
1 int a;
2 void plusplus(void){a++;}
3
4 int a;
5 void moinsmoins(void){a--;}
```

Nous recommandons :
– d'éviter les déclarations candidates à la définition,
– de toujours associer la définition avec une initialisation,
– d'utiliser le mot `extern` devant les déclarations.
Le programme 9.3 reprend notre programme 9.2 en étant conforme à ces recommandations.

Programme 9.3 Déclaration explicite et déclaration candidate d'une variable

```
1 extern int a;
2 void plusplus(void){a++;}
3
4 int a=0;
5 void moinsmoins(void){a--;}
```

`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 [2] d'objets des types suivants :
– types non prédéfinis,
– modèles et noms de structures,
– types et noms de variables,

[2] 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.

©Christian Bac 1985-2003

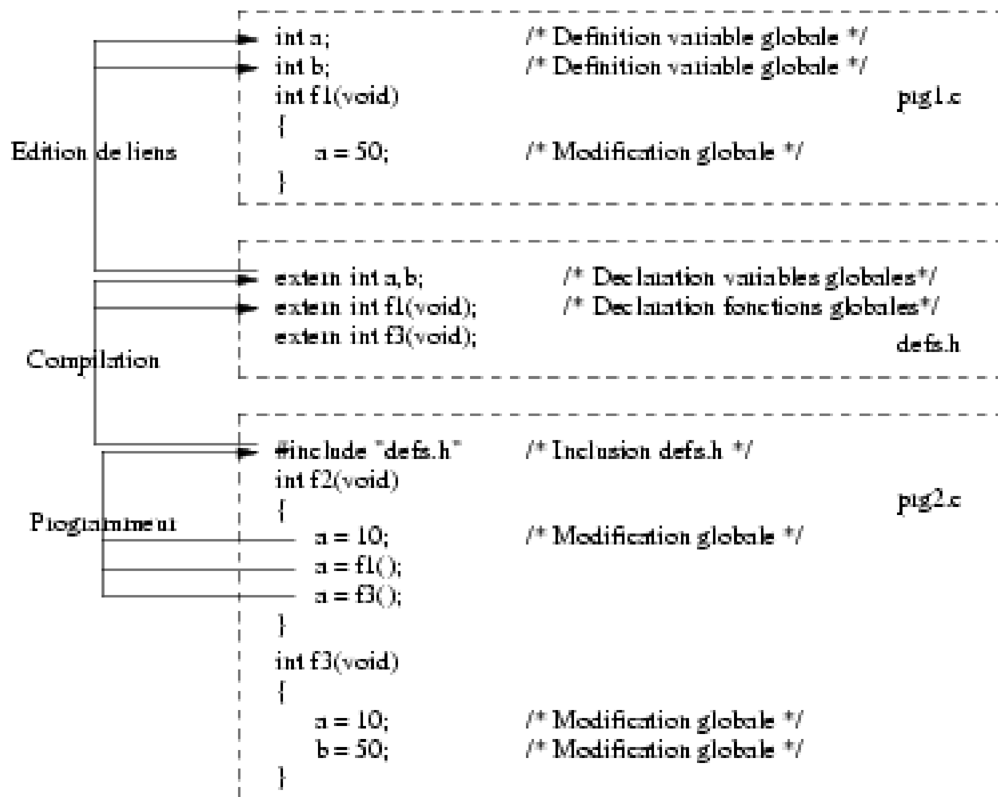

FIG. 9.7 – Utilisation de fichier d'inclusion

– prototypes de fonctions.

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

La figure 9.7 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 un mécanisme qui permet de restreindre l'accès de variables ou de fonctions pour éviter les effets de bords et permettre une meilleure utilisation des interfaces entre modules sans permettre l'accès aux fonctionnement interne d'un module.

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 du module 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

©Christian Bac 1985-2003

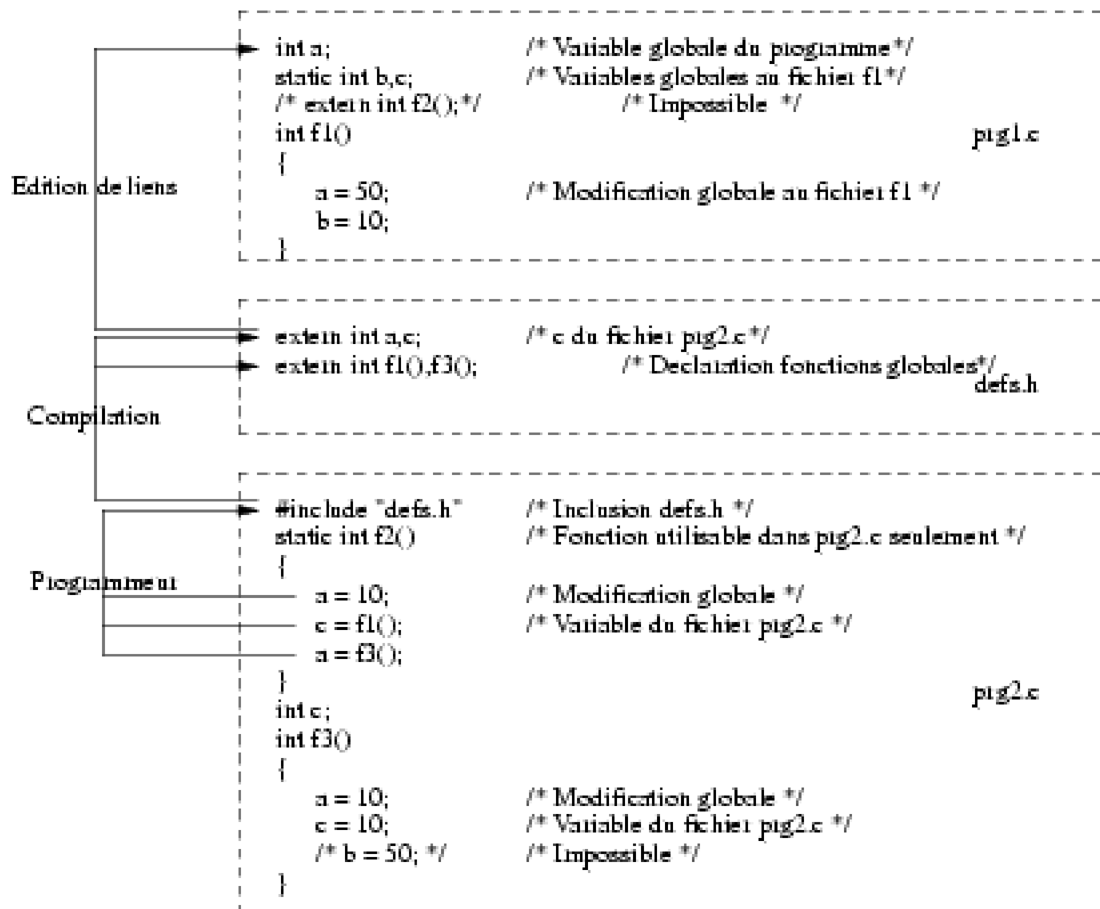

FIG. 9.8 – Réduction de visibilité

chacun d'entre-eux.

Comme le montre la figure 9.8, 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. 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 le surcoût causé par la création et la destruction de l'espace mémoire pour stocker ces variables de grande taille à l'entrée et à la sortie de la fonction.

L'isolation de la variable est sémantique, son nom n'étant pas connu du reste du programme. Cette isolation n'est pas totale en effet :

– si une fonction retourne un pointeur contenant l'adresse d'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

©Christian Bac 1985-2003

```
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. 9.9 – Variables locales statiques

non à l'exécution. Une variable locale statique peut aussi être modifiée par des effets de bord (par exemple un débordement de tableau peut écraser la variable statique locale que l'éditeur de lien a placé après le tableau) ;

– 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 9.9 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.

9.8.4 Travailler en groupe

La compilation séparée permet le découpage d'un programme en parties développées séparément par des programmeurs, c'est un début mais pour permettre un travail efficace de groupe il faut quelques règles supplémentaires.

Le projet GNU est un bon exemple quant-à l'utilisation de ce type de règles, vous trouverez les règles utilisées par les groupes de programmeurs de ce projet à l'adresse suivante :
`http ://www.gnu.org/prep/standards`.

9.9 Exercices sur les fonctions et la visibilité des variables

9.9.1 Exercice 1 : simulation d'un ascenseur

Le programme que vous devez réaliser simule une partie du fonctionnement d'un ascenseur. Il n'est pas dans ses buts de simuler complètement un fonctionnement d'ascenseur.

L'immeuble dans lequel se trouve cet ascenseur est constitué de dix étages et trois sous-sols. L'ascenseur dessert ces étages et reconnait les étages en sous-sol car ils correspondent à des chiffres négatifs.

Au départ, l'ascenseur se situe à l'étage 0.

©Christian Bac 1985-2003

Il y a trois étapes fondamentales dans le fonctionnement de l'ascenseur :
- l'appel de l'ascenseur à un étage ;
- l'entrée dans l'ascenseur suivie de la sélection de l'étage désiré ;
- le déplacement de l'ascenseur qui peut se faire entre l'appel et l'entrée ou entre la selection et la descente.

La marche de l'ascenseur que vous devez réaliser consiste en :
- demander le numéro de l'étage ou' l'on appelle l'ascenseur et le lire ;
- afficher le déplacement de l'ascenseur étage par étage avec une temporisation de 2 secondes entre chaque étage. La temporisation est obtenue par l'appel système `sleep()` ;
- signaler l'arrivée à l'étage de l'appel ;
- demander l'étage désiré et le lire ;
- se déplacer vers l'étage sélectionné.

Pour sortir de la boucle, entrer un numéro d'étage particulier (par exemple 100).

Le programme sera structuré de la manière suivante :
- une fonction `deplacement()`, qui accepte en argument le numéro de l'étage désiré et retourne le numéro de l'étage atteint ;
- une fonction `appel()`, qui réalise la lecture de l'étage de la demande ;
- une fonction `saisie()`, qui réalise la lecture de l'étage à atteindre ;
- une boucle dans la fonction `main()` qui controle la validité des saisies et termine le programme dans le cas où l'étage désiré est égal à 11.

9.9.2 Exercice 2 : racines d'une équation du deuxième degré

Calcul des racines d'une équation du deuxième degré. Une équation du deuxième degré se présente sous la forme : $a \cdot x^2 + b \cdot x + c$.

Le programme contient 4 fonctions :
- calcul du discriminant ;
- calcul de la racine double ;
- calcul des racines réélles ;
- calcul des racines complexes.

le programme principal saisit les coefficients `a`, `b` et `c`, il calcule le discriminant et aiguille vers le bon sous-programme.

9.9.3 Exercice 3 : utilisation des fichiers d'inclusion

Reprendre l'exercice en mettant les signatures (prototypes) des fonctions dans un fichier d'inclusion. Le fichier contenant les fonctions reste inchangé par rapport au corrigé 9.6.

©Christian Bac 1985-2003

Programme 9.4 Suggestion de corrigé chapitre 9 exercice 1 fonctions

```
 1 #include <stdio.h>
 2 /* PROGRAMME DE SIMULATION DE LA MARCHE D UN ASCENSEUR */
 3 int
 4 appel (){
 5   int etg;
 6   printf ("Appel ascenseur\n");
 7   printf ("A quel etage etes-vous? de -3 a 10 fin : 11\n");
 8   scanf ("%d", &etg);
 9   return etg;
10 }
11
12 int
13 selection (){
14   int selec;
15   printf ("Selection etage ? de -3 a 10\n");
16   scanf ("%d", &selec);
17   return (selec);
18 }
19
20 int
21 deplacement (int a, int asc){
22   /* si etage demande > etage d appel */
23   if(a > asc){
24     do{
25       printf ("Etage : %d\n", asc);
26       sleep (2);
27     } while(++asc < a);
28     printf("Arrivee etage %d\n", asc);
29   }
30   /* si etage demande < etage d appel */
31   else{
32     do{
33       printf ("Etage : %d\n", asc);
34       sleep (1);
35     }while(--asc > a);
36     printf("Arrivee etage %d\n", asc);
37   }
38   return asc;
39 }
```

©Christian Bac 1985-2003

PROGRAMME 9.5 SUGGESTION DE CORRIGÉ CHAPITRE 9 EXERCICE 1 MAIN()

```
 1 #include <stdio.h>
 2
 3 extern int appel ();
 4 extern int selection ();
 5 int deplacement (int, int);
 6 int
 7 main (int argc, char *argv[], char **envp)
 8 {
 9   int etg = 0, selec;
10   /* boucle generale */
11   do{
12     /* boucle appel */
13     do
14       selec = appel ();
15     while(selec > 11 || selec < -3);
16     if(selec != 11){
17       /* deplacement vers l etage d appel */
18       etg = deplacement (selec, etg);
19       /* boucle de selection */
20       do
21   selec = selection ();
22       while(selec > 10 || selec < -3);
23       /* deplacement vers l etage destination */
24       etg = deplacement (selec, etg);
25     }
26   }
27   while(selec != 11);          /* test fin */
28   printf("Arret ascenseur\n");
29   return 0;
30 }
```

DONNÉES EN ENTRÉE

```
2 4 11
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Appel ascenseur
A quel etage etes-vous? de -3 a 10 fin : 11
Etage : 0
Etage : 1
Arrivee etage 2
Selection etage ? de -3 a 10
Etage : 2
Etage : 3
Arrivee etage 4
Appel ascenseur
A quel etage etes-vous? de -3 a 10 fin : 11
Arret ascenseur
```

©Christian Bac 1985-2003

Programme 9.6 Suggestion de corrigé chapitre 9 exercice 2 fonctions

```
 1 #include <stdio.h>
 2
 3 /* fonction de calcul du discriminant */
 4 float
 5 dis (float a, float b, float c)
 6 {
 7   return (b * b - 4 * a * c);
 8 }
 9
10 /* fonction calculant les racines reelles */
11 void
12 rac2 (float r, float r1)
13 {
14   printf ("2 racines reelles : %f et %f\n", r + r1, r - r1);
15 }
16
17 /* fonction calculant les racines complexes */
18 void
19 complex (float r, float r1)
20 {
21   printf ("2 racines complexes : %f + %f i et %f - %f i\n", r, r1, r, r1);
22 }
23
24 /* fonction calculant la racine double */
25 void
26 racd (float r)
27 {
28   printf (" racine double : %f\n", r);
29 }
```

©Christian Bac 1985-2003

PROGRAMME 9.7 SUGGESTION DE CORRIGÉ CHAPITRE 9 EXERCICE 2 MAIN()

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 
 4 extern float dis (float, float, float);
 5 extern void rac2 (float, float);
 6 extern void complex (float, float);
 7 extern void racd (float);
 8 
 9 int
10 main (int argc, char *argv[], char **envp){
11   float a, b, c, r, r1;
12   double rdis;
13   float res;
14 
15   printf ("calcul des racines de ax2 + bx + c\n\n");
16   printf ("saisie des valeurs de a b et c");
17   scanf ("%f", &a);
18   scanf ("%f", &b);
19   scanf ("%f", &c);
20   if (a == 0){
21     printf (" Equation du premier degre \n");
22     printf (" La solution est x = %f \n", -c / b);
23     return 0;
24   }
25   r = -b / (2 * a);
26   res = dis (a, b, c);
27   switch (res < 0 ? -1 : (res > 0 ? 1 : 0)){
28   case 1:
29     rdis = sqrt (res);
30     r1 = rdis / (2 * a);
31     rac2 (r, r1);
32     break;
33   case -1:
34     rdis = sqrt (-res);
35     r1 = rdis / (2 * a);
36     complex (r, r1);
37     break;
38   case 0:
39     racd (r);
40     break;
41   }
42   return 0;
43 }
```

DONNÉES EN ENTRÉE

```
2 -4 2
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
calcul des racines de ax2 + bx + c

saisie des valeurs de a b et c racine double : 1.000000
```

Programme 9.8 Suggestion de corrigé chapitre 9 exercice 3 fichier d'inclusion

```
1 extern float dis(float, float, float);
2 extern void rac2(float, float);
3 extern void complex(float, float);
4 extern void racd(float);
```

©Christian Bac 1985-2003

PROGRAMME 9.9 SUGGESTION DE CORRIGÉ CHAPITRE 9 EXERCICE 3 `MAIN()`

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include "c09c03a.h"
 4
 5 int
 6 main (int argc, char *argv[], char **envp){
 7   float a, b, c, r, r1;
 8   double rdis;
 9   float res;
10   printf ("calcul des racines de ax2 + bx + c\n\n");
11   printf ("saisie des valeurs de a b et c \n");
12   scanf ("%f %f %f", &a, &b, &c);
13   if(a == 0){
14     printf (" Equation du premier degre \n");
15     printf (" La solution est x = %f \n", -c / b);
16     return 0;
17   }
18   r = -b / (2 * a);
19   res = dis (a, b, c);
20   switch (res < 0 ? -1 : (res > 0 ? 1 : 0)){
21   case 1:
22     rdis = sqrt (res);
23     r1 = rdis / (2 * a);
24     rac2 (r, r1);
25     break;
26   case -1:
27     rdis = sqrt (-res);
28     r1 = rdis / (2 * a);
29     complex (r, r1);
30     break;
31   case 0:
32     racd (r);
33     break;
34   }
35   return 0;
36 }
```

DONNÉES EN ENTRÉE

```
2 -4 2
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
calcul des racines de ax2 + bx + c

saisie des valeurs de a b et c
 racine double : 1.000000
```

©Christian Bac 1985-2003

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 contiguë 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 possible de ne pas spécifier la taille du tableau ou (exclusif) de ne pas initialiser tous les éléments du tableau. Dans ce programme, `tb1` est défini comme un tableau de 6 entiers initialisés, et `tb2` est défini comme un tableau de 10 entiers dont les 6 premiers sont initialisés. Depuis la normalisation du langage, l'initialisation des premiers éléments d'un

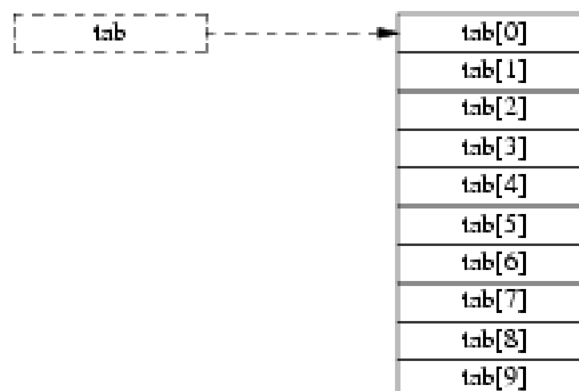

FIG. 10.1 – Tableau de dix entiers

©Christian Bac 1985-2003

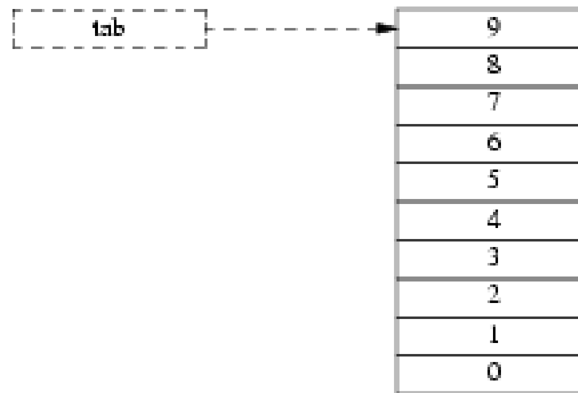

FIG. 10.2 – Tableau de dix entiers initialisé

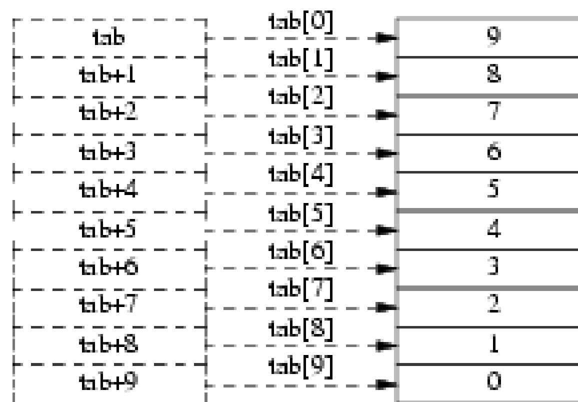

FIG. 10.3 – Adresses dans un tableau de dix entiers

tableau provoque l'initialisation de l'ensemble des éléments du tableau y compris pour les tableaux locaux. Les éléments pour lesquels des valeurs ne sont pas précisées 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.

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

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

©Christian Bac 1985-2003

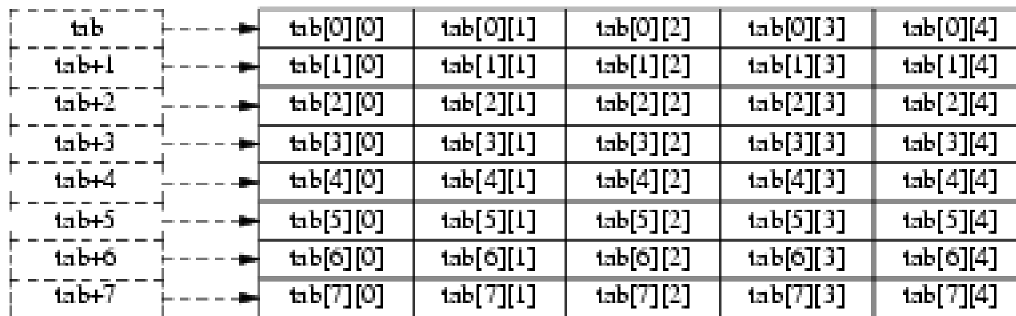

FIG. 10.4 – Tableau à deux dimensions

- il est possible d'additionner ou de soustraire un entier (n) à une adresse. Cette opération calcule une nouvelle adresse de la manière suivante :
  - l'opération suppose que l'adresse de départ et l'adresse résultante sont les adresses de deux variables contenues dans le même tableau.
  - l'opération suppose aussi que le tableau est d'une taille suffisamment grande, c'est-à-dire qu'elle suppose que le programmeur doit être conscient des risques de dépassement des bornes du tableau. Dans le cas du tableau `tab` pris en exemple dans la figure 10.3, les adresses doivent être comprises entre `&tab[0]` et `&tab[9]`. Pour une question de test de borne, l'adresse immédiatement supérieure à la fin du tableau est calculable. Il est possible d'écrire `&tab[10]` mais rien ne garantit que l'expression ne provoque pas une erreur si le rang dépasse la taille plus un. La zone mémoire correspondante au rang du tableau plus un ne doit cependant pas être accédée. Le compilateur garanti qu'il calcule correctement l'adresse mais il ne garantit pas que le fait d'essayer d'accéder à la zone mémoire correspondante soit possible.
  - 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 multidimensionnels

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];`.

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

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 d'objet. Ce type est celui des objets qui sont manipulés grâce au pointeur. L'objet peut être une variable ou une fonction.

Contrairement à ce que beaucoup d'apprentis espérent, la déclaration d'un pointeur n'implique pas la déclaration implicite d'une variable associée et l'affectation de l'adresse de la variable au pointeur. Il faut donc déclarer une variable du type correspondant et initialiser le pointeur avec l'adresse de cette variable. Par convention, l'adresse 0 est invalide et si le programme cherche à l'accéder, il obtient une erreur d'exécution du type bus-error sur UNIX. Ce comportement implique que l'utilisation de pointeurs globaux sans initialisation mène à ce résultat, car les pointeurs (comme les autres variables) déclarés en variables globales sont initialisés à 0.

Les pointeurs déclarés en variable locale (comme toutes les variables locales) ont des valeurs initiales dépendantes du contenu de la pile à cet instant, qui dépend de l'exécution précédent du programme mais correspond en général à n'importe quoi. Le comportement du programme qui utilise un pointeur local sans l'avoir affecté convenablement peut donner des comportements tels que violation de l'espace mémoire, mais parfois le pointeur reçoit une adresse valide et le programme se déroule sans erreurs flagrante mais en donnant des résultats faux (ce que d'aucuns appellent un effet de bord indésirable, ce que d'autres appellent un bug difficile à reproduire).

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 `n`^e objet à partir de l'adresse initiale ;
– soustraction de deux pointeurs du même type. Le calcul est réalisé dans les mêmes conditions que la différence entre deux adresses de variables contenues dans un même tableau. La soustraction calcule le nombre de variables entre les adresses contenues dans les pointeurs. Le résultat de type `ptrdiff_t` n'est valide que si les adresses contenues dans les deux pointeurs sont bien des adresses de variables appartenant à un même tableau, sinon le résultat est indéfini.

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));
```

Le tableau 10.2 est un exemple de soustraction à partir des définitions de variables suivantes : `int tab[20], *pt1, *p`
`ptrdiff_t i;`

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. soit le nombre d'objets entre pt2 et pt1. i contiendra 10 à la fin de cette instruction.

TAB. 10.2 – Soustraction de deux pointeurs

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 2e 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)`**. De même, **`*(pti+i)` est équivalent à `pti[i]`**.

La figure 10.5 est un exemple dans lequel sont décrites les différentes façons d'accéder au éléments d'u tableau `tab` et du pointeur `pt` 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. C'est le cas des arguments `argv` et `envp` de la fonction `main()` que nous avons eu l'occasion d'aborder dans la section 8.7 et que nous avons représentés dans la figure 8.5.

La figure 10.6 est un exemple d'utilisation de tableau de pointeurs à partir des définitions de variables suivantes :

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

10.6 Pointeurs vers un tableau

Un pointeur peut contenir l'adresse d'une variable assez 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. Nous verrons de manière plus détaillée dans

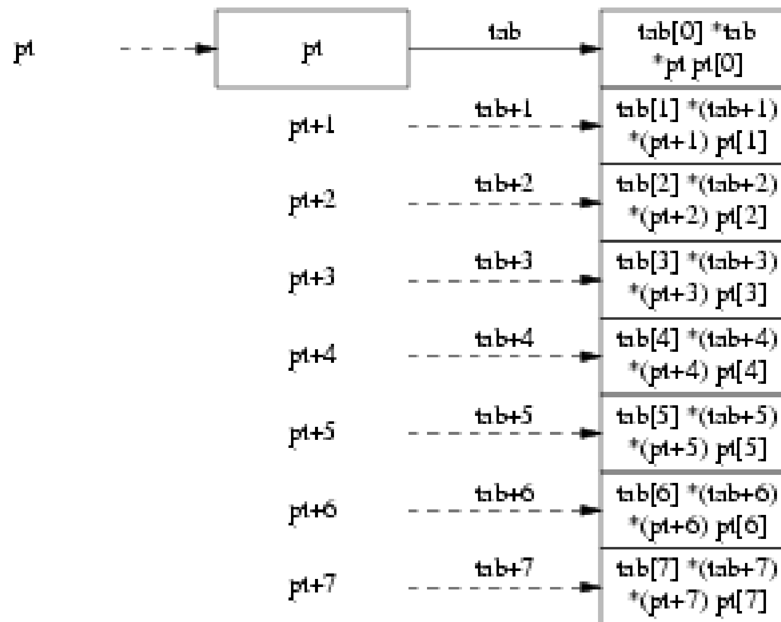

FIG. 10.5 – Pointeur et tableau

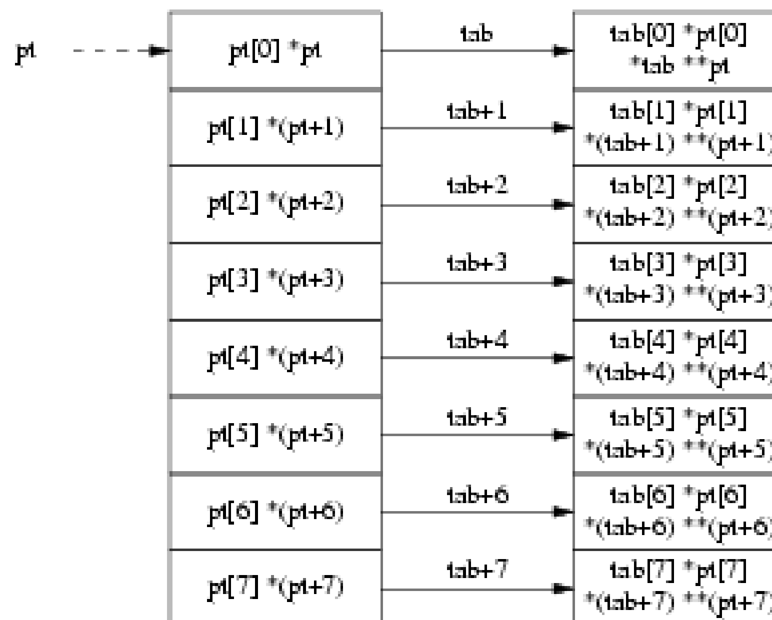

FIG. 10.6 – Tableau de pointeurs sur des variables dans un tableau

©Christian Bac 1985-2003

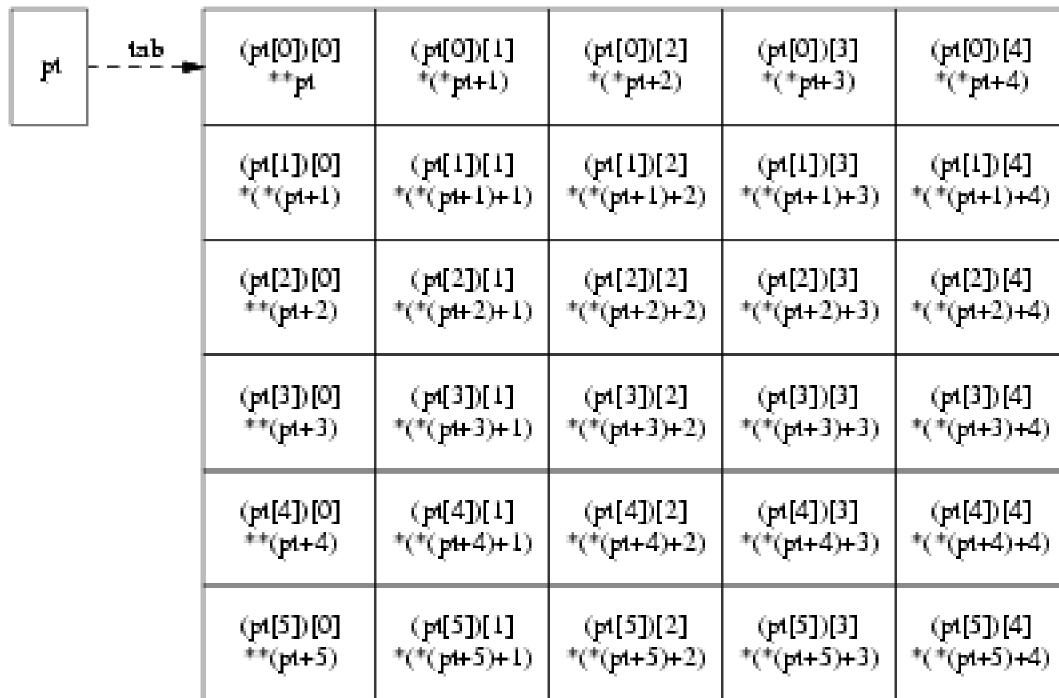

FIG. 10.7 – Accès à un tableau à deux dimensions avec un pointeur

la section 14.2 les règles de lecture et d'écriture d'une déclaration complexe, en suivant ces règles, un pointeur sur un tableau pris dans sa globalité se déclare par : `type (*nom)[taille];`

La figure 10.7 décrit les différentes façons d'accéder aux variables d'un tableau de six fois cinq entiers (certains diraient un tableau de 6 lignes et 5 colonnes) à partir d'un pointeur qui peut contenir une adresse de sous-tableau (de cinq entiers).

```
int tab[6][5]; tableau de 6 fois 5 entiers.
int (*pt)[5]= tab;
```

Le programme 10.2 montre les différentes manières d'accéder aux éléments du tableau défini ci-dessus par le pointeur associé.

10.7 Exercices sur les tableaux et les pointeurs

10.7.1 Exercice 1 : tri de tableaux d'entiers

L'exercice consiste à lire des entiers dans un tableau, séparer les nombres pairs et les impairs en les mettant dans deux tableaux, puis trier chacun des deux tableaux.

Les tableaux ont une taille arbitraire de 100. La lecture des entiers se termine lors de la saisie de 0, cet entier est mis dans le tableau et il est consideré comme une sentinelle de fin de tableau.

La séparation entre nombres pairs et impairs est faite dans une fonction qui reçoit en arguments les adresses des trois tableaux (adresse de l'élément de rang 0 du tableau), cette fonction termine les tableaux pairs et impairs par des zéros.

Le tri de chaque tableau est realisé par une fonction `tri(int *)` qui utilise un algorithme de tri simple, un exemple d'algorithme est donné à la fin de ce texte. La réécriture des tableaux triés permet de vérifier la validité de l'ensemble.

Le programme doit être composé d'un fichier source qui contient les fonctions etd'un autre qui contient la fonction `main()` et d'un fichier d'inclusion qui permet de faire la liaison pour le compilateur entre les deux fichiers.

Le fait de mettre des zéros en fin de tableau peut vous suggérer de manipuler ces tableaux avec des pointeurs.

Le programme 10.3 est un exemple d'algorithme de tri d'un tableau terminé par un zéro.

©Christian Bac 1985-2003

PROGRAMME 10.2 ACCÈS À UN TABLEAU À DEUX DIMENSIONS AVEC UN POINTEUR

```
 1 #include <stdio.h>
 2
 3 int
 4 main (int argc, char *argv[])
 5 {
 6   int tab[6][5];
 7   int (*pt)[5]= tab;
 8   int i,j;
 9
10   for(i=0;i<6;i++)
11     for(j=0;j<5;j++)
12       tab[i][j]= i*10+j;
13
14   for(i=2;i<6;i++)
15     for(j=3;j<5;j++){
16       printf("tab[%d][%d] = %d\t",i,j,tab[i][j]);
17       printf("%d\t",*(tab[i]+j));
18       printf("%d\t",*(*(tab+i)+j));
19       printf("%d\t",*(*(pt+i)+j));
20       printf("%d\t",*(pt[i]+j));
21       printf("%d\n",pt[i][j]);
22     }
23   return 0;
24 }
25
26
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
tab[2][3] = 23 23 23 23 23 23
tab[2][4] = 24 24 24 24 24 24
tab[3][3] = 33 33 33 33 33 33
tab[3][4] = 34 34 34 34 34 34
tab[4][3] = 43 43 43 43 43 43
tab[4][4] = 44 44 44 44 44 44
tab[5][3] = 53 53 53 53 53 53
tab[5][4] = 54 54 54 54 54 54
```

©Christian Bac 1985-2003

Programme 10.3 Algorithme d'un tri simple

```
i = 0;
Tant que tab[i]
faire
  j = i + 1;
  Tant que tab[j]
  faire
    si tab[j] est inferieur a tab[i]
     alors faire
       echanger tab[j] et tab[i]
     fait
   finsi
   j++;
  fait
  i++;
fait
```

Programme 10.4 Suggestion de corrigé chapitre 10 exercice 1 fichier d'inclusion

```
1 extern int saisie(int []); /* fonction de saisie de tableau
2                             *  retourne le nombre d'entiers saisis
3                             */
4 extern void tri(int []);  /* fonction de tri de tableau pointe */
5 extern void imptab(int [],int); /* fonction d'impression du tableau */
6 extern int separ(int [],int [],int []); /*separation pairs/impairs */
```

©Christian Bac 1985-2003

Programme 10.5 Suggestion de corrigé chapitre 10 exercice 1 fonctions

```
 1 #include <stdio.h>
 2
 3 int               /* fonction de saisie du tableau a manipuler */
 4 saisie (int ti[]){
 5   int *ptab = ti;
 6   printf (" Entrer des entiers \n");
 7   printf (" Terminer la liste avec 0\n\n");
 8   do
 9     scanf ("%d", ptab);
10   while (*ptab++ != 0);
11   return ptab - ti;
12 }
13
14 int        /* separation des pairs et des impairs en deux tableaux */
15 separ (int t[], int tp[], int ti[]){
16   register int *pttab = t, *ptimp = ti, *ptpair = tp;
17   while (*pttab)
18     if (*pttab & 0x01) *ptimp++ = *pttab++;
19     else  *ptpair++ = *pttab++;
20   *ptimp = *ptpair = 0;    /* chien de garde de fin de tableau */
21   return pttab - t;    /* nombre d entiers traites */
22 }
23
24 void                          /* fonction d'impression du tableau */
25 imptab (int tri[], int type){
26   register int *ptri = tri, i;
27   if (type == 0)
28     printf ("\n\tImpression du tableau initial \n");
29   else if (type == 1)
30     printf ("\n\tImpression du tableau impair \n");
31   else
32     printf ("\n\tImpression du tableau pair \n");
33   i = 0;
34   while (*ptri){
35       printf (" %6d ", *ptri++);
36       if (!*ptri)  break;
37       if (i++ && !(i % 8)) printf ("\n");
38     }
39 }
```

PROGRAMME 10.6 SUGGESTION DE CORRIGÉ CHAPITRE 10 EXERCICE 1 `MAIN`

```
 1 #include <stdio.h>
 2 #include "c10c01a.h"
 3 /* programme principal realisant les operations suivantes :
 4  *  - saisie du tableau
 5  *  - separation des nombres pairs et impairs
 6  *  - appel de la fonction de tri
 7  *  - appel de la fonction impression.
 8  */
 9 int tab[100], imp[100], pair[100];
10 int
11 main (int argc, char *argv[], char **envp){
12   register int count;   /* definition de 3 pointeurs */
13   /* saisie du tableau a manipuler */
14   count = saisie (tab);
15   printf (" Vous avez saisi %d nombres dans le tableau\n", count);
16   /* separation des pairs et des impairs en deux tableaux */
17   count = separ (tab, pair, imp);
18   printf (" Nous avons séparé %d nombres dans le tableau\n", count);
19   tri (tab);      /* tri et impression */
20   imptab (tab, 0);
21   tri (imp);      /* tri et impression */
22   imptab (imp, 1);
23   tri (pair);     /* tri et impression */
24   imptab (pair, 2);
25   return 0;
26 }
```

DONNÉES EN ENTRÉE

```
1 -2 4 7 -6 29 123 -345 -12 3 0
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer des entiers
 Terminer la liste avec 0

 Vous avez saisi 11 nombres dans le tableau
 Nous avons séparé 10 nombres dans le tableau

Impression du tableau initial
   -345     -12      -6      -2       1       3       4       7
     29     123
Impression du tableau impair
   -345       1       3       7      29     123
Impression du tableau pair
    -12      -6      -2       4
```

Programme 10.7 Suggestion de corrigé chapitre 10 exercice 1 fonction de tri

```
 1 void
 2 tri (int ti[]){    /* fonction de tri de tableau pointe par ptint */
 3   register int tamp, *ptrj, *ptint = ti;
 4   while(*ptint){
 5     ptrj = ptint + 1;
 6     while(*ptrj){
 7       if(*ptrj < *ptint){
 8   tamp = *ptint;
 9   *ptint = *ptrj;
10   *ptrj = tamp;
11       }
12       ptrj++;
13     }
14     ptint++;
15   }
16 }
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 11

Structures

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

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

11.1 Définition

Le langage C permet de définir des modèles de structures comme les autres langages évolués. Cela se fait selon la définition donnée par l'encart suivant.

```
struct   nom_de_structure    {
         type1 nom_champ1 ;
         type2 nom_champ2 ;
         type3 nom_champ3 ;
         type4 nom_champ4 ;
         ...
         typeN nom_champ_N ;
}        variables ;
```

Une définition de ce type sert à définir un modèle de structure associé à un nom de modèle, qui est optionnel, et à définir des variables construites à partir de ce modèle (cette définition est optionnelle aussi).

Comme le montre l'exemple 11.1 dans les lignes 2 à 4, 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.

La définition d'une structure ne réserve pas d'espace mémoire. Il faut définir les **variables** correspondant à ce modèle de structure (ligne 6).

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 (lignes 8 à 11). Ce modèle de structure ne peut pas être référencé par la suite, puisqu'il n'a pas de nom. Il est dit "anonyme".

Programme 11.1 Définition de structures

```
 1
 2 struct date {
 3   int jour, mois, annee ;
 4 } ;
 5
 6 struct date obdate, *ptdate = &obdate;
 7
 8 struct {
 9   int jour, mois ;
10   char annee[20] ;
11 } ob1, *pt1;
```

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

TAB. 11.1 – Accès aux champs d'une structure

11.2 Utilisation

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 en utilisant le nom de la variable suivi d'un point et du nom du champ désiré : **`nom_de_variable.nom_du_champ`**.

Pour accéder aux champs d'une structure à partir d'un **pointeur** associé avec une structure, il faut remplacer le point par un moins suivi d'un supérieur (qui symbolise une fléche) : **`nom_de_variable->nom_du_champ`**.

Une fois ces définitions réalisées, nous pouvons utiliser les variables `obdate` et `ptdate` comme le montre le tableau 11.1.

En prenant les définitions de données dans le programme 11.1 dans les lignes 2 à 6, le tableau 11.1, donne les différentes possibilités d'affectation pour chaque champ.

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

Pour le compilateur, l'accès à un champ d'une structure consiste à faire un calcul de déplacement par rapport au début de la structure puis à accéder à une variable à cette adresse.

11.2.2 Opérations sur la variable dans son ensemble

La normalisation du langage C a institutionalisé 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.

Il est possible de construire une fonction qui accepte en argument ou qui retourne une structure.

En prenant les définitions de variables et de fonctions du programme 11.2, les opérations suivantes sont possibles sur les structures :

1. l'affectation d'un variable par une autre du même type comme `obdate = obdate2;`
2. le retour d'une structure par une fonction comme `obdate = newdate();`

©Christian Bac 1985-2003

Programme 11.2 Définition de structures

```
1
2 struct date {
3   int jour, mois, annee ;
4 } ;
5
6 struct date obdate, obdate2e;
7 extern struct date newdate() ;
8 extern int checkdate(struct date) ;
9 int resul ;
```

3. le passage en argument d'une structure 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 le programme 11.3.

Programme 11.3 Structures et listes chaînées

```
1
2 struct noeud {
3   int val ;
4   struct noeud *suiv ;
5   struct noeud *pred ;
6 } node, tab[100] , *ptnoeud = tab ;
```

Note sur les pointeurs de structures : nous avons vu dans le chapitre précédent sur l'aritméthique des adresses que l'incrémentation d'un pointeur le faisait passer sur l'élément suivant lorsque le pointeur contient une adresse qui correspond à un tableau d'éléments correspondants au type de variable associé au pointeur. Il en est de même pour un pointeur qui contient une adresse de structure. Un pointeur incrémenté permet l'accès à la structure suivante lorsque les structures sont dans un tableau.

Reprenons le programme 11.3 dans lequel nous avons défini un tableau de structures et un pointeur sur la structure noeud.

Aprés la définition 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 à un découpage des octets pour obtenir des variables qui sont décrites sur un ensemble d'éléments binaires dont la taille est inférieure à celle de l'octet.

Il est en effet 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 de l'encart suivant.

©Christian Bac 1985-2003

```
struct  nom_de_structure      {
        unsigned nom_champ1    : longueur1 ;
        unsigned nom_champ2    : longueur2 ;
        unsigned nom_champ3    : longueur3 ;
        unsigned nom_champ4    : longueur4 ;
        ...
        unsigned nom_champ_N   : longueurN ;
}                             objets ;
```

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

Le programme 11.4 donne quelques exemples de définitions de modèles de structures correspondant à des champs de bits :

Programme 11.4 Structures et champs de bits

```
 1 struct mot
 2 {
 3   unsigned sign:1;
 4   unsigned val:15;
 5 };
 6
 7 struct flottant
 8 {
 9   unsigned exposant:7;
10   unsigned signe:1;
11   unsigned mantisse:24;
12 };
13
14 struct mixte
15 {
16   unsigned exposant:7;
17   unsigned signe:1;
18   unsigned mantisse:24;
19   unsigned comp:7;
20   unsigned : 9;
21 };
```

11.5 Exercices sur les structures

11.5.1 Exercice 1

Définir une structure date contenant trois champs de type entier pour identifier le jours, le mois et l'année.

Initialiser une variable de type structure date.

Imprimer cette structure :
- à l'aide de la variable.
- à l'aide d'un pointeur.

©Christian Bac 1985-2003

PROGRAMME 11.5 SUGGESTION DE CORRIGÉ CHAPITRE 11 EXERCICE 1

```
 1 #include <stdio.h>
 2 struct date      /* declaration du modele de structure date */
 3 {
 4   int jour;
 5   int mois;
 6   int annee;
 7 };
 8
 9 int
10 main (int argc, char *argv[], char **envp)
11 {
12   struct date dat, *ptdate = &dat;
13   /* saisie de la date */
14   printf (" Entrez une date ( jj mm aaaa ) :");
15   scanf ("%d %d %d", &dat.jour, &dat.mois, &dat.annee);
16   /* impression de la date */
17   printf (" La date est : %d\t%d\t%d\n", dat.jour, dat.mois, dat.annee);
18   /* impression de la date avec le pointeur */
19   printf (" La date est : %d\t%d\t%d\n", ptdate->jour, ptdate->mois,
20     ptdate->annee);
21   return 0;
22 }
```

DONNÉES EN ENTRÉE

```
01 04 2003
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrez une date ( jj mm aaaa ) : La date est : 1 4 2003
 La date est : 1 4 2003
```

11.5.2 Exercice 2

Définir un tableau de structures date.

Définir un pointeur sur ce tableau.

Initialiser ce tableau.

Imprimer le contenu du tableau.

©Christian Bac 1985-2003

PROGRAMME 11.6 SUGGESTION DE CORRIGÉ CHAPITRE 11 EXERCICE 2

```
 1 #include <stdio.h>
 2 /* declaration du modele de structure date */
 3 struct date{ int jour, mois, annee;};
 4 int
 5 main (int argc, char *argv[], char **envp)
 6 {
 7   struct date dat[5], *ptdate=dat;
 8   int i = 0;
 9   /* remplissage du tableau */
10   printf ("Entrez 5 dates au format ( jj mm aaaa )\n");
11   while (i < 5){
12       scanf ("%d %d %d", &dat[i].jour, &dat[i].mois, &dat[i].annee);
13       i++;
14   }
15   /* impression du tableau */
16   for (i = 0; i < 5; i++, ptdate++) {
17       /* sans pointeur */
18       printf (" Date numero %d : %d\t%d\t%d\n", i + 1, dat[i].jour,
19         dat[i].mois, dat[i].annee);
20       /* avec pointeur */
21       printf (" Date numero %d : %d\t%d\t%d\n", i + 1, ptdate->jour,
22         ptdate->mois, ptdate->annee);
23   }
24   return 0;
25 }
```

DONNÉES EN ENTRÉE

```
01 04 2003
03 05 2003
09 08 2003
02 09 2003
11 11 2003
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Entrez 5 dates au format ( jj mm aaaa )
 Date numero 1 : 1 4 2003
 Date numero 1 : 1 4 2003
 Date numero 2 : 3 5 2003
 Date numero 2 : 3 5 2003
 Date numero 3 : 9 8 2003
 Date numero 3 : 9 8 2003
 Date numero 4 : 2 9 2003
 Date numero 4 : 2 9 2003
 Date numero 5 : 11 11 2003
 Date numero 5 : 11 11 2003
```

©Christian Bac 1985-2003

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

La définition d'une union respecte une syntaxe proche de celle d'une structure et qui est donnée dans l'encart suivant.

```
union   nom_de_union        {
        type1 nom_champ1 ;
        type2 nom_champ2 ;
        type3 nom_champ3 ;
        type4 nom_champ4 ;
        ...
        typeN nom_champ_N ;
}       variables ;
```

Le programme 12.1 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.

Programme 12.1 Utilisation d'une union

```
1 union zone {
2   int entier;
3   long entlong;
4   float flottant;
5   double flotlong;
6 } 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`.

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 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, par exemple, utilisées dans les compilateurs.

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

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

PROGRAMME 12.2 UTILISATION D'UNE UNION

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   union etiq
 6   {
 7     short int ent;
 8     unsigned short int ent1;
 9   } lmot = { -1, };
10   printf ("Valeur de lmot.ent : %hd\n", lmot.ent);
11   printf ("Valeur de lmot.ent1 : %hd\n", lmot.ent1);
12   return 0;
13 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

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

Dans l'exemple 12.2, 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
```

12.3 Exercices sur les unions et les champs de bits

12.3.1 Exercice 1

Définir une union pouvant contenir une variable des types suivants : entier, réel et caractère.

Pour chacun des membres de l'union :

– lire une valeur correspondante au clavier ;
– afficher l'adresse du membre ;
– puis afficher son contenu.

PROGRAMME 12.3 SUGGESTION DE CORRIGÉ CHAPITRE 12 EXERCICE 1

```
 1 #include <stdio.h>
 2 union test { /* definition du modele de l'union */
 3   char un_char;
 4   int  un_int;
 5   float un_float;
 6 } ;
 7 int
 8 main (int argc, char *argv[], char **envp)
 9 {
10   union test un_test;
11   /* saisie du caractere et impression */
12   printf(" Entrer un caractere : ");
13   scanf("%c",&un_test.un_char);
14   printf(" caractere : %c \t adresse : %x \n",
15    un_test.un_char, &un_test.un_char);
16   /* saisie de l'entier et impression */
17   printf(" Entrer un entier : ");
18   scanf("%d",&un_test.un_int);
19   printf(" entier : %d \t adresse : %x \n",
20    un_test.un_int, &un_test.un_int);
21   /* saisie du flottant et impression */
22   printf(" Entrer un flottant : ");
23   scanf("%f",&un_test.un_float);
24   printf(" flottant : %f \t adresse : %x \n",
25    un_test.un_float, &un_test.un_float);
26   return 0;
27 }
```

DONNÉES EN ENTRÉE

```
c 22 3.14159
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrer un caractere :  caractere : c   adresse : bffff4c4
 Entrer un entier :  entier : 22   adresse : bffff4c4
 Entrer un flottant :  flottant : 3.141590   adresse : bffff4c4
```

12.3.2 Exercice 2

Definir un champ de bits composé de :
– 2 bits,
– 7 bits pour un premier caractère,
– 7 bits pour un deuxième caractère,
– 16 bits pour un entier.
Initialiser ce champ de bits.

Imprimer ce champ en hexadécimal.

Imprimer les différentes composantes de ce champ.

PROGRAMME 12.4 SUGGESTION DE CORRIGÉ CHAPITRE 12 EXERCICE 2

```
 1 #include <stdio.h>
 2 struct champ_bit {         /* definition du champ de bits */
 3   unsigned left : 2;
 4   unsigned car1 : 7;
 5   unsigned car2 : 7;
 6   unsigned nbre : 16;
 7 } champ = { 01, 'a', 'Z', 4532 }; /* initialisation */
 8 int
 9 main (int argc, char *argv[], char **envp)
10 {
11   /* impression du champ globalement */
12   printf(" Valeur du champ : %x\n", champ);
13   /* impression de chaque partie du champ */
14   printf(" Caractere 1 : %c\n", champ.car1);
15   printf(" Caractere 1 : %x\n", champ.car1);
16   printf(" Caractere 2 : %c\n", champ.car2);
17   printf(" Caractere 2 : %x\n", champ.car2);
18   printf(" Nombre : %d\n", champ.nbre);
19   printf(" Nombre : %x\n", champ.nbre);
20   printf(" Bits de gauche : %x\n", champ.left);
21   return 0;
22 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Valeur du champ : 11b4b585
 Caractere 1 : a
 Caractere 1 : 61
 Caractere 2 : Z
 Caractere 2 : 5a
 Nombre : 4532
 Nombre : 11b4
 Bits de gauche : 1
```

©Christian Bac 1985-2003

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 pré-processeur dans la description de constantes (voir chap. 15).

13.1 Définition

La définition d'une énumération respecte la syntaxe donnée dans l'encart suivant.

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

Les différents énumérateurs sont des constantes symboliques. Les valeurs associées aux énumérateurs 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 ; si l'énumérateur suivant n'est pas associé à une valeur, la progression de 1 reprend à la valeur courante.

Nous baserons nos exemples sur le programme 13.1.

Les lignes 2 et 3 de cet exemple d'énumération définissent 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.

Elles définissent la variable `rvb` construite sur ce modèle d'énumération. Cette partie utilise les possibilités standard des énumérations.

©Christian Bac 1985-2003

Programme 13.1 Utilisation d'énumérations

```
 1 #include <stdio.h>
 2 enum couleurs {
 3   rouge, vert, bleu
 4 } rvb;
 5 enum autrescouleurs {
 6   violet=4, orange, jaune=8
 7 } arcenciel;
 8 enum typefic {
 9   normal = 0100000,
10   repertoire = 0040000,
11   special_bloc = 0060000,
12   special_car = 0020000,
13   sock_unix = 0140000,
14   tube_nomme = 0010000,
15   lien_symb = 0120000
16 }tfic;
17 int
18 main (int argc, char *argv[])
19 {
20   rvb = 12 ;
21   arcenciel = violet + jaune;
22   if(rvb == arcenciel)
23     tfic =  repertoire;
24   else
25     tfic =  normal;
26     rvb = sock_unix;
27   return 0;
28 }
```

Les lignes 5 à 7 fixent les valeurs de certains énumérateurs eet définissent les constantes symboliques suivantes :

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.

Elles définissent la variable `arcenciel` construite sur le modèle d'énumération `autrescouleurs`.

Les lignes 8 à 16 montrent une utilisation des énumérations pour différencier les types de fichiers dans les inodes d'un système de fichiers de type UNIX.

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. Ceci présente l'avantage de pouvoir réaliser des expressions combinant des variables de type énuméré avec des entiers. Elles permettent d'utiliser des constantes symboliques. Les variables définies à partir de type énuméré sont cependant traitées comme 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 les lignes 20, 22 et 26 du programme 13.1 sans que le compilateur proteste. Dans ces lignes nous mélangeons les constantes symboliques appartenant à des types d'énumération différents sans avoir de message d'avertissement du compilateur.

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 14

Types synonymes et complexes

Ce chapitre aborde la définition de types 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 ce mot réservé 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 pré-processeur (chap. 15) mais il s'adresse au compilateur.

Attention : `typedef` ne réserve pas d'espace mémoire. Les noms sont des types ; ils sont donc inaccessibles comme variables.

L'utilisation de `typedef` permet de faciliter les définitions de prototypes pour permettre aux programmeurs d'écrire du code plus lisible.

©Christian Bac 1985-2003

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`.

```
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 l'adresse d'une variable du type `date` mais qui dans l'exemple n'est pas initialisé.

14.2 Types complexes

Nous allons nous intéresser aux types complexes sous l'angle du lecteur d'un programme écrit par un autre rédacteur. Ainsi notre premier objectif est de pouvoir comprendre une ligne comprenant des déclarations ou des définitions de variables qui semblent à première vue complexes.

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 du tableau de précédence qui nous intéressent dans cette entreprise sont celles qui portent sur les parenthèses correspondant aux regroupement ou aux fonctions, aux crochets correspondant aux tableaux, et aux étoiles correspondant aux pointeurs. Parmi ces quatre opérateurs les règles de précédences sont les suivantes :
- `()` représentant le regroupement est de priorité la plus élevée et une associativité de gauche à droite ;
- `()` correspondant à l'identification d'une fonction et `[]` correspondant à la définition d'un tableau sont de priorité égale et immédiatement inférieure au regroupement avec une associativité de gauche à droite ;
- `*` est de priorité la plus basse parmi nos quatre opérateurs 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 une fonction.

```
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 dé-référencer son adresse pour provoquer un saut du pointeur de programme courant à cette adresse.

©Christian Bac 1985-2003

Déclaration	Objet
`int *tab[10];`	tableau de 10 pointeurs d'entier
`int **ta[10];`	tableau de 10 pointeurs sur des pointeurs d'entier
`int (*tb)[10];`	pointeur sur un tableau de 10 entiers
`int *(*tb)[10];`	pointeur sur un tableau de 10 pointeurs sur un entier
`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 obligatoirement externe
`int *t4[];`	tableau de pointeurs externe
`int t5[][];`	déclaration illégale il faut fixer la taille de la 2^{e}dimension
`int **t7;`	Pointeur sur un pointeur d'entier (Souvent utilisé pour décrire un tableau de pointeurs d'entier)
`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

14.4 Exercices sur les déclarations complexes

14.4.1 Exercice 1

Définir les types suivants :
- structure date telle que définie dans le programme 11.5,
- pointeur sur une structure date,
- tableau de dix objets de structure date.

PROGRAMME 14.1 SUGGESTION DE CORRIGÉ CHAPITRE 14 EXERCICE 1

```
 1 struct date  /* definition du modele de structure date */
 2 {  int jour, mois, annee; };
 3
 4 /* definition des types synonymes  */
 5 typedef struct date tdate;
 6 typedef tdate * ptdate;
 7
 8 #include <stdio.h>
 9 int
10 main (int argc, char *argv[], char **envp)
11 {
12   tdate  dat;
13   ptdate ptd=&dat;
14   /* saisie de la date */
15   printf (" Entrez une date ( jj mm aaaa ) :");
16   scanf ("%d %d %d", &dat.jour, &dat.mois, &dat.annee);
17
18   /* impression de la date */
19   printf (" La date est : %d\t%d\t%d\n", dat.jour, dat.mois, dat.annee);
20
21   /* impression de la date avec le pointeur */
22   printf (" La date est : %d\t%d\t%d\n", ptd->jour, ptd->mois, ptd->annee);
23   return 0;
24 }
```

DONNÉES EN ENTRÉE

```
01 04 2003
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
 Entrez une date ( jj mm aaaa ) : La date est : 1 4 2003
 La date est : 1 4 2003
```

14.4.2 Exercice 2

Définir un type correspondant à la définition suivante :

Une union pouvant contenir :
- un tableau de 10 pointeurs sur des entiers,
- un tableau de 10 pointeurs sur des flottants,
- un tableau de 10 entiers,
- un tableau de 10 flottants.

©Christian Bac 1985-2003

14.4.3 Exercice 3

Cet exercice a pour but de manipuler le remplissage d'un tableau de structures date et de permettre son édition, à partir d'un menu en utilisant l'appel de fonction à travers des pointeurs.

Pour cela il faut écrire trois fonctions :
- une fonction de saisie,
- une fonction de consultation,
- une fonction de permettant de sortir de l'application.

La fonction `main()` contient une boucle infinie donnant un menu qui offre le choix entre saisie d'une structure, édition et arrêt.

©Christian Bac 1985-2003

PROGRAMME 14.2 SUGGESTION DE CORRIGÉ CHAPITRE 14 EXERCICE 2

```
 1 /* declaration des differents types de tableaux */
 2 typedef int int_10[10];
 3 typedef int *ptint_10[10];
 4 typedef float float_10[10];
 5 typedef float *ptfloat_10[10];
 6 /* definition des differents objets pouvant etre contenus dans l'union */
 7 union u1 {
 8   int_10 o1;
 9   ptint_10 o2;
10   float_10 o3;
11   ptfloat_10 o4;
12 };
13 /* definition du type union */
14 typedef union u1 utab;
15 /* definition d'un objet de ce type */
16 utab my_union ;
17 #include <stdio.h>
18 int
19 main (int argc, char *argv[], char **envp){
20   int i;
21   /* initialisation des entiers avec leur rang  */
22   for(i=0; i < 5;i++ )  my_union.o1[i] =  i;
23   /* impression du tableau */
24   for(i=0; i < 5;i++ )
25     printf("my_union.o1[%d] =  %d \n", i, my_union.o1[i]);
26   /* initialisation des pointeurs avec leurs propres adresses */
27   for(i=0; i < 5;i++ )
28     my_union.o2[i] =  & my_union.o1[i];
29   /* impression du tableau de pointeurs */
30   for(i=0; i < 5;i++ )
31     printf("my_union.o2[%d] =  %x \n", i, my_union.o2[i]);
32   return 0;
33 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
my_union.o1[0] =  0
my_union.o1[1] =  1
my_union.o1[2] =  2
my_union.o1[3] =  3
my_union.o1[4] =  4
my_union.o2[0] =  80495a0
my_union.o2[1] =  80495a4
my_union.o2[2] =  80495a8
my_union.o2[3] =  80495ac
my_union.o2[4] =  80495b0
```

©Christian Bac 1985-2003

Programme 14.3 Suggestion de corrigé chapitre 14 exercice 3 fichier d'inclusion

```
1 struct date {
2   int jour, mois, annee;
3 };
4
5 void saisie(struct date *);
6 void edition(struct date *);
7 void sortie();
```

Programme 14.4 Suggestion de corrigé chapitre 14 exercice 3 fonctions

```
 1 #include <stdio.h>
 2 #include "c14c03.h"
 3
 4 void saisie(struct date * ptd)
 5 {
 6   int i;
 7   /* rang dans le tableau de dates */
 8   printf(" Entrez le rang dans le tableau date :");
 9   scanf("%d", &i);
10   ptd += i;
11   /* saisie de la date */
12   printf(" Entrez une date ( jj mm aaaa ) :");
13   scanf("%d%d%d", &ptd->jour, &ptd->mois, &ptd->annee);
14 }
15 void edition(struct date * ptd)
16 {
17   int i;
18   /* rang dans le tableau de dates */
19   printf("Entrez le rang dans le tableau date :");
20   scanf("%d", &i);
21   ptd += i;
22   /* edition de la date */
23   printf("Date dans le tableau :");
24   printf("%d %d %d\n", ptd->jour, ptd->mois, ptd->annee);
25 }
26
27 void sortie()
28 {  exit(0); }
```

©Christian Bac 1985-2003

PROGRAMME 14.5 SUGGESTION DE CORRIGÉ CHAPITRE 14 EXERCICE 3

```
 1 #include <stdio.h>
 2 #include "c14c03.h"
 3
 4 static struct date dat[10];
 5
 6 int
 7 main (int argc, char *argv[], char **envp)
 8 {
 9   int i;
10   static void (*f[3])() = { saisie, edition, sortie };
11
12   for(;;) {
13     printf("\tsaisie\t1\n");
14     printf("\tedition\t2\n");
15     printf("\tarret\t3\n");
16     printf("Choix :");
17     scanf("%d",&i);
18     f[i-1](dat);
19     /* on peut aussi ecrire : (*f[i-1])(dat); */
20   }
21 }
```

DONNÉES EN ENTRÉE

```
1 1 1 4 2003
2 1
1 2 12 08 1998
3
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
saisie 1
edition 2
arret 3
Choix : Entrez le rang dans le tableau date : Entrez une date ( jj mm aaaa ) :
saisie 1
edition 2
arret 3
Choix :Entrez le rang dans le tableau date :Date dans le tableau :1 4 2003
saisie 1
edition 2
arret 3
Choix : Entrez le rang dans le tableau date : Entrez une date ( jj mm aaaa ) :
saisie 1
edition 2
arret 3
Choix :
```

©Christian Bac 1985-2003

Chapitre 15

Préprocesseur

Comme nous l'avons dit dans la section 1.4, le pré-processeur ou pré-compilateur (alias C Pre Processor ou CPP) 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 en utilisant la directive `#include`
- définition de variables de précompilation :
  - `#define NOM valeur`
  - `#undef NOM`
- définition de macro-fonction ou macro-expression : `#define m(x) (128*(x)+342*(x)*(x))`
- sélection du code en fonction des variables du pré-processeur :
  - `#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 pré-processeur 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 erreurs de compilation comme le montre le programme 15.2.

Le tableau 15.1 montre le retrait des commentaires dans un fichier source avant le passage vers le compilateur.

La figure15.2 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 pré-processeur est déclenchée par la rencontre de la directive :

PROGRAMME 15.1 TRAITEMENT DES COMMENTAIRES

```
 1 int
 2 main (int argc, char *argv[])
 3 {
 4   int a,b,c;
 5   a=1;
 6   b=1;
 7   c=1;
 8   /* ajout a+b a c */
 9   c += a +b ;
10   a=2;
11   return 0;
12 }
```

SOURCE APRÈS PRÉCOMPILATION

```
 1 # 1 "c15e01.c"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "c15e01.c"
 5 int
 6 main (int argc, char *argv[])
 7 {
 8   int a,b,c;
 9   a=1;
10   b=1;
11   c=1;
12
13   c += a +b ;
14   a=2;
15   return 0;
16 }
```

`#include nom_de_fichier`

Par convention, les fichiers à inclure ont des noms terminés par "`.h`" pour signifier "header".

Il existe trois façons de nommer un fichier à inclure. Ces façons déterminent l'endroit où le pré-processeur cherche 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 pré-processeur d'ajouter d'autres catalogues à sa recherche, en utilisant une option de compilation à l'appel de l'en-chaîneur de passes. Cette option est dans un en-chaîneur de

©Christian Bac 1985-2003

PROGRAMME 15.2 ERREUR DUE AU TRAITEMENT DES COMMENTAIRES

```
 1 int
 2 main (int argc, char *argv[])
 3 {
 4   int a,b,c;
 5   a=1;
 6   b=1;
 7   /* Ceci ne nous interesse plus
 8   c=1;
 9   /* ajout a+b a c */
10   c += a +b ;
11   */
12   a=2;
13   return 0;
14 }
```

SOURCE APRÈS PRÉCOMPILATION

```
 1 # 1 "c15e02.c"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "c15e02.c"
 5 int
 6 main (int argc, char *argv[])
 7 {
 8   int a,b,c;
 9   a=1;
10   b=1;
11
12
13
14   c += a +b ;
15   */
16   a=2;
17   return 0;
18 }
```

passe UNIX `-Inom_du_catalogue`. Cette option peut être utilisée plusieurs fois de manière à spécifier plusieurs catalogues de recherche.

Lorsque le fichier à inclure est spécifié entre guillemets, le pré-processeur cherche le fichier :

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

Si le nom du fichier à include est entre “<” et “>”, le pré-processeur cherche le fichier :

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

Les fichiers inclus sont traités par le pré-processeur. Ils peuvent contenir d’autres inclusions de fichiers.

©Christian Bac 1985-2003

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. 15.1 – Utilisation d'une constante de compilation

15.3 Variables de pré-compilation

Les variables de pré-compilation ont deux utilisations :
– la définition de constantes utilisées pour la compilation ;
– la définition de variables qui jouent un rôle de booléen et permettent la sélection du code.

15.3.1 Définition de constantes de compilation

Comme le montre la table 15.1, l'usage le plus courant des constantes de compilation est associé à la manipulation de tableaux. Il est plus simple et plus sûr d'avoir une constante qui soit utilisée lors de la définition et lors de la manipulation du tableau. La définition d'une constante de précompilation se fait par :

```
#define nom_de_la_variable valeur
```

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 15.3 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 comme nous le verrons dans la section 15.7.

La définition d'une variable de précompilation se fait par :

```
#define nom_de_la_variable
```

15.4 Définition de macro-expressions

CPP permet la définition de macro-expressions encore appelées macro-fonctions. Ces macro-expressions sont très semblables à celles de macro-assembleurs.

La définition d'une macro-expression se fait aussi 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 l'utilisation intensive du parenthésage en encadrant les noms des pseudo-variables par des parenthèses mais aussi en encadrant la macro-expression elle-même. Comme le montre l'exemple 15.4,

©Christian Bac 1985-2003

PROGRAMME 15.3 INTERPRÉTATION DES VARIABLES PAR LE PRÉPROCESSEUR

```
 1 #define PI 3.14159
 2 #define LG 20
 3
 4 int
 5 main (int argc, char *argv[])
 6 {
 7   int i,t[LG];
 8   for (i=0; i<LG; i++){
 9       printf("Valeur de PI \%d",t[i] = PI);
10     }
11   return 0;
12 }
```

SOURCE APRÈS PRÉCOMPILATION

```
 1 # 1 "c15e03.c"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "c15e03.c"
 5
 6
 7
 8 int
 9 main (int argc, char *argv[])
10 {
11   int i,t[20];
12   for (i=0; i<20; i++){
13       printf("Valeur de PI \%d",t[i] = 3.14159);
14     }
15   return 0;
16 }
```

le parenthésage garantit que l'expression résultante est correctement interprétée, la ligne 16 de l'exemple montre que l'expression `m(a+b)` est mal évaluée.

15.5 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.

15.6 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`.

©Christian Bac 1985-2003

– 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. Elles sont la plupart du temps d'une part fixées par la compilation de l'enchaîneur de passe lui-même ou par la détection de de l'environnement par ce dernier.

Par exemple, sur un PC au 05/04/2003 les variables suivantes sont prédéfinies par l'enchaîneur de passes :

```
__ELF__
unix
__gnu_linux__
linux
i386
__i386
__i386__
```

L'utilisation de l'option `-dumpspecs` avec `gcc` ou `cpp` donne ces informations parmi beaucoup d'autres.

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 utilisant l'option `-Unom`.

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

15.7 Sélection de code

La sélection de code permet de générer à partir d'un même fichier source des fichiers exécutables qui se comportent différemment. Cela permet en particulier de partir des même fichiers sources pour obtenir : des binaires exécutables pour des machines et des systèmes différents, ou des environnements d'exécution différents. Cela permet aussi d'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.

Le principe de base consiste à passer ou à supprimer des parties de code suivant des conditions fixées à partir des variables de précompilation.

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`. S'il n'y a pas de partie `#else` aucun code n'est passé.

©Christian Bac 1985-2003

15.7.1 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. Le programme 15.5 quelques exemples d'utilisaiton de `#if`.

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

15.7.2 Sélection avec `#ifdef` et `#ifndef`

La sélection avec `#ifdef` et `#ifndef` est semblable à celle avec `#if defined` ou `#if !defined`. Dans le programme 15.6 extrait du logiciel libre `sendmail`, 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 le montre le programme 15.7, lui aussi extrait du code de `sendmail`, les sélections peuvent être imbriquées.

15.8 Exercices sur le préprocesseur

15.8.1 Exercice 1

Définir les macros fonctions suivantes :
- valeur absolue d'une expression,
- minimum de deux expressions,
- maximum de deux expressions,
- minimum de trois expressions,
- maximum de trois expressions,
- minimum en valeur absolue de deux expressions (en utilisant valeur absolue d'une expression et en ne l'utilisant pas),
- maximum en valeur absolue de trois expressions (en utilisant valeur absolue d'une expression et en ne l'utilisant pas),

Utiliser ces définitions pour faire imprimer leurs résultats sur des entiers puis des flottants entrés au clavier. Utiliser les options -E ou -P de l'enchaineur de passes pour lire le programme C généré.

15.8.2 Exercice 2

Reprendre l'exercice sur le calcul des racines d'une équation du second degré, pour ajouter des lignes de debug imprimant la valeur du discriminant et les calculs intermédiaires.

Ces lignes doivent se trouver dans des test du préprocesseur de manière à n'être présente que lorsqu'on passe l'option -DDEBUG à la compilation.

Le fichier d'inclusion est identique à celui du programme 9.8 et les fonctions sont les mêmes que dans le programme 9.6.

©Christian Bac 1985-2003

PROGRAMME 15.4 EVALUATION DE MACROS PAR CPP

```
 1
 2 #define add(x1,x2) ((x1) += (x2))
 3 #define m(x) 128*x+342*x*x
 4 #define y(x) (128*(x)+342*(x)*(x))
 5 int
 6 main (int argc, char *argv[])
 7 {
 8   int a,b,c;
 9   int e,f,g;
10   a = b = c = 1;
11   d = e = f = 2;
12   add(a,b);
13   add(a,b+1);
14   d = m(a);
15   e = y(a);
16   d = m(a+b);
17   d = y(a+b);
18   f = m(add(a,b));
19   f = y(add(a,b));
20   return 0;
21 }
```

SOURCE APRÈS PRÉCOMPILATION

```
 1 # 1 "c15e04.c"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "c15e04.c"
 5
 6
 7
 8
 9 int
10 main (int argc, char *argv[])
11 {
12   int a,b,c;
13   int e,f,g;
14   a = b = c = 1;
15   d = e = f = 2;
16   ((a) += (b));
17   ((a) += (b+1));
18   d = 128*a+342*a*a;
19   e = (128*(a)+342*(a)*(a));
20   d = 128*a+b+342*a+b*a+b;
21   d = (128*(a+b)+342*(a+b)*(a+b));
22   f = 128*((a) += (b))+342*((a) += (b))*((a) += (b));
23   f = (128*(((a) += (b)))+342*(((a) += (b)))*(((a) += (b))));
24   return 0;
25 }
```

©Christian Bac 1985-2003

PROGRAMME 15.5 EXEMPLE D'UTILISATION DU #IF

```
 1 #define vrai 1
 2 #define faux 0
 3 #if vrai
 4 vrai_est_vrai
 5 #else
 6 vrai_est_faux
 7 #endif
 8 #if faux
 9 faux_est_vrai
10 #else
11 faux_est_faux
12 #endif
13 #if vrai || faux
14 vrai_ou_faux_est_vrai
15 #endif
16 #if !faux
17 not_faux_est_vrai
18 #endif
19 #if vrai && !faux
20 vrai_et_not_faux_est_vrai
21 #endif
```

SOURCE APRÈS PRÉCOMPILATION

```
 1 # 1 "c15e05.c"
 2 # 1 "<built-in>"
 3 # 1 "<command line>"
 4 # 1 "c15e05.c"
 5
 6
 7
 8 vrai_est_vrai
 9
10
11
12
13
14
15 faux_est_faux
16
17
18 vrai_ou_faux_est_vrai
19
20
21 not_faux_est_vrai
22
23
24 vrai_et_not_faux_est_vrai
```

©Christian Bac 1985-2003

Programme 15.6 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__ */
```

Programme 15.7 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 */
```

©Christian Bac 1985-2003

Programme 15.8 Suggestion de corrigé chapitre 15 exercice 1 définition des macros

```
 1 #define abs(a)  ((a) < 0 ? (-(a)) : (a))
 2 #define min(a1,a2) ((a1) < (a2) ? (a1) : (a2))
 3 #define max(a1,a2)  ((a1) > (a2) ? (a1) : (a2))
 4 #define min3(a1,a2,a3) ((a1) < (a2) ? (a1) < (a3) ? (a1) : (a3) : \
 5                        (a2) < (a3) ? (a2) : (a3))
 6 #define max3(a1,a2,a3) ((a1) > (a2) ? (a1) > (a3) ? (a1) : (a3) : \
 7                        (a2) > (a3) ? (a2) : (a3))
 8 #define min_abs(a1,a2) (min(abs(a1),abs(a2)))
 9 #define max_abs2(a1,a2)  \
10    (((a1) > 0 && (a2) > 0) ? ((a1) > (a2) ? (a1) : (a2)) : \
11    (((a1) < 0 && (a2) > 0) ? ((-(a1)) > (a2) ? (-(a1)) : (a2)) : \
12    (((a1) > 0 && (a2) < 0) ? ((-(a2)) > (a1) ? (-(a2)) : (a1)) : \
13    ((-(a2)) > (-(a1)) ? (-(a2)) : (-(a1))))))
14
15 #define max_abs2bis(a1,a2)  \
16    (((a1) < 0 ? -(a1) : (a1)) > ((a2) < 0) ? (-a2) : (a2)) ?  \
17    (((a1) < 0 ? -(a1) : (a1)) : ((a2) < 0) ? (-a2) : (a2))
18
19 #define max_abs3(a1,a2,a3) max3(abs(a1),abs(a2),abs(a3))
```

©Christian Bac 1985-2003

PROGRAMME 15.9 SUGGESTION DE CORRIGÉ CHAPITRE 15 EXERCICE 1

```
 1 #include <stdio.h>
 2 #include "c15c01.h"
 3
 4 int
 5 main (int argc, char *argv[], char **envp)
 6 {
 7   int i,j,k;
 8   printf("Rentrez les valeurs de i j et k :");
 9   scanf("%d%d%d",&i,&j,&k);
10
11   printf("Valeurs absolues : %d %d %d \n", abs(i), abs(j), abs(k));
12   printf("Minimum i et j : %d \n", min(i,j));
13   printf("Maximum i et k : %d \n", max(i,k));
14   printf("Minimum i j et k : %d \n", min3(i,j,k));
15   printf("Maximum i j et k : %d \n", max3(i,j,k));
16   printf("Minimum des valeurs absolues : %d \n", min_abs(i,j));
17   printf("Maximum des valeurs absolues : %d \n", max_abs2(i,j));
18   printf("Maximum des trois en valeur absolue : %d \n",max_abs3(i,j,k));
19   return 0;
20 }
```

DONNÉES EN ENTRÉE

```
12 -23 7
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
Rentrez les valeurs de i j et k :Valeurs absolues : 12 23 7
Minimum i et j : -23
Maximum i et k : 12
Minimum i j et k : -23
Maximum i j et k : 12
Minimum des valeurs absolues : 12
Maximum des valeurs absolues : 23
Maximum des trois en valeur absolue : 23
```

©Christian Bac 1985-2003

Programme 15.10 Suggestion de corrigé chapitre 15 exercice 2

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include "c09c03a.h"
 4 int
 5 main (int argc, char *argv[], char **envp){
 6   float a,b,c,r,r1;
 7   double rdis;
 8   float res;
 9   printf("calcul des racines de ax2 + bx + c\n\n");
10   /* saisie des coefficients */
11   printf("saisissez les coefficients a b et c\n");
12   scanf("%f %f %f",&a,&b,&c);
13   if( a == 0 ){
14     printf(" Equation du premier degre \n");
15     printf(" La solution est x = %f \n", -c / b);
16     exit(0);
17   }
18   r = -b/(2 * a);
19 #ifdef DEBUG
20   printf(" -b/2*a : %f \n", r);
21 #endif
22   res = dis(a,b,c);
23 #ifdef DEBUG
24   printf(" discriminant : %f \n", res);
25 #endif
26   switch ( res < 0 ? -1 : (res >0  ? 1 : 0 )) {
27   case 1:
28     rdis = sqrt(res);
29 #ifdef DEBUG
30     printf(" racine du discriminant : %f \n", rdis);
31 #endif
32     r1 = rdis / ( 2 * a);
33 #ifdef DEBUG
34     printf(" r1 : %f \n", r1);
35 #endif
36     rac2(r,r1);
37     break;
38   case -1:
39     rdis = sqrt(-res);
40 #ifdef DEBUG
41     printf(" racine du discriminant : %f \n", rdis);
42 #endif
43     r1 = rdis / ( 2 * a);
44 #ifdef DEBUG
45     printf(" r1 : %f \n", r1);
46 #endif
47     complex(r,r1);
48     break;
49   case 0: racd(r);
50     break;
51   }
52   return 0;
53 }
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

Chapitre 16

Entrées-sorties de la bibliothèque

Comme nous l'avons vu, le langage C est de bas niveau. Pour permettre aux programmeurs de réaliser des programmes sans être obligé de réinventer les fonctionalités les plus courantes, la bibliothèque du langage C fournit des fonctions de plus haut niveau ainsi que des fonctions qui permettent la réalisation des entrée-sorties.

La bibliothèque standard contient :

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

L'ensemble des fonctions qui permettent de faire des entrée-sorties standards 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`, ... ;
- la définition du tableau des fichiers ouverts.

Le langage C donne une vision "*UNIXienne*" des entrée-sorties. 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 standards

Conformément à la vision UNIXienne des entrées-sorties, 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 :

©Christian Bac 1985-2003

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

stdout pour fichier de standard sortie ; 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 standard d'erreur, 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 en utilisant le symbole $|$. Voici quelques exemples non exaustifs d'utilisation des redirections :
prog >fichier : lorsque prog écrit, les octets sont dirigés vers le fichier fichier
prog <fichier : prog lit dans le fichier fichier
prog 2 >fichier : prog écrit ses messages d'erreur dans le fichier fichier
prog1 | prog2 : la sortie standart de prog1 est associée à l'entrée standard de prog2.

À ces fichiers standards 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 : aucun ;

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.

Le programme 16.1 réalise la lecture de son fichier standard d'entrée caractère par caractère et 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 d'exploitation lors des écritures. Sa taille la plus courante est un kilo-octets.

[2] Avant la norme C99, la promotion implicite des arguments de fonctions (voir 3.8.1) faisait que de toute façon la valeur d'un caractère était convertie en une valeur d'entier lors du passage d'arguments à une fonction.

©Christian Bac 1985-2003

PROGRAMME 16.1 LECTURE ET ÉCRITURE CARACTÈRE PAR CARACTÈRE SUR LES FICHIERS STANDARDS

```
1 #include <stdio.h>
2 int
3 main (int argc, char *argv[])
4 {
5   int c;
6   while ((c = getchar ()) != EOF)
7     putchar (c);
8   return 0;
9 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

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 :** écriture d'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.

La version 16.2 reprend le programme précédent de lecture et réécriture entre le fichier standard d'entrée et le fichier standard de sortie mais réalise ces entrées-sorties 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[3].

[3]Cette fonction est l'un des points faibles des programmes écrits en langage C face à une attaque de type débordement de buffer, l'utilisation de `fgets()` que nous verrons plus loin et qui permet d'éviter ces débordements doit lui être préférée.

©Christian Bac 1985-2003

PROGRAMME 16.2 LECTURE LIGNE PAR LIGNE SUR LES FICHIERS STANDARDS

```
1 #include <stdio.h>
2 int
3 main (int argc, char *argv[])
4 {
5   char BigBuf[256];
6   while (gets (BigBuf) != NULL)
7     puts (BigBuf);
8   return 0;
9 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

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 standards d'entrée et de sortie sont les suivantes :

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

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 d'entrée standard 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 de sortie standard 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

À part les trois pseudo-fichiers dont nous avons parlé dans le paragraphe précédent, tout fichier doit être ouvert avant de pouvoir accéder à son contenu en lecture, écriture ou modification.

©Christian Bac 1985-2003

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. Sans précision, 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.

©Christian Bac 1985-2003

"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 une référence vers une structure de données qui sert pour toutes les autres opérations. Le pointeur qui reçoit cette référence (souvent appelé pointeur de fichier par une traduction de `FILE *`) doit être déclaré comme dans le programme 16.3 Si le système d'exploitation ne peut pas

Programme 16.3 Ouverture d'un fichier

```
1 #include <stdio.h>
2 int
3 main (int argc, char *argv[])
4 {
5   FILE *MyFich;
6   MyFich = fopen ("/etc/passwd", "r");
7   /*      ... */
8   return 0;
9 }
```

ouvrir le fichier, il retourne une référence égale au pointeur `NULL`. Si un fichier qui n'existe pas est ouvert en mode écriture ou ajout, il est créé par le système.

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 la référence vers la structure `FILE` 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 : une référence de type `FILE` 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.

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.

©Christian Bac 1985-2003

Programme 16.4 Ouverture et fermeture d'un fichier

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   FILE *MyFich;
 6   MyFich = fopen ("/etc/passwd", "r");
 7   /* ... */
 8   fclose (MyFich);
 9   return 0;
10 }
```

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 : lecture d'un caractère (`unsigned char`) dans le fichier associé ;

argument : une référence de type `FILE` valide correspondant à un fichier ouvert en lecture ;

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 la référence de type `FILE` du 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 (macro-expressions) `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)` ;

©Christian Bac 1985-2003

– `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`).

Le programme 16.5 est 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.

PROGRAMME 16.5 LECTURE CARACTÈRE PAR CARACTÈRE D'UN FICHIER APRÈS OUVERTURE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   FILE *MyFic;
 6   int TheCar;
 7   if (argc != 2)
 8     return 1;
 9   MyFic = fopen (argv[1], "r");
10   if (MyFic == NULL)
11     {
12       printf ("Impossible d ouvrir le fichier %s \n", argv[1]);
13       return 2;
14     }
15   while ((TheCar = fgetc (MyFic)) != EOF)
16     fputc (TheCar, stdout);
17   fclose (MyFic);
18   return 0;
19 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

16.4.2 Accès ligne par ligne

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

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

synopsis : lit une ligne de caractères ou au plus le nombre de caractéres correspondant au deuxième argument moins un, dans le fichier associé au troisième argument. Les caractères de la ligne sont rangés dans la mémoire à partir de l'adresse donnée en premier argument. Si le nombre de caractères lu est inférieur à la taille, 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.

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 la référence de type `FILE` du fichier ouvert.

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

conditions d'arrêt : un ligne est lue (rencontre du retour chariot), la fin de fichier est rencontrée, le nombre de caractére lu est égal à la taille du buffer moins un.

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 la référence de type `FILE` du 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 la 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.

Le programme 16.6 réalise la lecture du fichier correspondant au nom passé en argument sur la ligne de commande et l'écriture de ces lignes sur le fichier standard de sortie. Les opérations sont réalisées ligne par ligne.

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()` :
  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 enregistrement 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

©Christian Bac 1985-2003

PROGRAMME 16.6 LECTURE LIGNE À LIGNE D'UN FICHIER APRÈS OUVERTURE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[])
 4 {
 5   FILE *TheFic;
 6   char BigBuf[256];
 7
 8   if (argc != 2)
 9     return 1;
10   TheFic = fopen (argv[1], "r");
11   if (TheFic == NULL)
12     {
13       printf ("Impossible d ouvrir le fichier %s \n", argv[1]);
14       return 2;
15     }
16   while (fgets (BigBuf, sizeof BigBuf, TheFic) != NULL)
17     fputs (BigBuf, stdout);
18   fclose (TheFic);
19   return 0;
20 }
```

DONNÉES EN ENTRÉE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
The quick brown fox jumped over the lazy dog, 1234567890.
```

doit être ouvert en mode binaire (voir le programme 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 une référence vers une structure de type `FILE` correspondant à un 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.

©Christian Bac 1985-2003

Le programme 16.7 est un exemple, d'utilisation de la fonction `fread()`. Cet exemple réalise la 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 int
 9 main (int argc, char *argv[]) {
10   FILE *TheFic;
11   int i;
12   size_t fait;
13   TheFic = fopen ("FicParcAuto", "rb+");
14   if (TheFic == NULL)
15     {
16       printf ("Impossible d ouvrir le fichier FicParcAuto\n");
17       return 1;
18     }
19   for (i = 0; i < 20; i++)
20     {
21       fait = fread (&ParcAuto[i], sizeof (struct automobile), 1, TheFic);
22       if (fait != 1)
23   {
24     printf ("Erreur lecture fichier parcauto \n");
25     return 2;
26   }
27     }
28   fclose (TheFic);
29   return 0;
30 }
```

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);
if(fait != 20){
  printf ("Erreur lecture fichier parcauto \n");
  return 2;
}
```

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

©Christian Bac 1985-2003

nous allons étudier 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 [4] pour chaque variable simple ou pour un tableau de caractères [5].

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 fait que lorsque des caractères blancs (espace, tabulation, retour chariot) sont lus ils sont consommés et mis à la poubelle ;
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 pour `scanf()` sont décrits dans la table 16.1.

La **spécification** entre les crochets définit un alphabet[6] de caractères. La donnée lue doit être conforme à cette spécification. La lecture avec un format de spécification retourne une chaîne de caractères.

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

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

[6] 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.

©Christian Bac 1985-2003

code	conversion réalisée
%	lit un %
d	entier signé exprimé en base décimale
i	entier signé exprimé en base décimale
o	entier non signé exprimé en base octale
u	entier non signé exprimé en base décimale
x	entier non signé exprimé en hexadécimal
e f g	nombre avec partie décimale en notation point décimal ou 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 spécification de 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 majuscule (A-F).

©Christian Bac 1985-2003

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, plus petit en taille des formats 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
0	remplissage avec des 0 devant plutot que des blancs
'	pour les conversions décimales groupement des chiffres par 3
#	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

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`.

Selon la norme [ISO89], 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 standards

Nous avons déjà exploré dans la section 16.1.3 les deux fonctions d'entrée-sorties standards 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, ...);
```

©Christian Bac 1985-2003

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. référence vers la structure décrivant le fichier ouvert 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. référence vers la structure décrivant le 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 conversions réussies.

conditions d'erreur : la valeur `EOF` est retournée en cas d'erreur (fin de fichier atteinte avant la première conversion).

©Christian Bac 1985-2003

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) ou un `x` si le système utilise un fichier séparé (`shadow` pour stocker les mots de passe,
- 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).

16.6 Déplacement dans le fichier

Jusqu'à maintenant nous avons vu des fonctions qui modifient de manière automatique le pointeur courant dans le fichier correspondant (adresse de l'octet dans le fichier à partir duquel se fait 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. référence vers la structure décrivant le fichier ouvert ;
2. déplacement à l'intérieur du fichier en nombre d'octets ;
3. point de départ du déplacement. Cet argument peut prendre les valeurs suivantes qui selon la norme doivent être définies dans le fichier $< stdio.h >$ mais sont souvent dans le fichier $< unistd.h >$ sur les machines de type "UNIX system V" :

   `SEEK_SET` le déplacement est relatif au début du fichier ;

   `SEEK_CUR` le déplacement est relatif à la position courante ;

   `SEEK_END` le déplacement est 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 : référence vers la structure décrivant le 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. référence vers la structure décrivant le 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. référence vers la structure décrivant le 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 la référence vers la structure décrivant le fichier ouvert `fp` est valide cette fonction est équivalente 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[7] (sauf `stderr`).

[7] Anglicisme que l'on peut traduire par tamponnées. Prenons le cas des écritures. Elles sont d'abord réalisées dans un espace mémoire local au programme que l'on appelle tampon (buffer en anglais). Les caractères sont ensuite transférés au système d'exploitatin en bloc. Ceci permet de minimiser les appels au système d'exploitation (car le tampon de caractères est dans l'espace mémoire du programme) et en général d'améliorer les performances. Cette gestion de tampons intermédiaires 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.

©Christian Bac 1985-2003

Dans le cas général, l'allocation du tampon se fait de manière automatique lors de la première entrée-sortie, la taille de ce tampon est de l'ordre du kilo-octet. Il est cependant possible d'associer un buffer avec un fichier ouvert par les fonctions décrites ci-après, pour par exemple optimiser la taille de ce tampon par rapport aux écritures réalisées par le programme. 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. référence vers la structure décrivant le 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.

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

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

arguments :

1. référence vers la structure décrivant le 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 : référence vers la structure décrivant le fichier ouvert en mode écriture ou en mode mise-à-jour. Cette référence peut être égale 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.

©Christian Bac 1985-2003

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 :** la référence vers la structure décrivant le 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 positionné sur le fichier ouvert correspondant à la référence en argument.
- **argument :** le référence vers la structure décrivant le 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 à la référence donnée en argument.
- **argument :** la référence vers la structure décrivant le 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.

`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.

Le programme 16.11 illustre cette gestion d'erreur, par l'ajout des tests d'erreur dans l'exemple de parcours du fichier "FicParcAuto" avec modification de l'âge dans les différents champs.

©Christian Bac 1985-2003

PROGRAMME 16.8 LECTURE AVEC FORMAT DANS UN FICHIER TEXTE

```
 1 #include <stdio.h>
 2 int
 3 main (int argc, char *argv[]) {
 4   FILE *pwf;
 5   int i, res;
 6   char nom[10], passwd[16], gcos[128], rep[255], shell[255];
 7   int uid, gid;
 8 
 9   pwf = fopen ("passwd", "r");
10   if (pwf == NULL){
11     printf ("Impossible d ouvrir le fichier %s \n", "/etc/passwd");
12     return 1;
13   }
14   while (!feof (pwf)){
15     res = fscanf (pwf, " %[^:]:", nom);
16     if (res != 1) break;
17     res = fscanf (pwf, "%[^:]:", passwd);
18     if (res != 1) break;
19     res = fscanf (pwf, "%d:", &uid);
20     if (res != 1) break;
21     res = fscanf (pwf, "%d:", &gid);
22     if (res != 1) break;
23     for (i = 0; i < 128; i++){
24       res = fgetc (pwf);
25       if (res == ':'){
26               gcos[i] = '\0';
27               break;
28       } else  gcos[i] = res;
29     }
30     res = fscanf (pwf, "%[^:]:", rep);
31     if (res != 1) break;
32     res = fgetc (pwf);
33     if (res != '\n'){
34       ungetc (res, pwf);
35       res = fscanf (pwf, "%s", shell);
36       if (res != 1) break;
37     } else  shell[0] = '\0';
38     printf ("%s %s %d %d %s %s %s\n", nom, passwd, uid, gid, gcos,
39       rep, shell);
40   }
41   fclose (pwf);
42   return 0;
43 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD DE SORTIE

```
root x 0 0 root /root /bin/bash
daemon x 1 1 daemon /usr/sbin /bin/sh
bin x 2 2 bin /bin /bin/sh
sys x 3 3 sys /dev /bin/sh
gdm x 102 101 Gnome Display Manager /var/lib/gdm /bin/false
```

Programme 16.9 Modifications par déplacement 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 } uneauto;
 7
 8 int
 9 main (int argc, char *argv[]) {
10   FILE *fparc;
11   int i;
12   size_t fait;
13   fparc = fopen ("FicParcAuto", "r+b");
14   if (fparc == NULL){
15     printf ("Impossible d ouvrir le fichier FicParcAuto \n");
16     return 1;
17   }
18   for (i = 0; i < 20; i++) {
19     fait = fread (&uneauto, sizeof uneauto, 1, fparc);
20     if (fait != 1) {
21           printf ("Erreur lecture fichier FicParcAuto \n");
22           return 2;
23     }
24     uneauto.age++;
25     fait = fseek (fparc, (long) -sizeof uneauto, SEEK_CUR);
26     if (fait != 0){
27       printf ("Erreur deplacement fichier FicParcAuto \n");
28       return 3;
29     }
30     fait = fwrite (&uneauto, sizeof uneauto, 1, fparc);
31     if (fait != 1) {
32       printf ("Erreur ecriture fichier FicParcAuto fait = %d \n", fait);
33       return 4;
34     }
35   }
36   fclose (fparc);
37   return 0;
38 }
```

Programme 16.10 Déplacements dans un fichier avec fgetpos()

```
 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 } parc[20];
 7
 8 int
 9 main (int argc, char *argv[]) {
10   FILE *fparc;
11   int i;
12   size_t fait;
13   fpos_t curpos;
14   fparc = fopen ("FicParcAuto", "r+b");
15   if (fparc == NULL){
16     printf ("Impossible d ouvrir le fichier FicParcAuto \n");
17     return 1;
18   }
19   for (i = 0; i < 20; i++) {
20     fait = fgetpos (fparc, &curpos);
21     if (fait != 0) {
22       printf ("Erreur acquisition Position \n");
23       return 2;
24     }
25     fait = fread (&parc[i], sizeof (struct automobile), 1, fparc);
26     if (fait != 1) {
27       printf ("Erreur lecture fichier FicParcAuto \n");
28       return 3;
29     }
30     parc[i].age++;
31     fait = fsetpos (fparc, &curpos);
32     if (fait != 0) {
33       printf ("Erreur restitution Position \n");
34       return 4;
35     }
36     fait = fwrite (&parc[i], sizeof (struct automobile), 1, fparc);
37     if (fait != 1) {
38       printf ("Erreur ecriture fichier parcauto fait = %d \n", fait);
39       return 5;
40     }
41   }
42   fclose (fparc);
43   return 0;
44 }
```

©Christian Bac 1985-2003

PROGRAMME 16.11 GESTION DES CAS D'ERREURS PENDANT LA MANIPULATION D'UN FICHIER

```
 1 #include <stdio.h>
 2 #include <stddef.h>
 3 #include <unistd.h>
 4 struct automobile {
 5   int age;
 6   char couleur[20], numero[10], type[10], marque[10];
 7 } parc;
 8
 9 int
10 main (int argc, char *argv[]) {
11   FILE *fparc;
12   size_t fait;
13   fparc = fopen ("FicParcAuto", "r+b");
14   if (fparc == NULL) {
15     perror ("Impossible d ouvrir FicParcAuto");
16     return 1;
17   }
18   while (1){
19     fait = fread (&parc, sizeof parc, 1, fparc);
20     if (fait != 1) {
21       if (feof (fparc))
22             fprintf (stderr, "Fin de fichier FicParcAuto \n");
23       else
24   fprintf (stderr, "Erreur lecture FicParcAuto\n");
25       break;
26     }
27     parc.age++;
28     fait = fseek (fparc, (long) -sizeof (parc), SEEK_CUR);
29     if (fait != 0) {
30       perror ("Erreur deplacement FicParcAuto");
31       break;
32     }
33     fait = fwrite (&parc, sizeof parc, 1, fparc);
34     if (fait != 1) {
35       fprintf (stderr, "Erreur ecriture FicParcAuto fait = %d \n", fait);
36       break;
37     }
38     fflush (fparc);
39   }
40   clearerr (fparc);
41   fclose (fparc);
42   return 0;
43 }
```

DONNÉES ÉCRITES SUR LE FICHIER STANDARD D'ERREUR

```
Fin de fichier FicParcAuto
```

©Christian Bac 1985-2003

Chapitre 17

Autres fonctions de la bibliothèque

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 les tables 17.1 et 17.2.

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 un caractère de ponctuation.
– `isalnum(c)` vrai si c est alphabétique ou numérique.
– `isprint(c)` vrai si c est affichable 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).

Déclaration	Travail	Retour

`char *s1, *s2;`
`int n;`
`char *strcat (s1, s2)`
`char *strncat (s1, s2, n)`
	
	ajoute la chaîne s2 derrière 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 compare 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 copie sur au plus n caractères	
`char *s;` `int strlen (s)`		 taille de s

TAB. 17.1 – Fonctions de manipulation simples de chaînes de caractères

©Christian Bac 1985-2003

Déclaration	Travail	Retour

`char *s1, *s2;`
`int n;`
`int c;`
`char *strchr (s, c)`
`char *strrchr (s, c)`
`char *index(s, c)`
`char *rindex(s, c)`
	
	cherche caractère c dans s
	idem
	idem que strchr
	idem que strrchr
		
		pointeur sur première occurrence
		pointeur sur dernière occurrence

`char *strpbrk (s1, s2)`	cherche 1 caractère de s2 dans s1	pointeur sur première occurrence

`int strspn (s1, s2)`
 	
`int strcspn (s1, s2)`
	1er motif dans s1 ne contenant que des caractères de s2
	1er motif dans s1 ne contenant aucun caractère de s2
		longueur
		
		longueur

TAB. 17.2 – Fonctions de manipulation de motifs dans des chaînes de caractères

– `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.

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.

©Christian Bac 1985-2003

Programme 17.1 Utilisation de l'appel système `exit()`

```
 1 int
 2 ouvre (const char *nom_fichier)
 3 {
 4   int o;
 5   o = open (nom_fichier, 0);
 6   if (o == -1)
 7     {
 8       printf ("impossible d'ouvrir %s\n", nom_fichier);
 9       exit (1);
10     }
11   return o;
12 }
```

©Christian Bac 1985-2003

©Christian Bac 1985-2003

GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

17.4 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of

Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

17.5 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

17.6 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this

Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

17.7 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another ; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

17.8 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History” ; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

17.9 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

17.10 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

17.11 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

17.12 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

17.13 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http ://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM : How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page :

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation ; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST” ; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

©Christian Bac 1985-2003

Table des programmes exemples

3.1 Exemple de fichier `limits.h` 16
3.2 Exemple de fichier `float.h` 17
3.3 Exemple de fichier `stddef.h` 17
3.4 Suggestion de corrigé chapitre 3 exercice 1 24
3.5 Suggestion de corrigé chapitre 3 exercice 1 second fichier 24
3.6 Suggestion de corrigé chapitre 3 exercice 2 24
3.7 Suggestion de corrigé chapitre 3 exercice 3 25
3.8 Suggestion de corrigé chapitre 3 exercice 4 26
4.1 Lecture et écriture de chaîne par `scanf()` et `printf()` 28
4.2 Lectures multiples avec `scanf()` 31
4.3 Suggestion de corrigé chapitre 4 exercice 1 32
4.4 Suggestion de corrigé chapitre 4 exercice 2 33
4.5 Suggestion de corrigé chapitre 4 exercice 3 34
4.6 Suggestion de corrigé chapitre 4 exercice 4 35
5.1 Définitions de variables et d'un pointeur 38
5.2 Utilisation des opérateurs de masquage 42
5.3 Utilisation des opérateurs de décalage 43
5.4 Suggestion de corrigé chapitre 5 exercice 1 50
5.5 Suggestion de corrigé chapitre 5 exercice 2 51
5.6 Suggestion de corrigé chapitre 5 exercice 3 52
5.7 Suggestion de corrigé chapitre 5 exercice 4 53
5.8 Suggestion de corrigé chapitre 5 exercice 5 54
5.9 Suggestion de corrigé chapitre 5 exercice 6 55
6.1 Exemple de tests imbriqués 58
6.2 Exemple de table de sauts 67
6.3 Exemple de table de sauts 68
6.4 Exemple de table de sauts 69
6.5 Lecture d'une ligne avec `while` 70
6.6 Recopie d'une chaîne avec une boucle `while` 70
6.7 Lecture d'une ligne avec for 71
6.8 Décomposition des puissances de dix d'un nombre avec un `do while` 71
6.9 Utilisation du continue dans une boucle `for` 72
6.10 Utilisation des ruptures de séquence dans une boucle `for` 72
6.11 Lecture d'une ligne avec `for` et `break` 73
6.12 Utilisation de l'infâme `goto` 74
6.13 Utilisation de plusieurs `return` 75
6.14 Suggestion de corrigé chapitre 6 exercice 1 76
6.15 Suggestion de corrigé chapitre 6 exercice 2 77
6.16 Suggestion de corrigé chapitre 6 exercice 3 78
6.17 Suggestion de corrigé chapitre 6 exercice 4 79
6.18 Suggestion de corrigé chapitre 6 exercice 5 80
6.19 Suggestion de corrigé chapitre 6 exercice 6 81

©Christian Bac 1985-2003

7.1 Ensemble de tests non structuré . . . 87
7.2 Ensemble de tests structuré non mis en page . . . 88
7.3 Ensemble de tests structuré et mis en page . . . 89
7.4 Ensemble de tests structuré et mis en page avec commentaires . . . 90
7.5 Ensemble de tests structuré, mis en page et étiqueté . . . 91
7.6 Ensemble de tests par table de branchement . . . 92
7.7 Boucle réalisée par des tests et sauts . . . 93
7.8 Boucle réalisée par un for() . . . 93
7.9 Boucle réalisée par un for() et mise en page . . . 93
8.1 Affichage des arguments de `main()` . . . 105
8.2 Arguments de `main()` caractères un-à-un . . . 107
8.3 Suggestion de corrigé chapitre 8 exercice 1 . . . 108
8.4 Suggestion de corrigé chapitre 8 exercice 2 . . . 109
8.5 Suggestion de corrigé chapitre 8 exercice 3 . . . 110
9.1 Exemples de prototypes de fonctions . . . 115
9.2 Déclarations candidates multiple d'une variable . . . 117
9.3 Déclaration explicite et déclaration candidate d'une variable . . . 117
9.4 Suggestion de corrigé chapitre 9 exercice 1 fonctions . . . 122
9.5 Suggestion de corrigé chapitre 9 exercice 1 `main()` . . . 123
9.6 Suggestion de corrigé chapitre 9 exercice 2 fonctions . . . 124
9.7 Suggestion de corrigé chapitre 9 exercice 2 `main()` . . . 125
9.8 Suggestion de corrigé chapitre 9 exercice 3 fichier d'inclusion . . . 126
9.9 Suggestion de corrigé chapitre 9 exercice 3 `main()` . . . 127
10.1 Définition de tableaux et initialisations . . . 130
10.2 Accès à un tableau à deux dimensions avec un pointeur . . . 137
10.3 Algorithme d'un tri simple . . . 138
10.4 Suggestion de corrigé chapitre 10 exercice 1 fichier d'inclusion . . . 138
10.5 Suggestion de corrigé chapitre 10 exercice 1 fonctions . . . 139
10.6 Suggestion de corrigé chapitre 10 exercice 1 `main` . . . 140
10.7 Suggestion de corrigé chapitre 10 exercice 1 fonction de tri . . . 141
11.1 Définition de structures . . . 144
11.2 Définition de structures . . . 145
11.3 Structures et listes chaînées . . . 145
11.4 Structures et champs de bits . . . 146
11.5 Suggestion de corrigé chapitre 11 exercice 1 . . . 147
11.6 Suggestion de corrigé chapitre 11 exercice 2 . . . 148
12.1 Utilisation d'une union . . . 149
12.2 Utilisation d'une union . . . 150
12.3 Suggestion de corrigé chapitre 12 exercice 1 . . . 151
12.4 Suggestion de corrigé chapitre 12 exercice 2 . . . 152
13.1 Utilisation d'énumérations . . . 154
14.1 Suggestion de corrigé chapitre 14 exercice 1 . . . 160
14.2 Suggestion de corrigé chapitre 14 exercice 2 . . . 162
14.3 Suggestion de corrigé chapitre 14 exercice 3 fichier d'inclusion . . . 163
14.4 Suggestion de corrigé chapitre 14 exercice 3 fonctions . . . 163
14.5 Suggestion de corrigé chapitre 14 exercice 3 . . . 164
15.1 Traitement des commentaires . . . 166
15.2 Erreur due au traitement des commentaires . . . 167
15.3 Interprétation des variables par le préprocesseur . . . 169
15.4 Evaluation de macros par CPP . . . 172
15.5 Exemple d'utilisation du #if . . . 173
15.6 Exemples de sélection de code par `#ifdef` . . . 174
15.7 Sélection de code par `#ifdef` imbriqués . . . 174
15.8 Suggestion de corrigé chapitre 15 exercice 1 définition des macros . . . 175

©Christian Bac 1985-2003

15.9 Suggestion de corrigé chapitre 15 exercice 1 176
15.10 Suggestion de corrigé chapitre 15 exercice 2 177
16.1 Lecture et écriture caractère par caractère sur les fichiers standards 181
16.2 Lecture ligne par ligne sur les fichiers standards 182
16.3 Ouverture d'un fichier 184
16.4 Ouverture et fermeture d'un fichier 185
16.5 Lecture caractère par caractère d'un fichier après ouverture 186
16.6 Lecture ligne à ligne d'un fichier après ouverture 188
16.7 Lecture d'enregistrements dans un fichier 189
16.8 Lecture avec format dans un fichier texte 198
16.9 Modifications par déplacement dans un fichier 199
16.10 Déplacements dans un fichier avec fgetpos() 200
16.11 Gestion des cas d'erreurs pendant la manipulation d'un fichier 201
17.1 Utilisation de l'appel système `exit()` 205

©Christian Bac 1985-2003

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Table des figures

1.1 Structure d'un programme C . . . 3
1.2 Fichier source . . . 4
1.3 Structure d'une fonction C . . . 4
1.4 Schéma de fonctionnement de `cc` . . . 6

3.1 Chaîne de caractères constante . . . 15

4.1 Programme qui écrit Bonjour . . . 27

5.1 Exemple de relation entre pointeur et variable . . . 38
5.2 Mise en relation d'un pointeur et d'une variable . . . 39

6.1 Organigramme du `while` . . . 60
6.2 Organigramme du `for` . . . 61
6.3 Organigramme du `do while` . . . 62
6.4 `break` et `continue` dans un `for` . . . 64
6.5 `break` et `continue` dans un `while` . . . 64
6.6 `break` et `continue` dans un `do while` . . . 65

8.1 Structure d'une fonction . . . 95
8.2 Pile et passage de constantes . . . 98
8.3 Pile et passage de variables . . . 100
8.4 Pile et passage de variables avec référence . . . 101
8.5 Arguments de `main()` . . . 104

9.1 Du source à l'exécutable . . . 112
9.2 Survol d'un fichier source . . . 112
9.3 Exemple de visibilité . . . 113
9.4 Visibilité des variables entre modules . . . 115
9.5 Visibilité des fonctions entre modules . . . 116
9.6 Visibilité des fonctions dans un module . . . 116
9.7 Utilisation de fichier d'inclusion . . . 118

©Christian Bac 1985-2003

9.8 Réduction de visibilité 119
9.9 Variables locales statiques 120

10.1 Tableau de dix entiers 129
10.2 Tableau de dix entiers initialisé 130
10.3 Adresses dans un tableau de dix entiers 130
10.4 Tableau à deux dimensions 131
10.5 Pointeur et tableau 134
10.6 Tableau de pointeurs sur des variables dans un tableau 134
10.7 Accès à un tableau à deux dimensions avec un pointeur 135

©Christian Bac 1985-2003

Liste des tableaux

3.1 Longueur des types de base sur quelques machines 13
3.2 Variables et classes mémoire 18
3.3 Exemples d'initialisations 21
3.4 Exemples de conversion implicite 23

4.1 Conversions usuelles de `printf` et `scanf` 29
4.2 Exemples de `printf` et `scanf` 30

5.1 Liste des opérateurs unaires 37
5.2 Liste des opérateurs binaires 41
5.3 Liste des opérateurs binaires d'affectation 44
5.4 Exemples d'opérateurs binaires d'affectation 44
5.5 Opérateur ternaire 45
5.6 Précédence des opérateurs 46

6.1 Syntaxe du `while` 60
6.2 Comparaison du `for` et du `while` 61

7.1 C et Structures Fondamentales 85

8.1 Exemples de définition de fonctions 96
8.2 Exemples d'appels de fonctions 97
8.3 Pointeurs et appels de fonctions 99
8.4 Conversions de type un-aire 102

9.1 Règle fondamentale de visibilité 113

10.1 Addition d'un entier à un pointeur 132
10.2 Soustraction de deux pointeurs 133

11.1 Accès aux champs d'une structure 144

14.1 Exemples d'objets complexes 159

©Christian Bac 1985-2003

15.1 Utilisation d’une constante de compilation 168

16.1 Code de conversion pour `scanf()` 191
16.2 Codes de conversion pour `printf()` 192
16.3 Modificateurs de format pour printf 192

17.1 Fonctions de manipulation simples de chaînes de caractères 203
17.2 Fonctions de manipulation de motifs dans des chaînes de caractères 204

©Christian Bac 1985-2003

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.

[ISO89] ISO/IEC, editor. *Programming Language C*. ISO/IEC, 1989.

[ISO99] ISO/IEC, editor. *Programming Language C*. ISO/IEC, 1999.

[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.

©Christian Bac 1985-2003

Index

#define, 165, 168–170
#else, 165, 170
#endif, 165, 170
#if, 165, 170, 171
#ifdef, 165, 170, 171
#ifndef, 165, 170, 171
#include, 165, 166, 179
#undef, 165, 169

accolade, 5, 10, 20, 28, 47, 85, 96, 129
adresse, 11, 15, 19, 20, 23, 24, 29, 38, 39, 47, 48, 97, 99, 103, 105, 106, 113, 119, 120, 129–133, 135, 144, 145, 150, 151, 157, 158, 181, 182, 186–188, 190, 193, 194, 196
affectation, 3, 15, 16, 19, 20, 29, 37, 43–46, 48, 57, 62, 99, 114, 132, 144
alias, 5, 165
appel, 1–5, 11, 17, 19, 22–24, 28, 38, 40, 42, 43, 46, 63–65, 95–97, 99, 102, 103, 105, 106, 113–116, 119–121, 157, 158, 161, 166, 168–170, 180–182, 184, 185, 188, 189, 191, 192, 194–197, 204
argument, 11, 23, 27, 28, 65, 97, 99, 102, 103, 105, 111, 114–116, 121, 133, 135, 144, 168, 180–188, 191, 193–197
associativité, 46, 158

bibliothèque, 3, 7, 27, 28, 63, 96, 179, 183, 197, 203
binaire, 2, 4, 40, 42, 44, 46, 103, 114, 145, 170, 183, 184, 187, 188, 194
bit, 2, 11, 12, 15, 20, 22, 40–44, 46, 48, 102, 145, 146, 151, 180
bloc, 4, 5, 17, 47, 62–64, 95, 96, 114, 119, 195
booléen, 57, 168
boucle, 5, 60–64, 66, 84–86, 121, 161

caractère, 1–3, 5, 9–11, 13–15, 19, 20, 22–25, 27–29, 31, 44, 59, 60, 62–64, 66, 86, 102, 103, 132, 150, 151, 165, 168, 179–195, 197, 203, 204
case, 46, 59, 99, 133
chaîne, 2, 3, 5, 10, 14, 15, 20, 23, 28, 29, 31, 60, 64, 66, 86, 103, 116, 165, 168, 179, 181–183, 186, 187, 189–191, 193, 197, 203, 204
champ, 9, 22, 102, 143–146, 149–151, 190, 191, 194, 195, 197
champs
  de bit, 22, 102, 145, 146
char, 1, 103, 132, 182, 183, 187, 192, 193, 196, 197
chargeable, 7
classe, 10, 11, 16, 17, 106, 113, 114
commentaire, 5, 85, 111, 165, 182
compilateur, 1–3, 5, 7–9, 11–13, 15, 18–20, 22, 27, 28, 40, 41, 45, 46, 60, 64, 84, 105, 111, 113, 114, 116, 117, 131–133, 135, 144, 149, 150, 155, 157, 165–168, 170, 171
compilation, 2, 4, 5, 7, 9, 17, 25, 28, 84, 95, 111, 113, 114, 116, 118–120, 165, 166, 168–171
constante, 9–11, 13–16, 22, 37, 40, 59, 99, 102, 103, 129, 130, 133, 153–155, 168–171, 179
continue, 10, 59, 62, 86, 106
contrôle, 1, 5, 27, 43, 47, 57, 60–62, 114, 116, 119, 193, 196, 203
conversion, 20, 22, 29, 99, 102, 132, 150, 189–193
corps, 4, 23, 65, 95, 96, 111, 116, 168

déclaration, 4, 18, 23, 29, 95, 96, 111, 114–119, 129, 132, 135, 157, 158
default, 10
définition, 4, 5, 11, 12, 15–20, 23, 28, 38, 40, 42, 47, 83, 95, 96, 106, 111, 114, 116–118, 129, 131–133, 143–146, 149, 153, 157, 158, 160, 165, 168–171, 179, 203
double, 10, 43, 64, 121
durée
  de vie, 17, 119, 120

écrire, 1, 3, 12, 15, 23, 24, 27, 29, 31, 32, 37, 42–45, 47–49, 62, 66, 83–85, 96, 102, 115, 131, 155, 157, 161, 180, 182, 185, 187, 188, 190, 191, 195, 204

©Christian Bac 1985-2003

écriture, 2, 28, 29, 31, 42, 59, 66, 135, 179–187, 190, 191, 193, 195, 196
édition
  de lien, 3, 7, 23, 103, 114
else, 10, 57
enchaîneur
  de passe, 169, 170
entier, 1, 11–13, 15, 19, 20, 22–25, 29, 31, 38–44, 47–49, 62, 63, 65, 66, 85, 97, 99, 102, 103, 106, 114, 129–132, 135, 146, 149–151, 153–155, 157, 158, 160, 171, 180, 185, 186, 189–191, 194, 195, 197
entry, 10
énumération, 22, 102, 153–155
étiquette, 9, 64, 85
exécutable, 2, 4, 5, 7, 42, 103, 114, 170, 194
exécution, 4, 11, 15–19, 39, 40, 59, 61–63, 65, 97, 99, 102, 105, 119, 120, 132, 170, 183, 204
expression, 5, 10, 11, 22, 37–47, 49, 57, 59–62, 65, 96, 102, 106, 131, 133, 154, 155, 157, 165, 168, 169, 171, 179, 180, 203, 204
extern, 10
externe, 4, 23, 111, 190, 191

faux, 40, 57, 132
fichier, 2–5, 7, 11, 15, 17, 18, 23, 24, 27, 28, 31, 40–43, 60, 62, 64, 96, 103, 111, 113–119, 121, 135, 154, 165–171, 179–189, 193–197, 203, 204
float, 10, 114
flottant, 23, 25, 31, 160, 171, 191
fonction, 3, 4, 7, 9, 11, 16, 17, 22, 23, 27–29, 31, 46, 57, 60, 63–65, 86, 95–97, 99, 102, 103, 105, 106, 111, 113–121, 132, 133, 135, 144, 158, 161, 165, 168, 171, 179–189, 192–197, 203, 204
for, 2, 10, 61
format, 7, 9, 28, 29, 31, 103, 180, 182, 184, 189–193
fortran, 10

global, 17, 20, 23, 24, 114, 120
goto, 10

if, 10, 18, 57, 158
initialisation, 10, 11, 16, 20, 61, 62, 117, 120, 129, 132
instruction, 2, 5, 7, 10, 20, 23, 28, 38, 39, 42, 43, 47, 49, 57–63, 65, 66, 83, 84, 95, 96, 103, 106, 111, 168, 170, 195, 204
int, 10, 41, 63, 115–117, 132, 186, 196
interface, 4, 27, 28, 95, 96, 102, 113, 115, 118, 188

langage, 1–3, 5, 7–12, 16, 23, 27, 29, 37, 40, 41, 43, 44, 46, 47, 57, 59, 83–85, 95, 97, 102, 103, 111, 114–119, 129, 130, 143–145, 153, 154, 158, 171, 179, 181, 184, 204
lecture, 5, 15, 28, 29, 32, 42, 45, 62, 63, 66, 83, 85, 121, 135, 158, 165, 179–187, 189, 190, 193, 195
lire, 27–29, 32, 44, 45, 65, 66, 121, 135, 150, 171, 180, 181, 187, 190, 204
local, 17, 23, 114, 132, 195
long, 10, 14, 22, 25, 40, 114, 190, 191

main, 4, 23, 28, 31, 65, 102, 103, 106, 121, 133, 135, 161
mémoire, 1, 11, 12, 15–17, 19, 20, 28, 38, 41, 95, 97, 99, 102, 111, 114, 119, 129, 131, 132, 143, 149, 150, 157, 179, 181, 186–189, 192, 193, 195, 196, 204

norme, 1–3, 7, 9, 11, 12, 15, 20, 22, 40, 102, 114, 146, 153, 170, 180, 192, 194

objet, 2, 7, 18, 19, 28, 29, 99, 111, 114, 117, 130–132, 143, 146, 158, 160, 182, 183, 187, 204
octet, 1, 2, 11, 12, 14, 15, 38, 40, 60, 130, 145, 179, 180, 186, 188, 194, 196
opérateur, 12, 15, 19, 29, 32, 37–48, 57, 62, 66, 131, 158
optimiseur, 5, 7

paramètre, 4, 95, 97, 99, 102, 103, 106, 113, 196
pile, 17, 97, 99, 102, 103, 105, 106, 132
pointeur, 1, 11, 18–20, 23–25, 38, 39, 47, 48, 97, 99, 103, 105, 106, 119, 121, 129, 131–133, 135, 136, 145–147, 158, 160, 161, 181, 183–185, 194–196, 204
précédence, 45, 46, 158
préprocesseur, 168–171, 180
printf, 28, 29, 66, 103, 115, 182, 189, 191–193
procédure, 95
programmation, 2, 3, 5, 63, 66, 83, 84, 116, 179, 182
prototype, 4, 114–117, 157, 179, 184

qualificatif, 11, 12, 15, 17, 18

référence, 2–4, 7, 17, 23, 40, 114, 116, 168, 184, 185, 187, 188, 193–197
register, 10
relogeable, 7
retour, 11, 14, 31, 45, 63, 65, 66, 95–97, 99, 105, 106, 144, 171, 180–187, 190, 193–197
return, 10, 65, 96, 102, 204

©Christian Bac 1985-2003

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

rupture, 5, 10, 62, 63, 66, 204

scanf, 28, 29, 99, 115, 182, 189, 190, 192, 193
short, 10, 41, 190
sizeof, 10, 12, 40, 132, 204
source, 2–5, 17, 18, 43, 46, 85, 95, 96, 111, 113, 116–118, 135, 165, 168, 170
static, 119
statique, 17, 23, 102, 119, 120
struct, 10, 18
structure, 1–3, 9, 18–20, 47, 83–85, 103, 111, 117, 119, 133, 143–147, 149, 150, 158, 160, 161, 179, 184, 187–189, 193–197
structuré, 83, 84, 121, 144, 187
système, 1, 2, 5, 7, 15, 27, 28, 41, 42, 60, 84, 85, 102, 103, 121, 154, 166, 170, 171, 179, 180, 183, 184, 186, 194, 195, 197, 204

table
  de branchement, 58, 85
tableau, 12, 15, 18–20, 22–25, 28, 29, 37, 40, 44–48, 60–64, 66, 84, 86, 96, 97, 99, 102, 103, 120, 129–133, 135, 136, 144, 145, 147, 157, 158, 160, 161, 165, 168, 179, 181, 182, 187, 189, 190
ternaire, 45, 46, 66
test, 5, 42–47, 57, 59–62, 64, 65, 84, 85, 131, 154, 168, 171, 197, 203
type, 1–5, 7, 9–20, 22, 23, 27–29, 31, 38–45, 47, 57, 60, 62, 63, 65, 66, 84, 85, 95–97, 99, 102, 103, 105, 106, 111, 113–118, 120, 130–133, 135, 143–146, 149, 150, 153–155, 157, 158, 160, 166, 168, 170, 171, 179–181, 183–188, 190–192, 194–197, 203
typedef, 10
types
  prédéfinis, 11, 15

unaire, 12
union, 149, 150, 160
UNIX, 1, 2, 5, 7, 27, 28, 41, 42, 60, 85, 103, 132, 154, 166, 180, 186, 194
unsigned, 10

valeur, 2, 11–16, 19, 20, 22, 23, 25, 28, 38–45, 47–49, 57, 59–63, 65, 85, 96, 97, 99, 102, 103, 105, 106, 113, 120, 129–132, 150, 153–155, 168–171, 180–182, 185–187, 190, 191, 193–197, 204
variable, 3–5, 7, 9–12, 15–20, 22–24, 27–29, 31, 32, 37–45, 47, 49, 57, 59, 62, 65, 95–97, 99, 102, 103, 105, 106, 111, 113–115, 117–120, 131–133, 135, 143–146, 149, 150, 153–155, 157, 158, 165, 167–171, 182, 190, 193–197
visibilité, 3, 11, 17, 113, 114, 120
void, 10
vrai, 40, 57, 197, 203

while, 62

©Christian Bac 1985-2003