Université de la Méditerranée
Faculté des Sciences de Luminy

Le langage C

Licences Maths & Informatique – Master Mathématiques – Master C.C.I.

Henri Garreta
Département d'Informatique - LIF

Table des matières

1 Éléments de base **5**
1.1 Structure générale d'un programme . . . 5
1.2 Considérations lexicales . . . 5
1.2.1 Présentation du texte du programme . . . 5
1.2.2 Mots-clés . . . 6
1.2.3 Identificateurs . . . 6
1.2.4 Opérateurs . . . 6
1.3 Constantes littérales . . . 7
1.3.1 Nombres entiers . . . 7
1.3.2 Nombres flottants . . . 7
1.3.3 Caractères et chaînes de caractères . . . 7
1.3.4 Expressions constantes . . . 8
1.4 Types fondamentaux . . . 9
1.4.1 Nombres entiers et caractères . . . 9
1.4.2 Types énumérés . . . 11
1.4.3 Nombres flottants . . . 11
1.5 Variables . . . 11
1.5.1 Syntaxe des déclarations . . . 11
1.5.2 Visibilité des variables . . . 12
1.5.3 Allocation et durée de vie des variables . . . 13
1.5.4 Initialisation des variables . . . 13
1.5.5 Variables locales statiques . . . 14
1.5.6 Variables critiques . . . 14
1.5.7 Variables constantes et volatiles . . . 15
1.6 Variables, fonctions et compilation séparée . . . 15
1.6.1 Identificateurs publics et privés . . . 15
1.6.2 Déclaration d'objets externes . . . 16

2 Opérateurs et expressions **18**
2.1 Généralités . . . 18
2.1.1 Lvalue et rvalue . . . 18
2.1.2 Priorité des opérateurs . . . 19
2.2 Présentation détaillée des opérateurs . . . 19
2.2.1 Appel de fonction `()` . . . 19
2.2.2 Indexation `[]` . . . 20
2.2.3 Sélection `.` . . . 21
2.2.4 Sélection dans un objet pointé `->` . . . 21
2.2.5 Négation `!` . . . 22
2.2.6 Complément à 1 `~` . . . 22
2.2.7 Les célèbres `++` et `--` . . . 22
2.2.8 Moins unaire `-` . . . 23
2.2.9 Indirection `*` . . . 23
2.2.10 Obtention de l'adresse `&` . . . 24
2.2.11 Opérateur `sizeof` . . . 24
2.2.12 Conversion de type ("cast" operator) . . . 25
2.2.13 Opérateurs arithmétiques . . . 27
2.2.14 Décalages `<< >>` . . . 28
2.2.15 Comparaisons `== != < <= > >=` . . . 28
2.2.16 Opérateurs de bits `& | ^` . . . 29
2.2.17 Connecteurs logiques `&&` et `||` . . . 30
2.2.18 Expression conditionnelle `? :` . . . 31
2.2.19 Affectation `=` . . . 31
2.2.20 Autres opérateurs d'affectation `+= *=` *etc.* . . . 32
2.2.21 L'opérateur virgule `,` . . . 33
2.3 Autres remarques . . . 34
2.3.1 Les conversions usuelles . . . 34
2.3.2 L'ordre d'évaluation des expressions . . . 34
2.3.3 Les opérations non abstraites . . . 34

3 Instructions **36**
3.1 Syntaxe . . . 36
3.2 Présentation détaillée des instructions . . . 37
3.2.1 Blocs . . . 37
3.2.2 Instruction-expression . . . 37
3.2.3 Etiquettes et instruction `goto` . . . 38
3.2.4 Instruction `if...else...` . . . 38
3.2.5 Instructions `while` et `do...while` . . . 39
3.2.6 Instruction `for` . . . 40
3.2.7 Instruction `switch` . . . 41
3.2.8 Instructions `break` et `continue` . . . 42
3.2.9 Instruction `return` . . . 43

4 Fonctions **44**
4.1 Syntaxe ANSI ou “avec prototype” . . . 44
4.1.1 Définition . . . 44
4.1.2 Type de la fonction et des arguments . . . 44
4.1.3 Appel des fonctions . . . 45
4.1.4 Déclaration “externe” d’une fonction . . . 46
4.2 Syntaxe originale ou “sans prototype” . . . 46
4.2.1 Déclaration et définition . . . 46
4.2.2 Appel . . . 47
4.2.3 Coexistence des deux syntaxes . . . 48
4.3 Arguments des fonctions . . . 48
4.3.1 Passage des arguments . . . 48
4.3.2 Arguments de type tableau . . . 48
4.3.3 Arguments par adresse . . . 49
4.3.4 Arguments en nombre variable . . . 50

5 Objets structurés **52**
5.1 Tableaux . . . 52
5.1.1 Cas général . . . 52
5.1.2 Initialisation des tableaux . . . 52
5.1.3 Chaînes de caractères . . . 53
5.2 Structures et unions . . . 54
5.2.1 Structures . . . 54
5.2.2 Unions . . . 56
5.2.3 Champs de bits . . . 56
5.3 Enumérations . . . 57
5.4 Déclarateurs complexes . . . 58
5.4.1 Cas des déclarations . . . 58
5.4.2 Pointeurs et tableaux constants et volatils . . . 60
5.4.3 La déclaration `typedef` . . . 61
5.4.4 Cas des types désincarnés . . . 62

6 Pointeurs **64**
6.1 Généralités . . . 64
6.1.1 Déclaration et initialisation des pointeurs . . . 64
6.1.2 Les pointeurs génériques et le pointeur `NULL` . . . 65
6.2 Les pointeurs et les tableaux . . . 66
6.2.1 Arithmétique des adresses, indirection et indexation . . . 66
6.2.2 Tableaux dynamiques . . . 68
6.2.3 Tableaux multidimensionnels . . . 69
6.2.4 Tableaux multidimensionnels dynamiques . . . 70
6.2.5 Tableaux de chaînes de caractères . . . 72
6.2.6 Tableaux multidimensionnels formels . . . 73
6.2.7 Tableaux non nécessairement indexés à partir de zéro . . . 74
6.2.8 Matrices non dynamiques de taille inconnue . . . 76
6.3 Les adresses des fonctions . . . 77
6.3.1 Les fonctions et leurs adresses . . . 77
6.3.2 Fonctions formelles . . . 78

6.3.3 Tableaux de fonctions . . . 79
6.3.4 Flou artistique . . . 80
6.4 Structures récursives . . . 81
6.4.1 Déclaration . . . 81
6.4.2 Exemple . . . 81
6.4.3 Structures mutuellement récursives . . . 82

7 Entrées-sorties **84**
7.1 Flots . . . 84
7.1.1 Fonctions générales sur les flots . . . 85
7.1.2 Les unités standard d'entrée-sortie . . . 87
7.2 Lecture et écriture textuelles . . . 87
7.2.1 Lecture et écriture de caractères et de chaînes . . . 87
7.2.2 Ecriture avec format `printf` . . . 89
7.2.3 Lecture avec format `scanf` . . . 92
7.2.4 A propos de la fonction `scanf` et des lectures interactives . . . 94
7.2.5 Les variantes de `printf` et `scanf` . . . 95
7.3 Opérations en mode binaire . . . 96
7.3.1 Lecture-écriture . . . 96
7.3.2 Positionnement dans les fichiers . . . 96
7.4 Exemples . . . 98
7.4.1 Fichiers "en vrac" . . . 98
7.4.2 Fichiers binaires et fichiers de texte . . . 98
7.4.3 Fichiers en accès relatif . . . 100
7.5 Les fichiers de bas niveau d'UNIX . . . 100

8 Autres éléments du langage C **102**
8.1 Le préprocesseur . . . 102
8.1.1 Inclusion de fichiers . . . 102
8.1.2 Définition et appel des "macros" . . . 103
8.1.3 Compilation conditionnelle . . . 104
8.2 La modularité de C . . . 106
8.2.1 Fichiers en-tête . . . 107
8.2.2 Exemple : `stdio.h` . . . 108
8.3 Deux ou trois choses bien pratiques... . . . 110
8.3.1 Les arguments du programme principal . . . 110
8.3.2 Branchements hors fonction : `setjmp.h` . . . 112
8.3.3 Interruptions : `signal.h` . . . 113
8.4 La bibliothèque standard . . . 114
8.4.1 Aide à la mise au point : `assert.h` . . . 115
8.4.2 Fonctions utilitaires : `stdlib.h` . . . 116
8.4.3 Traitement de chaînes : `string.h` . . . 118
8.4.4 Classification des caractères : `ctype.h` . . . 119
8.4.5 Fonctions mathématiques : `math.h` . . . 119
8.4.6 Limites propres à l'implémentation : `limits.h`, `float.h` . . . 119

Imprimé le 19 septembre 2004

`Henri.Garreta@luminy.univ-mrs.fr`

1 Éléments de base

1.1 Structure générale d'un programme

La transformation d'un texte écrit en langage C en un programme exécutable par l'ordinateur se fait en deux étapes : la *compilation* et l'*édition de liens*. La compilation est la traduction des fonctions écrites en C en des procédures équivalentes écrites dans un langage dont la machine peut exécuter les instructions. Le compilateur lit toujours *un* fichier, appelé *fichier source*, et produit *un* fichier, dit *fichier objet*.

Chaque fichier objet est incomplet, insuffisant pour être exécuté, car il contient des appels de fonctions ou des références à des variables qui ne sont pas définies dans le même fichier. Par exemple, le premier programme que vous écrirez contiendra déjà la fonction `printf` que vous n'aurez certainement pas écrite vous-même. L'édition de liens est l'opération par laquelle plusieurs fichiers objets sont mis ensemble pour se compléter mutuellement : un fichier apporte des définitions de fonctions et de variables auxquelles un autre fichier fait référence et réciproquement. L'éditeur de liens (ou *linker*) prend en entrée plusieurs fichiers objets et *bibliothèques* (une variété particulière de fichiers objets) et produit un unique *fichier exécutable*. L'éditeur de liens est largement indépendant du langage de programmation utilisé pour écrire les fichiers sources, qui peuvent même avoir été écrits dans des langages différents.

Chaque fichier source entrant dans la composition d'un programme exécutable est fait d'une succession d'un nombre quelconque d'éléments indépendants, qui sont :

- des directives pour le préprocesseur (lignes commençant par `#`),
- des constructions de types (`struct`, `union`, `enum`, `typedef`),
- des déclarations de variables et de fonctions *externes*,
- des définitions de variables et
- des définitions de fonctions.

Seules les expressions des deux dernières catégories font grossir le fichier objet : les définitions de fonctions laissent leur traduction en langage machine, tandis que les définitions de variables se traduisent par des réservations d'espace, éventuellement garni de valeurs initiales. Les autres directives et déclarations s'adressent au compilateur et il n'en reste pas de trace lorsque la compilation est finie.

En C on n'a donc pas une structure syntaxique englobant tout, comme la construction « `Program` ... `end.` » du langage Pascal ; un programme n'est qu'une collection de fonctions assortie d'un ensemble de variables globales. D'où la question : par où l'exécution doit-elle commencer ? La règle généralement suivie par l'éditeur de liens est la suivante : parmi les fonctions données il doit en exister une dont le nom est `main`. C'est par elle que l'exécution commencera ; le lancement du programme équivaut à l'appel de cette fonction par le système d'exploitation. Notez bien que, à part cela, `main` est une fonction comme les autres, sans aucune autre propriété spécifique ; en particulier, les variables internes à `main` sont locales, tout comme celles des autres fonctions.

Pour finir cette entrée en matière, voici la version C du célèbre programme-qui-dit-bonjour, sans lequel on ne saurait commencer un cours de programmation[1] :

```
#include <stdio.h>

int main() {
    printf("Bonjour\n");
    return 0;
}
```

1.2 Considérations lexicales

1.2.1 Présentation du texte du programme

Le programmeur est maître de la disposition du texte du programme. Des blancs, des tabulations et des sauts à la ligne peuvent être placés à tout endroit où cela ne coupe pas un identificateur, un nombre ou un symbole composé[2].

[1] Le programme montré ici est écrit selon des règles strictes. En fait, la plupart des compilateurs acceptent que `main` soit déclarée `void` au lieu de `int`, ou que ce type ne figure pas, et que l'instruction « `return 0 ;` » n'apparaisse pas explicitement.

[2] Néanmoins, les directives pour le préprocesseur (cf. section 8.1) doivent comporter un # dans la première position de la ligne. Cela ne constitue pas une exception à la règle donnée ici, car le préprocesseur n'est pas le compilateur C et ne travaille pas sur la syntaxe du langage.

Les commentaires commencent par `/*` et se terminent par `*/` :

```
/* Ce texte est un commentaire et sera donc
ignoré par le compilateur */
```

Les commentaires ne peuvent pas être imbriqués : écrit dans un programme, le texte « `/* voici un grand /* et un petit */ commentaire */` » est erroné, car seul « `/* voici un grand /* et un petit */` » sera vu comme un commentaire par le compilateur.

Les langages C et C++ cohabitant dans la plupart des compilateurs actuels, ces derniers acceptent également comme commentaire tout texte compris entre le signe `//` et la fin de la ligne où ce signe apparaît :

```
// Ceci est un commentaire à la mode C++.
```

Le caractère anti-slash `\` précédant *immédiatement* un saut à la ligne masque ce dernier : la ligne suivante est considérée comme devant être concaténée à la ligne courante. Cela est vrai en toute circonstance, y compris à l'intérieur d'une chaîne de caractères. Par exemple, le texte

```
message = "anti\
    constitutionnellement";
```

est compris comme ceci : « `message = "anti    constitutionnellement";` »

1.2.2 Mots-clés

Les mots suivants sont réservés. Leur fonction est prévue par la syntaxe de C et ils ne peuvent pas être utilisés dans un autre but :

```
auto      break     case      char      const     continue  default   do
double    else      enum      extern    float     for       goto      if
int       long      register  return    short     signed    sizeof    static
struct    switch    typedef   union     unsigned  void      volatile  while
```

1.2.3 Identificateurs

Un identificateur est une suite de lettres et chiffres contigus, dont le premier est une lettre. Lorsque seul le compilateur est concerné, c'est-à-dire lorsqu'il s'agit d'identificateurs dont la portée est incluse dans un seul fichier (nous dirons de tels identificateurs qu'ils sont *privés*) :

– en toute circonstance une lettre majuscule est tenue pour différente de la lettre minuscule correspondante ;
– dans les identificateurs, le nombre de caractères discriminants est au moins de 31.

Attention, lorsqu'il s'agit d'identificateurs externes, c'est-à-dire partagés par plusieurs fichiers sources, il est possible que sur un système particulier l'éditeur de liens sous-jacent soit trop rustique pour permettre le respect de ces deux prescriptions.

Le caractère `_` (appelé « blanc souligné ») est considéré comme une lettre ; il peut donc figurer à n'importe quelle place dans un identificateur. Cependant, par convention un programmeur ne doit pas utiliser des identificateurs qui commencent par ce caractère. Cela assure qu'il n'y aura jamais de conflit avec les noms introduits (à travers les fichiers « `.h` ») pour les besoins des bibliothèques, car ces noms commencent par un tel blanc souligné. ♡

1.2.4 Opérateurs

Symboles simples :

```
(     )     [     ]     .     !     ~     <     >     ?     :
=     ,     +     -     *     /     %     |     &     ^
```

Symboles composés :

```
->    ++    --    <=    >=    ==    !=    &&    ||    <<    >>
+=    -=    *=    /=    %=    <<=   >>=   |=    &=    ^=
```

Tous ces symboles sont reconnus par le compilateur comme des opérateurs. Il est interdit d'insérer des caractères blancs à l'intérieur d'un symbole composé. En outre, il est conseillé d'encadrer par des blancs toute utilisation d'un opérateur. Dans certaines circonstances cette règle est plus qu'un conseil, car sa non-observance crée une expression ambiguë.

1.3 Constantes littérales

1.3.1 Nombres entiers

Les constantes littérales numériques entières ou réelles suivent les conventions habituelles, avec quelques particularités.

Les constantes littérales sont sans signe : l'expression −123 est comprise comme l'application de l'opérateur unaire − à la constante 123 ; mais puisque le calcul est fait pendant la compilation, cette subtilité n'a aucune conséquence pour le programmeur. Notez aussi qu'en C original, comme il n'existe pas d'opérateur + unaire, la notation +123 est interdite.

Les constantes littérales entières peuvent aussi s'écrire en octal et en hexadécimal :
– une constante écrite en octal (base 8) commence par `0` (zéro) ;
– une constante écrite en hexadécimal (base 16) commence par `0x` ou `0X`.
Voici par exemple trois manières d'écrire le même nombre :

```
27        033       0x1B
```

Détail à retenir : on ne doit pas écrire de zéro non significatif à gauche d'un nombre : 0123 ne représente pas la même valeur que 123.

Le type d'une constante entière est le plus petit type dans lequel sa valeur peut être représentée. Ou, plus exactement :
– si elle est décimale : si possible `int`, sinon `long`, sinon `unsigned long` ;
– si elle est octale ou hexadécimale : si possible `int`, sinon `unsigned int`, sinon `unsigned long`.
Certains suffixes permettent de changer cette classification :
– `U`, `u` : indique que la constante est d'un type `unsigned` ;
– `L`, `l` : indique que la constante est d'un type `long`.
Exemples : `1L`, `0x7FFFU`. On peut combiner ces deux suffixes : `16UL`.

1.3.2 Nombres flottants

Une constante littérale est l'expression d'un nombre flottant si elle présente, dans l'ordre :
– une suite de chiffres décimaux (la partie entière),
– un point, qui joue le rôle de virgule décimale,
– une suite de chiffres décimaux (la partie fractionnaire),
– une des deux lettres `E` ou `e`,
– éventuellement un signe `+` ou `-`,
– une suite de chiffres décimaux.
Les trois derniers éléments forment l'exposant. Exemple : `123.456E-78`.

On peut omettre :
– la partie entière ou la partie fractionnaire, mais pas les deux,
– le point ou l'exposant, mais pas les deux.
Exemples : `.5e7`, `5.e6`, `5000000.`, `5e6`

Une constante flottante est supposée de type double, à moins de comporter un suffixe explicite :
– les suffixes `F` ou `f` indiquent qu'elle est du type `float` ;
– les suffixes `L` ou `l` indiquent qu'elle est du type `long double`.
Exemples : `1.0L`, `5.0e4f`

1.3.3 Caractères et chaînes de caractères

Une constante de type caractère se note en écrivant le caractère entre apostrophes. Une constante de type chaîne de caractères se note en écrivant ses caractères entre guillemets. Exemples, trois caractères :

```
'A'       '2'       '"'
```

Quatre chaînes de caractères :

```
"A"       "Bonjour à tous !"       ""       "'"
```

On peut faire figurer n'importe quel caractère, même non imprimable, dans une constante caractère ou chaîne de caractères en utilisant les combinaisons suivantes, appelées *séquences d'échappement* :

`\n`	nouvelle ligne (LF)
`\t`	tabulation (HT)
`\b`	espace-arrière (BS)
`\r`	retour-chariot (CR)
`\f`	saut de page (FF)
`\a`	signal sonore (BELL)
`\\`	`\`
`'`	`'`
`"`	`"`
$\backslash d_3 d_2 d_1$	le caractère qui a pour code le nombre octal $d_3 d_2 d_1$. S'il commence par un ou deux zéros et si cela ne crée pas une ambiguïté, on peut aussi le noter $\backslash d_2 d_1$ ou $\backslash d_1$

Par exemple, la chaîne suivante définit la suite des 9 caractères[3] `A`, *escape* (de code ASCII 27), `B`, ", `C`, saut de page, `D`, `\` et `E` :

```
"A\033B\"C\fD\\E"
```

Une constante de type caractère appartient au type `char`, c'est-à-dire *entier représenté sur un octet*. La valeur d'une constante caractère est le nombre qui représente le caractère de manière interne ; de nos jours il s'agit presque toujours du code ASCII[4].

Une constante de type chaîne de caractères représente une suite finie de caractères, de longueur quelconque. Le codage interne d'une chaîne de caractères est le suivant (voyez la figure 1) :
- les caractères constituant la chaîne sont rangés en mémoire, de manière contiguë, dans l'ordre où ils figurent dans la chaîne ;
- un caractère nul est ajouté immédiatement après le dernier caractère de la chaîne, pour en indiquer la fin ;
- la constante chaîne représente alors, à l'endroit où elle est écrite, l'*adresse* de la cellule où a été rangé le premier caractère de la chaîne.

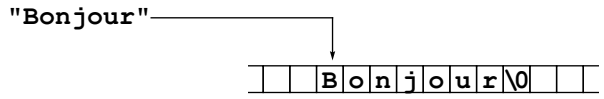

FIG. 1 – Représentation de la chaîne `"Bonjour"`

Par conséquent, une constante chaîne de caractères a pour type celui d'un tableau de caractères (c'est-à-dire « `char[]` ») et pour valeur l'adresse d'une cellule de la mémoire.

Par caractère nul on entend le caractère dont le code interne est 0 ; on peut le noter indifféremment `0`, `'\000'` ou `'\0'` (mais certainement pas `'0'`) ; il est utilisé très fréquemment en C. Notez que, dans une expression, `'\0'` est toujours interchangeable avec `0`.

1.3.4 Expressions constantes

Une expression constante est une expression de l'un des types suivants :
- toute constante littérale ; exemples : `1`, `'A'`, `"HELLO"`, `1.5e-2` ;
- une expression correcte formée par l'application d'un opérateur courant (arithmétique, logique, etc.) à une ou deux expressions constantes ; exemples : `-1`, `'A' - 'a'`, `2 * 3.14159265`, `"HELLO" + 6` ;
- l'expression constituée par l'application de l'opérateur `&` (opérateur de calcul de l'adresse, voyez la section 2.2.10) à une variable statique, à un champ d'une variable statique de type structure ou à un élément d'un tableau statique dont le rang est donné par une expression constante ; exemples : `&x`, `&fiche.nom`, `&table[50]` ;
- l'expression constituée par l'application de l'opérateur `sizeof` à un descripteur de type. Exemples : `sizeof(int)`, `sizeof(char *)` ;
- l'expression constituée par l'application de l'opérateur `sizeof` à une expression quelconque, qui ne sera pas évaluée ; exemples : `sizeof x`, `sizeof(2 * x + 3)`.

Les expressions constantes peuvent être évaluées pendant la compilation. Cela est fait à titre facultatif par les compilateurs de certains langages. En C ce n'est pas facultatif : il est garanti que toute expression constante (et donc toute sous-expression constante d'une expression quelconque) sera effectivement évaluée avant que

[3] Nous verrons qu'en fait cette chaîne comporte un caractère de plus qui en marque la fin.
[4] En standard le langage C ne prévoit pas le codage *Unicode* des caractères.

l'exécution ne commence. En termes de temps d'exécution, l'évaluation des expressions constantes est donc entièrement « gratuite ».

1.4 Types fondamentaux

TYPES DE BASE	
Nombres entiers	
Anonymes	
Petite taille	
– signés	`char`
– non signés	`unsigned char`
Taille moyenne	
– signés	`short`
– non signés	`unsigned short`
Grande taille	
– signés	`long`
– non signés	`unsigned long`
Nommés	
	`enum`
Nombres flottants	
Simples	`float`
Grande précision	`double`
Précision encore plus grande	`long double`
TYPES DÉRIVÉS	
Tableaux	`[ ]`
Fonctions	`( )`
Pointeurs	`*`
Structures	`struct`
Unions	`union`

TAB. 1 – Les types du langage C

Le tableau 1 présente l'ensemble des types connus du compilateur C. L'organisation générale de cet ensemble est évidente : on dispose de deux sortes de types de base, les nombres entiers et les nombres flottants, et d'une famille infinie de types dérivés obtenus en appliquant quelques procédés récursifs de construction soit à des types fondamentaux soit à des types dérivés définis de la même manière.

Cette organisation révèle un trait de l'esprit de C : le pragmatisme l'emporte sur l'esthétisme, parfois même sur la rigueur. Dans d'autres langages, les caractères, les booléens, les constantes symboliques, etc., sont codés de manière interne par des nombres, mais ce fait est officiellement ignoré par le programmeur, qui reste obligé de considérer ces données comme appartenant à des ensembles disjoints. En C on a fait le choix opposé, laissant au programmeur le soin de réaliser lui-même, à l'aide des seuls types numériques, l'implantation des types de niveau supérieur.

1.4.1 Nombres entiers et caractères

La classification des types numériques obéit à deux critères :

- Si on cherche à représenter un ensemble de nombres tous positifs on pourra adopter un type *non signé* ; au contraire si on doit représenter un ensemble contenant des nombres positifs et des nombres négatifs on devra utiliser un type *signé*[5].
- Le deuxième critère de classification des données numériques est la taille requise par leur représentation. Comme précédemment, c'est un attribut d'un ensemble, et donc d'une variable devant représenter tout élément de l'ensemble, non d'une valeur particulière. Par exemple, le nombre 123 considéré comme un élément de l'ensemble {0 ... 65535} est plus encombrant que le même nombre 123 quand il est considéré comme un élément de l'ensemble {0 ... 255}.

[5]On dit parfois qu'une donnée « est un entier signé » ou « est un entier non signé ». C'est un abus de langage : le caractère signé ou non signé n'est pas un attribut d'un nombre (un nombre donné est positif ou négatif, c'est tout) mais de l'ensemble de nombres qu'on a choisi de considérer et, par extension, de toute variable censée pouvoir représenter n'importe quelle valeur de cet ensemble.

Avec N chiffres binaires (ou *bits*) on peut représenter :
– soit les 2^N nombres positifs $0, 1, ... 2^N - 1$ (cas non signé) ;
– soit les 2^N nombres positifs et négatifs $-2^{N-1}, ... 2^{N-1} - 1$ (cas signé).
De plus, la représentation signée et la représentation non signée des éléments communs aux deux domaines (les nombres $0, 1, ... 2^{N-1} - 1$) coïncident.

Le type caractère. Un objet de type `char` peut être défini, au choix, comme
– un nombre entier pouvant représenter n'importe quel caractère du jeu de caractères de la machine utilisée ;
– un nombre entier occupant la plus petite cellule de mémoire adressable séparément[6]. Sur les machines actuelles les plus répandues cela signifie généralement un octet (8 bits).

Le plus souvent, un `char` est un entier signé ; un `unsigned char` est alors un entier non signé. Lorsque les `char` sont par défaut non signés, la norme ANSI prévoit la possibilité de déclarer des `signed char`.

On notera que la signification d'un `char` en C, un entier petit, est très différente de celle d'un `char` en Pascal (dans ce langage, l'ensemble des caractères et celui des nombres sont disjoints). En C, `ch` étant une variable de type `char`, rien ne s'oppose à l'écriture de l'expression `ch - 'A' + 32` qui est tout à fait homogène, puisque entièrement faite de nombres.

Le caractère « impossible ». Toutes les valeurs qu'il est possible de ranger dans une variable de type `char` sont en principe des caractères légaux. Or la plupart des programmes qui lisent des caractères doivent être capables de manipuler une valeur supplémentaire, distincte de tous les « vrais » caractères, signifiant « la fin des données ». Pour cette raison, les variables et fonctions qui représentent ou renvoient des caractères sont souvent déclarées `int`, non `char` : n'importe quelle valeur appartenant au type `int` mais n'appartenant pas au type `char` peut alors servir d'indicateur de fin de données. Par exemple, une telle valeur est définie dans le fichier `stdio.h`, c'est la constante symbolique `EOF`.

Les entiers courts et longs. Il est garanti que toute donnée représentable dans le type `short` est représentable aussi dans le type `long`[7] (en bref : un `long` n'est pas plus court qu'un `short` !), mais la taille exacte des données de ces types n'est pas fixée par la norme du langage.

De nos jours on trouve souvent :

`unsigned short`	: 16 bits pour représenter un nombre entier compris entre 0 et 65.535
`short`	: 16 bits pour représenter un nombre entier compris entre -32.768 et 32.767
`unsigned long`	: 32 bits pour représenter un nombre entier entre 0 et 4.294.967.296
`long`	: 32 bits pour représenter un entier entre -2.147.483.648 et 2.147.483.647

Le type int. En principe, le type `int` correspond à la taille d'entier la plus efficace, c'est-à-dire la plus adaptée à la machine utilisée. Sur certains systèmes et compilateurs `int` est synonyme de `short`, sur d'autres il est synonyme de `long`.

Le type int peut donc poser un problème de portabilité[8] : le même programme, compilé sur deux machines distinctes, peut avoir des comportements différents. D'où un conseil important : n'utilisez le type `int` que pour des variables *locales* destinées à contenir des valeurs raisonnablement *petites* (inférieures en valeur absolue à 32767) . Dans les autres cas il vaut mieux expliciter `char`, `short` ou `long` selon le besoin.

A propos des booléens. En C il n'existe donc pas de type booléen spécifique. Il faut savoir qu'à tout endroit où une expression booléenne est requise (typiquement, dans des instructions comme *if* ou *while*) on peut faire figurer n'importe quelle expression ; elle sera tenue pour vraie si elle est non nulle, elle sera considérée fausse sinon. Ainsi, dans un contexte conditionnel,

expr

(c'est-à-dire *expr* « vraie ») équivaut à

expr `!= 0`

(*expr* différente de 0). Inversement, lorsqu'un opérateur (égalité, comparaison, etc.) produit une valeur booléenne,

[6] A retenir : un objet de type `char` est « unitaire » aussi bien du point de vue des tailles que de celui des adresses. Quelle que soit la machine utilisée, le compilateur C fera en sorte que le programmeur voie ces objets de la manière suivante : si `t` est un tableau de `char`, la taille (au sens de l'opérateur `sizeof`, cf. section 2.2.11) de `t[0]` vaut une unité de taille, et l'écart entre les adresses de `t[1]` et `t[0]` vaut une unité d'adressage. On peut dire que ces propriétés *définissent* le type char (ou, si vous préférez, les unités de taille et d'adressage).

[7] Si on considère un type comme l'*ensemble* de ses valeurs, on a donc les inclusions larges `char` ⊆ `short` ⊆ `long` (et aussi `float` ⊆ `double` ⊆ `long double`).

[8] Un programme écrit pour une machine ou un système A est dit portable s'il suffit de le recompiler pour qu'il tourne correctement sur une machine différente B. Par exemple, « `putchar('A') ;` » est une manière portable d'obtenir l'affichage du caractère `A`, tandis que « `putchar(65) ;` » est (sur un système utilisant le code ASCII) une manière non portable d'obtenir le même affichage. Etre portable est un critère de qualité et de fiabilité important. On invoque l'efficacité pour justifier l'écriture de programmes non portables ; l'expérience prouve que, lorsque son écriture est possible, un programme portable est toujours meilleur qu'un programme non portable prétendu équivalent.

il rend 0 pour faux et 1 pour vrai.

Signalons aux esthètes que le fichier `<types.h>` comporte les déclarations :

```
enum { false, true };
typedef unsigned char Boolean;
```

qui introduisent la constante `false` valant 0, la constante `true` valant 1 et le type `Boolean` comme le type le moins encombrant dans lequel on peut représenter ces deux valeurs.

1.4.2 Types énumérés

Un type énuméré, ou énumération, est constitué par une famille finie de nombres entiers, chacun associé à un identificateur qui en est le nom. Mis à part ce qui touche à la syntaxe de leur déclaration, il n'y a pas grand-chose à dire à leur sujet. La syntaxe de la déclaration des énumérations est expliquée à la section 5.3. Par exemple, l'énoncé :

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

introduit un type énuméré, appelé `enum jour_semaine`, constitué par les constantes `lundi` valant 0, `mardi` valant 1, `mercredi` valant 2, etc. Ainsi, les expressions `mardi + 2` et `jeudi` représentent la même valeur.

Les valeurs d'un type énuméré se comportent comme des constantes entières ; elles font donc double emploi avec celles qu'on définit à l'aide de `#define` (cf. section 8.1.2). Leur unique avantage réside dans le fait que certains compilateurs détectent parfois, mais ce n'est pas exigé par la norme, les mélanges entre objets de types énumérés distincts ; ces types sont alors le moyen d'augmenter la sécurité des programmes.

A propos des types énumérés voyez aussi la section 5.3.

1.4.3 Nombres flottants

La norme ANSI prévoit trois types de nombres flottants : `float` (simple précision), `double` (double précision) et `long double` (précision étendue). La norme ne spécifie pas les caractéristiques de tous ces types. Il est garanti que toute valeur représentable dans le type `float` est représentable sans perte d'information dans le type `double`, et toute valeur représentable dans le type `double` l'est dans le type `long double`.

Typiquement, sur des systèmes de taille moyenne, un `float` occupe 32 bits et un `double` 64, ce qui donne par exemple des `float` allant de `-1.70E38` à `-0.29E-38` et de `0.29E-38` à `1.70E38` avec 7 chiffres décimaux significatifs, et des double allant de `-0.90E308` à `-0.56E-308` et de `0.56E-308` à `0.90E308` avec 15 chiffres décimaux significatifs.

Les `long double` correspondent généralement aux flottants de grande précision manipulés par certains coprocesseurs arithmétiques ou les bibliothèques de sous-programmes qui les simulent. Mais il n'est pas exclu que sur un système particulier un `long double` soit la même chose qu'un `double`.

1.5 Variables

1.5.1 Syntaxe des déclarations

La forme complète de la déclaration d'une variable sera expliquée à la section 5.4. Dans le cas le plus simple on trouve

spécification *var-init* , *var-init* , ... *var-init* ;

où spécification est de la forme :

{ `auto` | `register` | `static` | `extern` | *rien* } { `const` | `volatile` | *rien* } { { { `unsigned` | `signed` | *rien* } { `char` | `short` | `long` | `int` } } | `float` | `double` | `long double` }

et chaque *var-init* est de la forme :

identificateur { `=` *expression* | *rien* }

Exemples :

```
int x, y = 0, z;
extern float a, b;
static unsigned short cpt = 1000;
```

Les déclarations de variables peuvent se trouver :
- en dehors de toute fonction, il s'agit alors de *variables globales* ;
- à l'intérieur d'un bloc, il s'agit alors de *variables locales* ;
- dans l'en-tête d'une fonction, il s'agit alors d'*arguments formels*, placés
  - soit dans les parenthèses de l'en-tête (fonction définie en syntaxe ANSI avec un prototype),
  - soit entre le nom de la fonction et le { initial (fonction définie en syntaxe originale ou sans prototype).

Exemple. Avec prototype :

```
long i = 1;

int une_fonction(int j) {
    short k;
    ...
}
```

Sans prototype :

```
long i = 1;

int une_fonction(j)
    int j;
{
    short k;
    ...
}
```

Ci-dessus, `i` est une variable globale, `k` une variable locale et `j` un argument formel de `une_fonction`.

1.5.2 Visibilité des variables

La question de la visibilité des identificateurs (c'est-à-dire « quels sont les identificateurs auxquels on peut faire référence en un point d'un programme ? ») est réglée en C comme dans la plupart des langages comportant la structure de bloc, avec une simplification : les fonctions ne peuvent pas être imbriquées les unes dans les autres, et une complication : tout bloc peut comporter ses propres définitions de variables locales.

Un bloc est une suite de déclarations et d'instructions encadrée par une accolade ouvrante "{" et l'accolade fermante "}" correspondante. Le corps d'une fonction est lui-même un bloc, mais d'autres blocs peuvent être imbriqués dans celui-là.

Variables locales. Tout bloc peut comporter un ensemble de déclarations de variables, qui sont alors dites *locales* au bloc en question. Une variable locale ne peut être référencée que depuis l'intérieur du bloc où elle est définie ; en aucun cas on ne peut y faire référence depuis un point extérieur à ce bloc. Dans le bloc où il est déclaré, le nom d'une variable locale *masque* toute variable de même nom définie dans un bloc englobant le bloc en question.

Toutes les déclarations de variables locales à un bloc doivent être écrites au début du bloc, avant la première instruction.

Arguments formels. Pour ce qui concerne leur visibilité, les arguments formels des fonctions sont considérés comme des variables locales du niveau le plus haut, c'est-à-dire des variables déclarées au début du bloc le plus extérieur[9]. Un argument formel est accessible de l'intérieur de la fonction, partout où une variable locale plus profonde ne le masque pas. En aucun cas on ne peut y faire référence depuis l'extérieur de la fonction.

Variables globales. Le nom d'une variable globale ou d'une fonction peut être utilisé depuis n'importe quel point compris entre sa déclaration (pour une fonction : la fin de la déclaration de son en-tête) et la fin du fichier où la déclaration figure, sous réserve de ne pas être masquée par une variable locale ou un argument formel de même nom.

La question de la visibilité inter-fichiers est examinée à la section 1.6. On peut noter d'ores et déjà qu'elle ne se pose que pour les variables globales et les fonctions, et qu'elle concerne l'édition de liens, non la compilation,

[9]Par conséquent, on ne doit pas déclarer un argument formel et une variable locale du niveau le plus haut avec le même nom.

car le compilateur ne traduit qu'un fichier source à la fois et, pendant la traduction d'un fichier, il ne « voit » pas les autres.

1.5.3 Allocation et durée de vie des variables

Les variables globales sont toujours statiques, c'est-à-dire permanentes : elles existent pendant toute la durée de l'exécution. Le système d'exploitation se charge, immédiatement avant l'activation du programme, de les allouer dans un espace mémoire de taille adéquate, éventuellement garni de valeurs initiales.

A l'opposé, les variables locales et les arguments formels des fonctions sont *automatiques* : l'espace correspondant est alloué lors de l'activation de la fonction ou du bloc en question et il est rendu au système lorsque le contrôle quitte cette fonction ou ce bloc. Certains qualifieurs (`static`, `register`, voir les sections 1.5.5 et 1.5.6) permettent de modifier l'allocation et la durée de vie des variables locales.

Remarque. On note une grande similitude entre les variables locales et les arguments formels des fonctions : ils ont la même visibilité et la même durée de vie. En réalité c'est presque la même chose : les arguments formels sont de vraies variables locales avec l'unique particularité d'être automatiquement initialisés (par les valeurs des arguments effectifs) lors de l'activation de la fonction.

1.5.4 Initialisation des variables

Variables statiques. En toute circonstance la déclaration d'une variable statique peut indiquer une valeur initiale à ranger dans la variable. Cela est vrai y compris pour des variables de types complexes (tableaux ou structures). Exemple :

```
double x = 0.5e3;
int t[5] = { 11, 22, 33, 44, 55 };
```

Bien que la syntaxe soit analogue, une telle initialisation n'a rien en commun avec une affectation comme celles qui sont faites durant l'exécution du programme. Il s'agit ici uniquement de préciser la valeur qui doit être déposée dans l'espace alloué à la variable, *avant* que l'exécution ne commence. Par conséquent :

- la valeur initiale doit être définie par une expression constante (calculable durant la compilation) ;
- une telle initialisation est entièrement gratuite, elle n'a aucune incidence ni sur la taille ni sur la durée du programme exécutable produit.

Les variables statiques pour lesquelles aucune valeur initiale n'est indiquée sont remplies de zéros. L'interprétation de ces zéros dépend du type de la variable.

Variables automatiques. Les arguments formels des fonctions sont automatiquement initialisés lors de leur création (au moment de l'appel de la fonction) par les valeurs des arguments effectifs. Cela est la définition même des arguments des fonctions.

La déclaration d'une variable locale peut elle aussi comporter une initialisation. Mais il ne s'agit pas de la même sorte d'initialisation que pour les variables statiques : l'initialisation représente ici une affectation tout à fait ordinaire. Ainsi, placée à l'intérieur d'un bloc, la construction

```
int i = exp;      /* déclaration + initialisation */
```

équivaut au couple

```
int i;            /* déclaration */
...
i = exp ;         /* affectation */
```

Par conséquent :

- l'expression qui donne la valeur initiale n'a pas à être constante, puisqu'elle est évaluée à l'exécution, chaque fois que la fonction ou le bloc est activé ;
- une telle initialisation « coûte » le même prix que l'affectation correspondante, c'est-à-dire le temps d'évaluation de l'expression qui définit la valeur initiale.

Les variables automatiques pour lesquelles aucune valeur initiale n'est indiquée sont allouées avec une valeur imprévisible.

Remarque. Dans le C original, une variable automatique ne peut être initialisée que si elle est simple (c'est-à-dire autre que tableau ou structure). Cette limitation ne fait pas partie du C ANSI.

1.5.5 Variables locales statiques

Le qualifieur `static`, placé devant la déclaration d'une variable *locale*, produit une variable qui est
– pour sa visibilité, locale ;
– pour sa durée de vie, statique (c'est-à-dire permanente).

Elle n'est accessible que depuis l'intérieur du bloc où elle est déclarée, mais elle est créée au début de l'activation du programme et elle existe aussi longtemps que dure l'exécution de celui-ci. Exemple :

```
void bizarre1(void) {
    static int cpt = 1000;
    printf("%d ", cpt);
    cpt++;
}
```

Lorsque la déclaration d'une telle variable comporte une initialisation, il s'agit de l'initialisation d'une variable statique : elle est effectuée une seule fois avant l'activation du programme. D'autre part, une variable locale statique conserve sa valeur entre deux activations consécutives de la fonction. Ainsi, des appels successifs de la fonction ci-dessus produisent l'affichage des valeurs 1000, 1001, 1002, etc. On aurait pu obtenir un effet analogue avec le programme

```
int cpt = 1000;

void bizarre2(void) {
    printf("%d ", cpt);
    cpt++;
}
```

mais ici la variable `cpt` est globale et peut donc être modifiée inconsidérément par une autre fonction, ou entrer en conflit avec un autre objet de même nom, tandis que dans la première version elle n'est visible que depuis l'intérieur de la fonction et donc à l'abri des manipulations maladroites et des collisions de noms. On notera pour finir que la version suivante est erronée :

```
void bizarre3(void) {
    int cpt = 1000;
    printf("%d ", cpt);
    cpt++;
}
```

En effet, tous les appels de `bizarre3` afficheront la même valeur 1000.

Attention. Malgré tout le bien qu'on vient d'en dire, les variables locales statiques ont une particularité potentiellement fort dangereuse : il en existe *une seule instance* pour toutes les activations de la fonction dans laquelle elles sont déclarées. Ainsi, dans l'exemple suivant :

```
void fonction_suspecte(void) {
    static int i;
    ...
 α  fonction_suspecte();  β
    ...
}
```

la valeur de la variable `i` avant et après l'appel de `fonction_suspecte` (c'est-à-dire aux points α et β) peut ne pas être la même, car la deuxième activation de `fonction_suspecte` accède aussi à `i`. Cela est tout à fait inhabituel pour une variable locale. Conséquence à retenir : *les variables locales statiques se marient mal avec la récursivité.*

1.5.6 Variables critiques

Le qualifieur `register` précédant une déclaration de variable informe le compilateur que la variable en question est très fréquemment accédée pendant l'exécution du programme et qu'il y a donc lieu de prendre toutes les dispositions utiles pour en accélérer l'accès. Par exemple, dans certains calculateurs de telles variables sont logées dans un registre de l'unité centrale de traitement (CPU) plutôt que dans la mémoire centrale ; de cette manière l'accès à leur valeur ne met pas en œuvre le *bus* de la machine.

Les variables ainsi déclarées doivent être *locales* et d'un *type simple* (nombre, pointeur). Elles sont automatiquement initialisées à zéro chaque fois qu'elles sont créées. Le compilateur accorde ce traitement spécial aux

variables dans l'ordre où elles figurent dans les déclarations. Lorsque cela n'est plus possible (par exemple, parce que tous les registres de la CPU sont pris) les déclarations `register` restantes sont ignorées. Il convient donc d'appliquer ce qualifieur aux variables les plus critiques d'abord. Exemple :

```
char *strcpy(char *dest, char *srce) {
    register char *d = dest, *s = srce;
    while ((*d++ = *s++) != 0)
        ;
    return dest;
}
```

ATTENTION. L'utilisation du qualifieur `register` est intéressante lorsque l'on doit utiliser un compilateur rustique, peu « optimisateur ». Or de nos jours les compilateurs de C ont fini par devenir très perfectionnés et intègrent des algorithmes d'optimisation, parmi lesquels la détermination des variables critiques et leur allocation dans les registres de la CPU. Il s'avère alors que le programmeur, en appliquant le qualifieur `register` à ses variables préférées (qu'il croit critiques alors qu'elles ne le sont pas réellement), gène le travail du compilateur et obtient un programme moins efficace que s'il n'avait jamais utilisé ce qualifieur.

1.5.7 Variables constantes et volatiles

Le qualifieur `const` placé devant une variable ou un argument formel informe le compilateur que la variable ou l'argument en question ne changera pas de valeur tout au long de l'exécution du programme ou de l'activation de la fonction. Ce renseignement permet au compilateur d'optimiser la gestion de la variable, la nature exacte d'une telle optimisation n'étant pas spécifiée. Par exemple un compilateur peut juger utile de ne pas allouer du tout une variable qualifiée const et de remplacer ses occurrences par la valeur initiale[10] indiquée lors de la déclaration. Il est conseillé de toujours *déclarer* `const` *les variables et les arguments formels qui peuvent l'être.*

NOTE. C'est regrettable mais, pour la plupart des compilateurs, une variable qualifiée `const` n'est pas tout à fait une expression constante au sens de la section 1.3.4. En particulier, pour ces compilateurs une variable, même qualifiée `const`, ne peut pas être utilisée pour indiquer le nombre d'éléments dans une déclaration de tableau.

Le C ANSI introduit aussi les notions de *pointeur constant* et de *pointeur sur constante*, expliquées à la section 5.4.2.

Le sens du qualifieur `volatile` dépend lui aussi de l'implémentation. Il diminue le nombre d'hypothèses, et donc d'optimisations, que le compilateur peut faire sur une variable ainsi qualifiée. Par exemple toute variable dont la valeur peut être modifiée de manière asynchrone (dans une fonction de détection d'interruption, ou par un canal d'entrée-sortie, etc.) doit être qualifiée `volatile`, sur les systèmes où cela a un sens. Cela prévient le compilateur que sa valeur peut changer mystérieusement, y compris dans une section du programme qui ne comporte aucune référence à cette variable.

Les compilateurs sont tenus de signaler toute tentative décelable de modification d'une variable `const`. Mis à part cela, sur un système particulier ces deux qualifieurs peuvent n'avoir aucun autre effet. Ils n'appartiennent pas au C original.

1.6 Variables, fonctions et compilation séparée

1.6.1 Identificateurs publics et privés

Examinons maintenant les règles qui régissent la visibilité inter-fichiers des identificateurs. La question ne concerne que les noms de variables et de fonctions, car les autres identificateurs (noms de structures, de types, etc.) n'existent que pendant la compilation et ne peuvent pas être partagés par deux fichiers. Il n'y a pas de problème pour les variables locales, dont la visibilité se réduit à l'étendue de la fonction ou du bloc contenant leur définition. Il ne sera donc question que des noms des variables globales et des noms des fonctions.

JARGON. Identificateurs publics et privés. Un nom de variable ou de fonction défini dans un fichier source et pouvant être utilisé dans d'autres fichiers sources est dit *public*. Un identificateur qui n'est pas public est appelé *privé*.

REGLE 1.

[10] La déclaration d'une variable `const` doit nécessairement comporter une initialisation car sinon, une telle variable ne pouvant pas être affectée par la suite, elle n'aurait jamais de valeur définie.

– Sauf indication contraire, tout identificateur global est public ;
– le qualifieur `static`, précédant la déclaration d'un identificateur global, rend celui-ci privé.

On prendra garde au fait que le qualifieur `static` n'a pas le même effet quand il s'applique à un identificateur local (`static` change la durée de vie, d'automatique en statique, sans toucher à la visibilité) et quand il s'applique à un identificateur global (`static` change la visibilité, de publique en privée, sans modifier la durée de vie).

Lorsqu'un programme est décomposé en plusieurs fichiers sources il est fortement conseillé, pour ne pas dire obligatoire, d'utiliser le qualifieur `static` pour *rendre privés tous les identificateurs qui peuvent l'être*. Si on ne suit pas cette recommandation on verra des fichiers qui étaient corrects séparément devenir erronés lorsqu'ils sont reliés, uniquement parce qu'ils partagent à tort des identificateurs publics.

1.6.2 Déclaration d'objets externes

Nous ne considérons donc désormais que les noms publics. Un identificateur référencé dans un fichier alors qu'il est défini dans un autre fichier est appelé *externe*. En général, les noms externes doivent faire l'objet d'une déclaration : le compilateur ne traitant qu'un fichier à la fois, les propriétés de l'objet externe doivent être indiquées pour que la compilation puisse avoir lieu correctement.

Jargon. *Définition et déclaration d'une variable ou d'une fonction.* Aussi bien une déclaration qu'une définition d'un nom de variable ou de fonction est une formule qui spécifie la classe syntaxique (variable ou fonction) et les attributs (type, valeur initiale, etc.) de l'identificateur en question. En plus de cela :
– une *définition* produit la création de l'objet dont l'identificateur est le nom ;
– une *déclaration* se limite à indiquer que l'objet en question a dû être créé dans un autre fichier qui sera fourni lors de l'édition de liens.

(« Créer » une variable ou une fonction c'est réserver l'espace correspondant, rempli par l'éventuelle valeur initiale de la variable ou par le code de la fonction).

Regle 2.
– Toute variable doit avoir été *définie* (c'est-à-dire déclarée normalement) ou *déclarée externe* avant son utilisation ;
– une fonction peut être référencée alors qu'elle n'a encore fait l'objet d'aucune définition ni déclaration externe ; elle est alors supposée être
  – externe,
  – à résultat entier (`int`),
  – sans prototype (cf. section 4.2) ;
– par conséquent, si une fonction n'est pas à résultat entier alors elle *doit* être soit définie soit déclarée externe avant son appel, même si elle est ultérieurement définie dans le fichier où figure l'appel.

La déclaration externe d'une variable s'obtient en faisant précéder une déclaration ordinaire du mot-clé `extern`. Exemple :

```
extern unsigned long n;
```

Dans un autre fichier cette variable aura été définie :

```
unsigned long n;
```

La déclaration externe d'une variable doit être identique, au mot `extern` près, à sa définition. Sauf pour les deux points suivants :
– une déclaration externe ne doit pas comporter d'initialisateur (puisque la déclaration externe n'alloue pas la variable),
– dans une déclaration externe de tableau, il est inutile d'indiquer la taille de celui-ci (puisque la déclaration externe n'alloue pas le tableau).

Exemple. Dans le fichier où sont définies les variables `n` et `table`, on écrira :

```
unsigned long n = 1000;
int table[100];
```

Dans un autre fichier, où ces variables sont uniquement référencées, on écrira :

```
extern unsigned long n;
extern int table[];
```

La déclaration externe d'une fonction s'obtient en écrivant l'en-tête de la fonction, précédé du mot `extern` et suivi d'un point-virgule ; le mot `extern` est facultatif. Exemple : définition de la fonction


```
double carre(double x) {
    return x * x;
}
```

Déclaration externe dans un autre fichier :

```
double carre(double x);
```

ou

```
double carre(double);
```

ou l'un ou l'autre de ces énoncés, précédé du mot `extern`.

En syntaxe originale (c'est-à-dire « sans prototype ») il faut en outre ne pas écrire les arguments formels. Définition :

```
double carre(x)
    double x;
    {
    return x * x;
    }
```

Déclaration externe dans un autre fichier :

```
double carre();
```

Regle 3. Dans l'ensemble des fichiers qui constituent un programme, chaque nom public :
– doit faire l'objet d'une et une seule définition ;
– peut être déclaré externe (y compris dans le fichier où il est défini) un nombre quelconque de fois.

Cette règle volontariste est simple et elle exprime la meilleure façon de programmer. Il faut savoir cependant que chaque système tolère des écarts, qui révèlent surtout la rusticité de l'éditeur de liens sous-jacent. La clarté des concepts et la fiabilité des programmes y perdent beaucoup.

Un comportement fréquent est le suivant : appelons momentanément « déclaration-définition » une expression générale de la forme

$$\texttt{extern}_{opt}\ \ \textit{declaration} \left\{ \begin{array}{c} \texttt{=}\ \textit{initialisateur} \\ \textit{rien} \end{array} \right\} ;$$

Nous pouvons donner la règle relâchée :

Regle 3'. Dans l'ensemble des fichiers qui constituent un programme, chaque nom public peut faire l'objet d'un nombre quelconque de déclarations-définitions, mais :
– il doit y avoir au moins une déclaration-définition sans le mot-clé `extern` ;
– il peut y avoir au plus une déclaration-définition comportant un initialisateur.

Des techniques et conseils pour écrire des programmes modulaires en C sont exposés à la section 8.2.

2 Opérateurs et expressions

2.1 Généralités

Dans cette section nous étudions les opérateurs et les expressions du langage C. Les expressions simples sont les constantes littérales (`0`, `'A'`, `0.31416e1`, etc.), les constantes symboliques (`lundi`, `false`, etc.) et les noms de variables (`x`, `nombre`, etc.). Les opérateurs servent à construire des expressions complexes, comme `2 * x + 3` ou `sin(0.31416e1)`. Les propriétés d'une expression complexe découlent essentiellement de la nature de l'opérateur qui chapeaute l'expression.

On peut distinguer les expressions *pures* des expressions *avec effet de bord*[11]. Dans tous les cas une expression représente une valeur. Une expression pure ne fait que cela : l'état du système est le même avant et après son évaluation. Au contraire, une expression à effet de bord modifie le contenu d'une ou plusieurs variables. Par exemple, l'expression `y + 1` est pure, tandis que l'affectation `x = y + 1` (qui en C est une expression) est à effet de bord, car elle modifie la valeur de la variable `x`. Comme nous le verrons, en C un grand nombre d'expressions ont de tels effets.

Remarque 1. Les opérateurs dont il sera question ici peuvent aussi apparaître dans les déclarations, pour la construction des types dérivés (tableaux, pointeurs et fonctions) comme dans la déclaration complexe :

```
char (*t[20])();
```

La signification des expressions ainsi écrites est alors très différente de celle des expressions figurant dans la partie exécutable des programmes, mais on peut signaler d'ores et déjà que toutes ces constructions obéissent aux mêmes règles de syntaxe, notamment pour ce qui concerne la priorité des opérateurs et l'usage des parenthèses. La question des déclarations complexes sera vue à la section 5.4.

Remarque 2. C'est une originalité de C que de considérer les « désignateurs » complexes (les objets pointés, les éléments des tableaux, les champs des structures, etc.) comme des expressions construites avec des opérateurs et obéissant à la loi commune, notamment pour ce qui est des priorités. On se souvient qu'en Pascal les signes qui permettent d'écrire de tels désignateurs (c'est-à-dire les sélecteurs `[]`, `^` et `.`) n'ont pas statut d'opérateur. Il ne viendrait pas à l'idée d'un programmeur Pascal d'écrire `2 + (t[i])` afin de lever une quelconque ambiguïté sur la priorité de `+` par rapport à celle de `[]`, alors qu'en C de telles expressions sont habituelles. Bien sûr, dans `2 + (t[i])` les parenthèses sont superflues, car la priorité de l'opérateur `[]` est supérieure à celle de `+`, mais ce n'est pas le cas dans `(2 + t)[i]`, qui est une expression également légitime.

2.1.1 Lvalue et rvalue

Toute expression possède au moins deux attributs : un type et une valeur. Par exemple, si `i` est une variable entière valant `10`, l'expression `2 * i + 3` possède le type entier et la valeur `23`.

Dans la partie exécutable des programmes on trouve deux sortes d'expressions :

- *Lvalue* (expressions signifiant « le contenu de... »). Certaines expressions sont représentées par une formule qui détermine un emplacement dans la mémoire ; la valeur de l'expression est alors définie comme le *contenu* de cet emplacement. C'est le cas des noms des variables, des composantes des enregistrements et des tableaux, etc. Une *lvalue* possède trois attributs : une adresse, un type et une valeur. Exemples : `x`, `table[i]`, `fiche.numero`.
- *Rvalue* (expressions signifiant « la valeur de... »). D'autres expressions ne sont pas associées à un emplacement de la mémoire : elles ont un type et une valeur, mais pas d'adresse. C'est le cas des constantes et des expressions définies comme le résultat d'une opération arithmétique. Une `rvalue` ne possède que deux attributs : type, valeur. Exemples : `12`, `2 * i + 3`.

Toute *lvalue* peut être vue comme une *rvalue* ; il suffit d'ignorer le contenant (adresse) pour ne voir que le contenu (type, valeur). La raison principale de la distinction entre *lvalue* et *rvalue* est la suivante : seule une *lvalue* peut figurer à gauche du signe `=` dans une affectation ; n'importe quelle *rvalue* peut apparaître à droite. Cela justifie les appellations *lvalue* (« left value ») et *rvalue* (« right value »).

Les sections suivantes préciseront, pour chaque sorte d'expression complexe, s'il s'agit ou non d'une *lvalue*. Pour les expressions simples, c'est-à-dire les constantes littérales et les identificateurs, la situation est la suivante :

- sont des *lvalue* :
  - les noms des variables *simples* (nombres et pointeurs),
  - les noms des variables d'un type struct ou union

[11] *Effet de bord* est un barbarisme ayant pour origine l'expression anglaise *side effect* qu'on peut traduire par *effet secondaire souvent un peu caché, parfois dangereux.*

– ne sont pas des *lvalue* :
  – les constantes,
  – les noms des variables de type tableau,
  – les noms des fonctions.

2.1.2 Priorité des opérateurs

C comporte de très nombreux opérateurs. La plupart des caractères spéciaux désignent des opérations et subissent les mêmes règles syntaxiques, notamment pour ce qui concerne le jeu des priorités et la possibilité de parenthésage. La table 2 montre l'ensemble de tous les opérateurs, classés par ordre de priorité.

prior	*opérateurs*											*sens de l'associativité*
15	`()`	`[]`	`.`	`->`								→
14	`!`	`~`	`++`	`--`	$-_{un}$	$*_{un}$	$\&_{un}$	`sizeof (type)`				←
13	$*_{bin}$	`/`	`%`									→
12	`+`	$-_{bin}$										→
11	`<<`	`>>`										→
10	`<`	`<=`	`>`	`>=`								→
9	`==`	`!=`										→
8	$\&_{bin}$											→
7	`^`											→
6	`|`											→
5	`&&`											→
4	`||`											→
3	`? :`											←
2	`=`	`*=`	`/=`	`%=`	`+=`	`-=`	`<<=`	`>>=`	`&=`	`^=`	`|=`	←
1	`,`											→

TAB. 2 – Opérateurs du langage C

REMARQUE. Dans certains cas un même signe, comme `-`, désigne deux opérateurs, l'un unaire (à un argument), l'autre binaire (à deux arguments). Dans le tableau 2, les suffixes *un* et *bin* précisent de quel opérateur il s'agit.

Le signe → indique l'associativité « de gauche à droite » ; par exemple, l'expression `x - y - z` signifie `(x - y) - z`. Le signe ← indique l'associativité de « droite à gauche » ; par exemple, l'expression `x = y = z` signifie `x = (y = z)`.

Notez que le sens de l'associativité des opérateurs précise la signification d'une expression, mais en aucun cas la *chronologie* de l'évaluation des sous-expressions.

2.2 Présentation détaillée des opérateurs

2.2.1 Appel de fonction `()`

Opération : application d'une fonction à une liste de valeurs. Format :

$$exp_0 \texttt{ (} exp_1 \texttt{ , ... } exp_n \texttt{)}$$

exp_1 , ... exp_n sont appelés les *arguments effectifs* de l'appel de la fonction. exp_0 doit être de type *fonction rendant une valeur de type T* ou bien[12] *adresse d'une fonction rendant une valeur de type T*. Alors exp_0 `(` exp_1 , ... exp_n `)` possède le type *T*. La valeur de cette expression découle de la définition de la fonction (à l'intérieur de la fonction, cette valeur est précisée par une ou plusieurs instructions « `return` *exp* `;` »).

L'expression exp_0 `(` exp_1 , ... exp_n `)` n'est pas une *lvalue*.

Exemple :

```
y = carre(2 * x) + 3;
```

sachant que `carre` est le nom d'une fonction rendant un `double`, l'expression `carre(2 * x)` a le type `double`. Un coup d'œil au corps de la fonction (cf. section 1.6.2) pourrait nous apprendre que la valeur de cette expression n'est autre que le carré de son argument, soit ici la valeur $4x^2$.

[12] Cette double possibilité est commentée à la section 6.3.4.

Les contraintes supportées par les arguments effectifs ne sont pas les mêmes dans le C ANSI et dans le C original (ceci est certainement la plus grande différence entre les deux versions du langage) :

A. En C ANSI, si la fonction a été définie ou déclarée avec prototype (cf. section 4.1) :
– le nombre des arguments effectifs doit correspondre à celui des arguments formels[13] ;
– chaque argument effectif doit être compatible, au sens de l'affectation, avec l'argument formel correspondant ;
– la valeur de chaque argument effectif subit éventuellement les mêmes conversions qu'elle subirait dans l'affectation

argument formel `=` *argument effectif*

B. En C original, ou en C ANSI si la fonction n'a pas été définie ou déclarée avec prototype :
– aucune contrainte (de nombre ou de type) n'est imposée aux arguments effectifs ;
– tout argument effectif de type `char` ou `short` est converti en `int` ;
– tout argument effectif de type `float` est converti en `double` ;
– les autres arguments effectifs ne subissent aucune conversion.

Par exemple, avec la fonction `carre` définie à la section 1.6.2, l'appel

```
y = carre(2);
```

est erroné en C original (la fonction reçoit la représentation interne de la valeur entière 2 en « croyant » que c'est la représentation interne d'un `double`) mais il est correct en C ANSI (l'entier 2 est converti en `double` au moment de l'appel).

Toutes ces questions sont reprises plus en détail à la section 4.

Remarque 1. Bien noter que les parenthèses doivent apparaître, même lorsque la liste des arguments est vide. Leur absence ne provoque pas d'erreur, mais change complètement le sens de l'expression. Cela constitue un piège assez vicieux tendu aux programmeurs dont la langue maternelle est Pascal.

Remarque 2. Il faut savoir que lorsqu'une fonction a été déclarée comme ayant des arguments en nombre variable (cf. section 4.3.4) les arguments correspondant à la partie variable sont traités comme les arguments des fonctions sans prototype, c'est-à-dire selon les règles de la section *B* ci-dessus.

Cette remarque est loin d'être marginale car elle concerne, excusez du peu, les deux fonctions les plus utilisées : `printf` et `scanf`. Ainsi, quand le compilateur rencontre l'instruction

`printf(`$expr_0$, $expr_1$, ... $expr_k$`);`

il vérifie que $expr_0$ est bien de type `char *`, mais les autres paramètres effectifs $expr_1$, ... $expr_k$ sont traités selon les règles de la section *B*.

2.2.2 Indexation `[]`

Définition restreinte (élément d'un tableau). Opération : accès au i^{eme} élément d'un tableau. Format :

exp_0 `[` exp_1 `]`

exp_0 doit être de type « tableau d'objets de type T », exp_1 doit être d'un type entier. Alors exp_0`[`exp_1`]` est de type T ; cette expression désigne l'élément du tableau dont l'indice est donné par la valeur de exp_1.

Deux détails auxquels on tient beaucoup en C :
– le premier élément d'un tableau a toujours l'indice 0 ;
– il n'est jamais vérifié que la valeur de l'indice dans une référence à un tableau appartient à l'intervalle $0...N-1$ déterminé par le nombre N d'éléments alloués par la déclaration du tableau. Autrement dit, il n'y a jamais de « test de débordement ».

Exemple. L'expression `t[0]` désigne le premier élément du tableau `t`, `t[1]` le second, etc.

En C, les tableaux sont toujours à un seul indice ; mais leurs composantes peuvent être à leur tour des tableaux. Par exemple, un élément d'une matrice rectangulaire sera noté :

```
m[i][j]
```

Une telle expression suppose que `m[i]` est de type « tableau d'objets de type T » et donc que `m` est de type « tableau de tableaux d'objets de type T ». C'est le cas, par exemple, si `m` a été déclarée par une expression de la forme (`NL` et `NC` sont des constantes) :

```
double m[NL][NC];
```

[13] Il existe néanmoins un moyen pour écrire des fonctions avec un nombre variable d'arguments (cf. section 4.3.4)

Définition complète (indexation au sens large). Opération : accès à un objet dont l'adresse est donnée par une adresse de base et un déplacement. Format :

$$exp_0 \texttt{ [} exp_1 \texttt{]}$$

exp_0 doit être de type « adresse d'un objet de type T », exp_1 doit être de type entier. Alors exp_0`[`exp_1`]` désigne l'objet de type T ayant pour adresse (voir la figure 2) :

$$valeur(exp_0) + valeur(exp_1) \times taille(T)$$

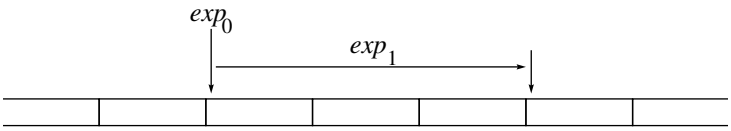

Fig. 2 – L'indexation

Il est clair que, si exp_0 est de type tableau, les deux définitions de l'indexation données coïncident. Exemple : si `t` est un tableau d'entiers et `p` un « pointeur vers entier » auquel on a affecté l'adresse de `t[0]`, alors les expressions suivantes désignent le même objet :

```
t[i]    p[i]    *(p + i)    *(t + i)
```

Dans un cas comme dans l'autre, l'expression exp_0`[`exp_1`]` est une *lvalue*, sauf si elle est de type tableau.

2.2.3 Sélection .

Opération : accès à un champ d'une structure ou d'une union. Format :

$$exp \texttt{ . } identif$$

exp doit posséder un type `struct` ou `union`, et identif doit être le nom d'un des champs de la structure ou de l'union en question. En outre, *exp* doit être une *lvalue*. Alors, *exp*`.`*identif* désigne le champ *identif* de l'objet désigne par *exp*.

Cette expression est une *lvalue*, sauf si elle est de type tableau.

Exemple. Avec la déclaration

```
struct personne {
    long int num;
    struct {
        char rue[32];
        char *ville;
    } adresse;
} fiche;
```

les expressions

```
fiche.num    fiche.adresse    fiche.adresse.rue    etc.
```

désignent les divers champs de la variable fiche. Seules les deux premières sont des *lvalue*.

2.2.4 Sélection dans un objet pointé ->

Opération : accès au champ d'une structure ou d'une union pointée. Format :

$$exp\texttt{->}identif$$

exp doit posséder le type « adresse d'une structure ou d'une union » et *identif* doit être le nom d'un des champs de la structure ou de l'union en question. Dans ces conditions, *exp*`->`*identif* désigne le champ *identif* de la structure ou de l'union dont l'adresse est indiquée par la valeur de *exp*.

Cette expression est une *lvalue*, sauf si elle est de type tableau.

Ainsi, *exp*`->`*identif* est strictement synonyme de `(*`*exp*`).`*identif* (remarquez que les parenthèses sont indispensables).

Par exemple, avec la déclaration :


```
struct noeud {
    int info;
    struct noeud *fils, *frere;
} *ptr;
```

les expressions suivantes sont correctes :

```
ptr->info    ptr->fils->frere    ptr->fils->frere->frere
```

2.2.5 Négation !

Opération : négation logique. Format :

$$!\,exp$$

Aucune contrainte. Cette expression désigne une valeur de l'ensemble $\{0, 1\}$, définie par :

$$!\,exp \equiv \begin{cases} 1, \text{ si } exp = 0 \\ 0, \text{ si } exp \neq 0 \end{cases}$$

Cette expression n'est pas une *lvalue*.

REMARQUE. Bien que cela ne soit pas exigé par le langage C, on évitera de « nier » (et plus généralement de comparer à zéro) des expressions d'un type flottant (`float`, `double`). A cause de l'imprécision inhérente à la plupart des calculs avec de tels nombres, l'égalité à zéro d'un flottant n'est souvent que le fruit du hasard.

2.2.6 Complément à 1 ~

Opération : négation bit à bit. Format :

$$\sim exp$$

exp doit être d'un type entier. Cette expression désigne l'objet de même type que *exp* qui a pour codage interne la configuration de bits obtenue en inversant chaque bit du codage interne de la valeur de *exp* : 1 devient 0, 0 devient 1.

Cette expression n'est pas une *lvalue*.

REMARQUE. Le complément à un n'est pas une opération abstraite (cf. section 2.3.3). La portabilité d'un programme où cet opérateur figure n'est donc pas assurée.

2.2.7 Les célèbres ++ et --

Il existe deux opérateurs unaires `++` différents : l'un est postfixé (écrit derrière l'opérande), l'autre préfixé (écrit devant).

1. Opération : post-incrémentation. Format :

$$exp\texttt{++}$$

exp doit être de type numérique (entier ou flottant) ou pointeur. Ce doit être une *lvalue*. Cette expression est caractérisée par :
- un type : celui de *exp* ;
- une valeur : la même que *exp* avant l'évaluation de *exp*`++` ;
- un effet de bord : le même que celui de l'affectation *exp* `=` *exp* `+ 1`.

2. Opération : pré-incrémentation. Format :

$$\texttt{++}exp$$

exp doit être de type numérique (entier ou flottant) ou pointeur. Ce doit être une *lvalue*. Cette expression est caractérisée par :
- un type : celui de *exp* ;
- une valeur : la même que *exp* après l'évaluation de *exp*`++` ;
- un effet de bord : le même que celui de l'affectation *exp* `=` *exp* `+ 1`.

Les expressions *exp*`++` et `++`*exp* ne sont pas des *lvalue*.

EXEMPLE.

L'affectation	équivaut à
`y = x++;`	`y = x; x = x + 1;`
`y = ++x;`	`x = x + 1; y = x;`

L'opérateur `++` bénéficie de l'arithmétique des adresses au même titre que `+`. Ainsi, si *exp* est de type « pointeur vers un objet de type T », la quantité effectivement ajoutée à *exp* par l'expression *exp*`++` dépend de la taille de T.

Il existe de même deux opérateurs unaires `--` donnant lieu à des expressions *exp*`--` et `--`*exp*. L'explication est la même, en remplaçant $+1$ par -1.

APPLICATION : réalisation d'une pile. Il est très agréable de constater que les opérateurs `++` et `--` et la manière d'indexer les tableaux en C se combinent harmonieusement et permettent par exemple la réalisation simple et efficace de piles par des tableaux, selon le schéma suivant :

– déclaration et initialisation d'une pile (`OBJET` est un type prédéfini, dépendant du problème particulier considéré ; `MAXPILE` est une constante représentant un majorant du nombre d'éléments dans la pile) :

```
OBJET espace[MAXPILE];
int nombreElements = 0;
```

– opération « empiler la valeur de `x` » :

```
if (nombreElements >= MAXPILE)
    erreur("tentative d'empilement dans une pile pleine");
espace[nombreElements++] = x;
```

– opération « dépiler une valeur et la ranger dans `x` » :

```
if (nombreElements <= 0)
    erreur("tentative de depilement d'une pile vide");
x = espace[--nombreElements];
```

On notera que, si on procède comme indiqué ci-dessus, la variable `nombreElements` possède constamment la valeur que son nom suggère : le nombre d'éléments effectivement présents dans la pile.

2.2.8 Moins unaire -

Opération : changement de signe. Format :

$$-exp$$

exp doit être une expression numérique (entière ou réelle). Cette expression représente l'objet de même type que *exp* dont la valeur est l'opposée de celle de *exp*. Ce n'est pas une *lvalue*.

2.2.9 Indirection *

Opération : accès à un objet pointé. On dit aussi « déréference ». Format :

$$*exp$$

exp doit être une expression de type « adresse d'un objet de type T ». **exp* représente alors l'objet de type T ayant pour adresse la valeur de *exp*.

L'expression **exp* est une *lvalue*.

REMARQUE 1. On prendra garde au fait qu'il existe un bon nombre d'opérateurs ayant une priorité supérieure à celle de `*`, ce qui oblige souvent à utiliser des parenthèses. Ainsi par exemple les expressions Pascal `e↑[i]` et `e[i]↑` doivent respectivement s'écrire, en C, `(*e)[i]` et `*(e[i])`, la deuxième pouvant aussi s'écrire `*e[i]`.

REMARQUE 2. L'expression `*p` signifie *accès à la mémoire dont* `p` *contient l'adresse*. C'est une opération « sans filet ». Quel que soit le contenu de `p`, la machine pourra toujours le considérer comme une adresse et accéder à la mémoire correspondante. Il appartient au programmeur de prouver que cete mémoire a été effectivement allouée au programme en question. Si c'est le pas on dit que la valeur de `p` est une adresse valide ; dans le cas contraire, les pires erreurs sont à craindre à plus ou moins court terme.

2.2.10 Obtention de l'adresse `&`

Opération : obtention de l'adresse d'un objet occupant un emplacement de la mémoire. Format :

`&`*exp*

exp doit être une expression d'un type quelconque *T*. Ce doit être une *lvalue*.

L'expression `&`*exp* a pour type « adresse d'un objet de type *T* » et pour valeur l'adresse de l'objet représenté par *exp*. L'expression `&`*exp* n'est pas une *lvalue*.

Ainsi, si `i` est une variable de type `int` et `p` une variable de type « pointeur vers un `int` », alors à la suite de l'instruction

```
p = &i;
```

`i` et `*p` désignent le même objet.

Exemple 1. Une utilisation fréquente de cet opérateur est l'obtention de l'adresse d'une variable en vue de la passer à une fonction pour qu'elle modifie la variable :

```
scanf("%d%lf%s", &i, &t[j], &p->nom);
```

Exemple 2. Une autre utilisation élégante de cet opérateur est la création de composantes « fixes » dans les structures chaînées. Le programme suivant déclare une liste chaînée circulaire représentée par le pointeur `entree` et réduite pour commencer à un unique maillon qui est son propre successeur (voir figure 3) ; d'autres maillons seront créés dynamiquement durant l'exécution :

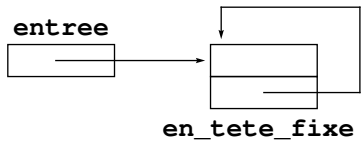

Fig. 3 – Maillon fixe en tête d'une liste chaînée

```
struct en_tete {
    long taille;
    struct en_tete *suivant;
} en_tete_fixe = { 0, &en_tete_fixe };

struct en_tete *entree = &en_tete_fixe;
```

Remarque. Une expression réduite à un nom de tableau, à un nom de fonction ou à une constante chaîne de caractères est considérée comme une constante de type adresse ; l'opérateur `&` appliqué à de telles expressions est donc sans objet. Un tel emploi de `&` devrait être considéré comme erroné, mais beaucoup de compilateurs se contentent de l'ignorer.

2.2.11 Opérateur `sizeof`

Opération : calcul de la taille correspondant à un type. Première forme :

`sizeof (` *descripteur-de-type* `)`

Cette expression représente un nombre entier qui exprime la taille qu'occuperait en mémoire un objet possédant le type indiqué (les descripteurs de types sont expliqués à la section 5.4).

Deuxième forme :

`sizeof` *exp*

exp est une expression quelconque (qui n'est pas évaluée par cette évaluation de `sizeof`). L'expression `sizeof` *exp* représente la taille qu'occuperait en mémoire un objet possédant le même type que *exp*.

Dans un cas comme dans l'autre `sizeof`... est une expression constante. En particulier, ce n'est pas une *lvalue*.

La taille des objets est exprimée en nombre d'octets. Plus exactement, l'unité choisie est telle que la valeur de `sizeof(char)` soit 1 (on peut aussi voir cela comme une définition du type `char`).

REMARQUE 1. Dans le C original, le type de l'expression `sizeof`... est `int`. Dans le C ANSI, ce type peut changer d'un système à un autre (`int`, `long`, `unsigned`, etc.). Pour cette raison il est défini, sous l'appellation `size_t`, dans le fichier `stddef.h`. Il est recommandé de déclarer de type `size_t` toute variable (resp. toute fonction) devant contenir (resp. devant renvoyer) des nombres qui représentent des tailles.

REMARQUE 2. Lorsque son opérande est un tableau (ou une expression de type tableau) la valeur rendue par `sizeof` est l'encombrement *effectif* du tableau. Par exemple, avec la déclaration

```
char tampon[80];
```

l'expression `sizeof tampon` vaut 80, même si cela paraît en contradiction avec le fait que tampon peut être vu comme de type « adresse d'un `char` ». Il en découle une propriété bien utile : quel que soit le type du tableau `t`, la formule

```
sizeof t / sizeof t[0]
```

exprime toujours le *nombre d'éléments* de `t`.

2.2.12 Conversion de type ("cast" operator)

Opération : conversion du type d'une expression. Format :

$$\texttt{(}\ type_2\ \texttt{)}\ exp$$

$type_2$ représente un descripteur de type ; *exp* est une expression quelconque. L'expression ci-dessus désigne un élément de type $type_2$ qui est le résultat de la conversion vers ce type de l'élément représenté par *exp*.

L'expression $\texttt{(}type_2\texttt{)}\,exp$ n'est pas une *lvalue*.

Les conversions légitimes (et utiles) sont[14] :
- Entier vers un entier plus long (ex. `char` → `int`). Le codage de *exp* est étendu de telle manière que la valeur représentée soit inchangée.
- Entier vers un entier plus court (ex. `long` → `short`). Si la valeur de *exp* est assez petite pour être représentable dans le type de destination, sa valeur est la même après conversion. Sinon, la valeur de *exp* est purement et simplement tronquée (une telle troncation, sans signification abstraite, est rarement utile).
- Entier signé vers entier non signé, ou le contraire. C'est une *conversion sans travail* : le compilateur se borne à interpréter autrement la valeur de *exp*, sans effectuer aucune transformation de son codage interne.
- Flottant vers entier : la partie fractionnaire de la valeur de *exp* est supprimée. Par exemple, le flottant `3.14` devient l'entier `3`. Attention, on peut voir cette conversion comme une réalisation de la fonction mathématique *partie entière*, mais uniquement pour les nombres positifs : la conversion en entier de `-3.14` donne `-3`, non `-4`.
- Entier vers flottant. Sauf cas de débordement (le résultat est alors imprévisible), le flottant obtenu est celui qui approche le mieux l'entier initial. Par exemple, l'entier `123` devient le flottat `123.0`.
- Adresse d'un objet de type T_1 vers adresse d'un objet de type T_2. C'est une conversion sans travail : le compilateur donne une autre interprétation de la valeur de *exp*, sans effectuer aucune transformation de son codage interne.
  Danger ! Une telle conversion est entièrement placée sous la responsabilité du programmeur, le compilateur l'accepte toujours.
- Entier vers adresse d'un objet de type T. C'est encore une conversion sans travail : la valeur de *exp* est interprétée comme un pointeur, sans transformation de son codage interne.
  Danger ! Une telle conversion est entièrement placée sous la responsabilité du programmeur. De plus, si la représentation interne de *exp* n'a pas la taille voulue pour un pointeur, le résultat est imprévisible.

Toutes les conversions où l'un des types en présence, ou les deux, sont des types `struct` ou `union` sont interdites.

NOTE 1. Si $type_2$ et le type de *exp* sont numériques, alors la conversion effectuée à l'occasion de l'évaluation de l'expression $\texttt{(}\ type_2\ \texttt{)}\ exp$ est la même que celle qui est faite lors d'une affectation

```
x = expr;
```

où `x` représente une variable de type $type_2$.

NOTE 2. En toute rigueur, le fait que l'expression donnée par un opérateur de conversion de type ne soit pas une *lvalue* interdit des expressions qui auraient pourtant été pratiques, comme

```
((int) x)++;          /* DANGER ! */
```

[14] Attention, il y a quelque chose de trompeur dans la phrase « conversion de *exp* ». N'oubliez pas que, contrairement à ce que suggère une certaine manière de parler, l'évaluation de l'expression $\texttt{(}type\texttt{)}\,exp$ ne change ni le type ni la valeur de *exp*.

Cependant, certains compilateurs acceptent cette expression, la traitant comme

```
x = (le type de x)((int) x + 1) ;
```

EXEMPLE 1. Si `i` et `j` sont deux variables entières, l'expression `i / j` représente leur division entière (ou *euclidienne*, ou encore leur quotient par défaut). Voici deux manières de ranger dans `x` (une variable de type `float`) leur quotient décimal :

```
x = (float) i / j;                          x = i / (float) j;
```

Et voici deux manières de se tromper (en n'obtenant que le résultat de la conversion vers le type `float` de leur division entière)

```
x = i / j;    /* ERREUR */                  x = (float)(i / j);    /* ERREUR */
```

EXEMPLE 2. Une utilisation pointue et dangereuse, mais parfois nécessaire, de l'opérateur de conversion de type entre types pointeurs consiste à s'en servir pour « voir » un espace mémoire donné par son adresse comme possédant une certaine structure alors qu'en fait il n'a pas été ainsi déclaré :

– déclaration d'une structure :

```
struct en_tete {
    long taille;
    struct en_tete *suivant;
};
```

– déclaration d'un pointeur « générique » :

```
void *ptr;
```

– imaginez que –pour des raisons non détaillées ici– `ptr` possède à un endroit donné une valeur qu'il est légitime de considérer comme l'adresse d'un objet de type `struct en_tete` (alors que `ptr` n'a pas ce type-là). Voici un exemple de manipulation cet objet :

```
((struct en_tete *) ptr)->taille = n;
```

Bien entendu, une telle conversion de type est faite sous la responsabilité du programmeur, seul capable de garantir qu'à tel moment de l'exécution du programme `ptr` pointe bien un objet de type `struct en_tete`.

N.B. L'APPEL D'UNE FONCTION BIEN ÉCRITE NE REQUIERT JAMAIS UN « CAST ». L'opérateur de changement de type est parfois nécessaire, mais son utilisation diminue *toujours* la qualité du programme qui l'emploie, pour une raison facile à comprendre : *cet opérateur fait taire le compilateur*. En effet, si *expr* est d'un type pointeur et $type_2$ est un autre type pointeur, l'expression `(`$type_2$`)`*expr* est toujours acceptée par le compilateur sans le moindre avertissement. C'est donc une manière de cacher des erreurs sans les résoudre.

Exemple typique : si on a oublié de mettre en tête du programme la directive `#include <stdlib.h>`, l'utilisation de la fonction `malloc` de la bibliothèque standard soulève des critiques :

```
...
MACHIN *p;
...
p = malloc(sizeof(MACHIN));
...
```

A la compilation on a des avertissements. Par exemple (avec `gcc`) :

```
monProgramme.c: In function ‘main’
monProgramme.c:9:warning: assignment makes pointer from integer without a cast
```

On croit résoudre le problème en utilisant l'opérateur de changement de type

```
...
p = (MACHIN *) malloc(sizeof(MACHIN));    /* CECI NE REGLE RIEN! */
...
```

et il est vrai que la compilation a lieu maintenant en silence, mais le problème n'est pas résolu, il est seulement caché. Sur un ordinateur où les entiers et les pointeurs ont la même taille cela peut marcher, mais ailleurs la valeur rendue par `malloc` sera endommagée lors de son affectation à `p`. A ce sujet, voyez la remarque de la page 65.

Il suffit pourtant de *bien* lire l'avertissement affiché par le compilateur pour trouver la solution. L'affectation `p = malloc(...)` lui fait dire qu'on fabrique un pointeur (`p`) à partir d'un entier (le résultat de `malloc`) sans opérateur *cast*. *Ce qui est anormal n'est pas l'absence de cast, mais le fait que le résultat de* `malloc` *soit tenu pour un entier*[15], et il n'y a aucun moyen de rendre cette affectation juste aussi longtemps que le compilateur

[15] Rappelez-vous que toute fonction appelée et non déclarée est supposée de type `int`.

fera une telle hypothèse fausse. La solution est donc d'informer ce dernier à propos du type de `malloc`, en faisant précéder l'affectation litigieuse soit d'une déclaration de cette fonction

```
void *malloc (size_t);
```

soit, c'est mieux, d'une directive ad hoc :

```
#include <stdlib.h>
...
p = malloc(sizeof(MACHIN));
...
```

2.2.13 Opérateurs arithmétiques

Ce sont les opérations arithmétiques classiques : addition, soustraction, multiplication, division et modulo (reste de la division entière). Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{+} \\ \texttt{-} \\ \texttt{*} \\ \texttt{/} \\ \texttt{\%} \end{array} \right\} exp_2$$

Aucune de ces expressions n'est une *lvalue*.

Avant l'évaluation de l'expression, les opérandes subissent les conversions « usuelles » (cf. section 2.3.1). Le langage C supporte l'arithmétique des adresses (cf. section 6.2.1).

A PROPOS DE LA DIVISION. En C on note par le même signe la division entière, qui prend deux opérandes entiers (`short`, `int`, `long`, etc.) et donne un résultat entier, et la division flottante dans laquelle le résultat est flottant (`float`, `double`). D'autre part, les règles qui commandent les types des opérandes et du résultat des expressions arithmétiques (cf. section 2.3.1), et notamment la règle dite « du plus fort », ont la conséquence importante suivante : dans l'expression

$$expr_1 \ / \ expr_2$$

- si $expr_1$ et $expr_2$ sont toutes deux entières alors « / » est traduit par l'opération « division entière »[16], et le résultat est entier,
- si au moins une des expressions $expr_1$ ou $expr_2$ n'est pas entière, alors l'opération faite est la division flottante des valeurs de $expr_1$ et $expr_2$ toutes deux converties dans le type `double`. Le résultat est une valeur `double` qui approche le rationnel $\frac{expr_1}{expr_2}$ du mieux que le permet la la précision du type `double`.

Il résulte de cette règle un piège auquel il faut faire attention : `1/2` ne vaut pas 0.5 mais 0. De même, dans

```
int somme, nombre;
float moyenne;
...
moyenne = somme / 100;
```

la valeur de `moyenne` n'est pas ce que ce nom suggère, car `somme` et `100` sont tous deux entiers et l'expression `somme / 100` est entière, donc tronquée (et il ne faut pas croire que l'affectation ultérieure à la variable flottante `moyenne` pourra retrouver les décimales perdues). Dans ce cas particulier, la solution est simple :

```
moyenne = somme / 100.0;
```

Même problème si le dénominateur avait été une variable entière, comme dans

```
moyenne = somme / nombre;
```

ici la solution est un peu plus compliquée :

```
moyenne = somme / (double) nombre;
```

[16] On prendra garde au fait que si les opérandes ne sont pas tous deux positifs la division entière du langage C ne coincide pas avec le « quotient par défaut » (quotient de la « division euclidienne » des matheux).

Ici, si q est le quotient de a par b alors $|q|$ est le quotient de $|a|$ par $|b|$. Par exemple, la valeur de $(-17)/5$ ou de $17/(-5)$ est -3 alors que le quotient par défaut de -17 par 5 est plutôt -4 ($-17 = 5 \times (-4) + 3$).

2.2.14 Décalages << >>

Opération : décalages de bits. Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{<<} \\ \texttt{>>} \end{array} \right\} exp_2$$

exp_1 et exp_2 doivent être d'un type entier (`char`, `short`, `long`, `int`...). L'expression exp_1 `<<` exp_2 (resp. exp_1 `>>` exp_2) représente l'objet de même type que exp_1 dont la représentation interne est obtenue en décalant les bits de la représentation interne de exp_1 vers la gauche[17] (resp. vers la droite) d'un nombre de positions égal à la valeur de exp_2. Autrement dit,

$$exp_1 \texttt{ << } exp_2$$

est la même chose (si `<< 1` apparaît exp_2 fois) que

$$\texttt{((}exp_1\texttt{ << 1) << 1) ... << 1}$$

Remarque analogue pour le décalage à droite `>>`.

Les bits sortants sont perdus. Les bits entrants sont :
- dans le cas du décalage à gauche, des zéros ;
- dans le cas du décalage à droite : si exp_1 est d'un type non signé, des zéros ; si exp_1 est d'un type signé, des copies du bit de signe[18].

Par exemple, si on suppose que la valeur de exp_1 est codée sur huit bits, notés

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

(chaque b_i vaut 0 ou 1), alors

exp_1 `<< 1`	$\equiv$	$b_6 b_5 b_4 b_3 b_2 b_1 b_0$ `0`	
exp_1 `>> 1`	$\equiv$	`0`$b_7 b_6 b_5 b_4 b_3 b_2 b_1$	(cas non signé)
exp_1 `>> 1`	$\equiv$	$b_7 b_7 b_6 b_5 b_4 b_3 b_2 b_1$	(cas signé)

Avant l'évaluation du résultat, les opérandes subissent les conversions usuelles (cf. section 2.3.1). Ces expressions ne sont pas des *lvalue*.

Remarque. Les opérateurs de décalage de bits ne sont pas des opérations abstraites (cf. section 2.3.3). La portabilité d'un programme où ils figurent n'est donc pas assurée.

2.2.15 Comparaisons == != < <= > >=

Il s'agit des comparaisons usuelles : égal, différent, inférieur, inférieur ou égal, supérieur, supérieur ou égal. Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{==} \\ \texttt{!=} \\ \texttt{<} \\ \texttt{<=} \\ \texttt{>} \\ \texttt{>=} \end{array} \right\} exp_2$$

exp_1 et exp_2 doivent être d'un type simple (nombre ou pointeur). Cette expression représente l'un des éléments (de type `int`) 1 ou 0 selon que la relation indiquée est ou non vérifiée par les valeurs de exp_1 et exp_2.

Avant la prise en compte de l'opérateur, les opérandes subissent les conversions usuelles (cf. section 2.3.1). Ces expressions ne sont pas des *lvalue*.

Notez que, contrairement à ce qui se passe en Pascal, ces opérateurs ont une priorité supérieure à celle des connecteurs logiques, ce qui évite beaucoup de parenthèses disgracieuses. Ainsi, en C, l'expression

```
0 <= x && x < 10
```

est correctement écrite.

A propos de « être vrai » et « être non nul ». Comme on a dit (cf. section 1.4.1), le type booléen

[17] Par convention la droite d'un nombre codé en binaire est le côté des bits de poids faible (les unités) tandis que la gauche est celui des bits de poids forts (correspondant aux puissances 2^k avec k grand).

[18] De cette manière le décalage à gauche correspond (sauf débordement) à la multiplication par 2^{exp_2}, tandis que le décalage à droite correspond à la division entière par 2^{exp_2}, aussi bien si exp_1 est signée que si elle est non signée

n'existe pas en C. N'importe quelle expression peut occuper la place d'une condition ; elle sera tenue pour fausse si elle est nulle, pour vraie dans tous les autres cas. Une conséquence de ce fait est la suivante : à la place de

```
if (i != 0) etc.
while (*ptchar != '\0') etc.
for (p = liste; p != NULL; p = p->suiv) etc.
```

on peut écrire respectivement

```
if (i) etc.
while (*ptchar) etc.
for (p = liste; p; p = p->suiv) etc.
```

On admet généralement qu'à cause de propriétés techniques des processeurs, ces expressions raccourcies représentent une certaine optimisation des programmes. C'est pourquoi les premiers programmeurs C, qui ne disposaient que de compilateurs simples, ont pris l'habitude de les utiliser largement. On ne peut plus aujourd'hui conseiller cette pratique. En effet, les compilateurs actuels sont suffisamment perfectionnés pour effectuer spontanément cette sorte d'optimisations et il n'y a plus aucune justification de l'emploi de ces comparaisons implicites qui rendent les programmes bien moins expressifs.

ATTENTION. La relation d'égalité se note `==`, non `=`. Voici une faute possible chez les nouveaux venus à C en provenance de Pascal qui, pour traduire le bout de Pascal `if a = 0 then` *etc.*, écrivent

```
if (a = 0) etc.
```

Du point de vue syntaxique cette construction est correcte, mais elle est loin d'avoir l'effet escompté. En tant qu'expression, `a = 0` vaut `0` (avec, comme effet de bord, la mise à zéro de `a`). Bien sûr, il fallait écrire

```
if (a == 0) etc.
```

2.2.16 Opérateurs de bits `&` | `^`

Ces opérateurs désignent les opérations logiques *et*, *ou* et *ou exclusif* sur les bits des représentations internes des valeurs des opérandes. Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{\&} \\ \texttt{|} \\ \texttt{\^{}} \end{array} \right\} exp_2$$

exp_1 et exp_2 doivent être d'un type entier. Cette expression représente un objet de type entier dont le codage interne est construit bit par bit à partir des bits correspondants des valeurs de exp_1 et exp_2, selon la table suivante, dans laquelle $(exp)_i$ signifie « le i^{eme} bit de exp » :

$(exp_1)_i$	$(exp_2)_i$	$(exp_1$ `&` $exp_2)_i$	$(exp_1$ `^` $exp_2)_i$	$(exp_1$ `|` $exp_2)_i$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Avant l'évaluation du résultat, les opérandes subissent les conversions usuelles (cf. section 2.3.1). Ces expressions ne sont pas des *lvalue*.

EXEMPLE. Il n'y a pas beaucoup d'exemples « de haut niveau »[19] d'utilisation de ces opérateurs. Le plus utile est sans doute la réalisation d'ensembles de nombres naturels inférieurs à une certaine constante pas trop grande, c'est-à-dire des sous-ensembles de l'intervalle d'entiers [0 ... $N-1$], N valant 8, 16 ou 32.

Par exemple, les bibliothèques graphiques comportent souvent une fonction qui effectue l'affichage d'un texte affecté d'un *ensemble d'attributs* (gras, italique, souligné, capitalisé, etc.). Cela ressemble à ceci :

```
void afficher(char *texte, unsigned long attributs);
```

Pour faire en sorte que l'argument `attributs` puisse représenter un ensemble d'attributs quelconque, on associe les attributs à des entiers conventionnels :

[19] En effet, ces opérateurs sont principalement destinés à l'écriture de logiciel de bas niveau, comme les pilotes de périphérique (les célèbres « drivers »), c'est-à-dire des programmes qui reçoivent les informations transmises par les composants matériels ou qui commandent le fonctionnement de ces derniers. Dans de telles applications on est souvent aux pries avec des nombres dont chaque bit a une signification propre, indépendante des autres. Ces programmes sortent du cadre de ce cours.

```
#define GRAS         1        /* en binaire: 00...000001 */
#define ITALIQUE     2        /* en binaire: 00...000010 */
#define SOULIGNE     4        /* en binaire: 00...000100 */
#define CAPITALISE   8        /* en binaire: 00...001000 */
```
etc.

Les valeurs utilisées pour représenter les attributs sont des puissances de 2 distinctes, c'est-à-dire des nombres qui, écrits en binaire, comportent un seul 1 placé différemment de l'un à l'autre.

L'utilisateur d'une telle fonction emploie l'opérateur | pour composer, lors de l'appel, l'ensemble d'attributs qu'il souhaite :

```
afficher("Bonjour", GRAS | SOULIGNE);    /* affichage en gras et italique */
```

(avec les constantes de notre exemple, la valeur de l'expression `GRAS | SOULIGNE` est un nombre qui, en binaire, s'écrit `00...000101`).

De son côté, le concepteur de la fonction utilise l'opérateur `&` pour savoir si un attribut appartient ou non à l'ensemble donné :

```
...
if ((attributs & GRAS) != 0)
    prendre les dispositions nécessaires pour afficher en gras
else if ((attributs & ITALIQUE) != 0)
    prendre les dispositions nécessaires pour afficher en italique
...
```

2.2.17 Connecteurs logiques `&&` et `||`

Opérations : conjonction et disjonction. Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{\&\&} \\ \texttt{||} \end{array} \right\} exp_2$$

exp_1 et exp_2 sont deux expressions de n'importe quel type. Cette expression représente un élément de type `int` parmi { 0, 1 } défini de la manière suivante :

Pour évaluer exp_1 `&&` exp_2 : exp_1 est évaluée d'abord et
– si la valeur de exp_1 est nulle, exp_2 n'est pas évaluée et exp_1 `&&` exp_2 vaut 0 ;
– sinon exp_2 est évaluée et exp_1 `&&` exp_2 vaut 0 ou 1 selon que la valeur de exp_2 est nulle ou non.

Pour évaluer exp_1 `||` exp_2 : exp_1 est évaluée d'abord et
– si la valeur de exp_1 est non nulle, exp_2 n'est pas évaluée et exp_1 `||` exp_2 vaut 1 ;
– sinon exp_2 est évaluée et exp_1 `||` exp_2 vaut 0 ou 1 selon que la valeur de exp_2 est nulle ou non.

Ces expressions ne sont pas des *lvalue*.

Applications. Ainsi, C garantit que le premier opérande sera évalué d'abord et que, s'il suffit à déterminer le résultat de la conjonction ou de la disjonction, alors le second opérande ne sera même pas évalué. Pour le programmeur, cela est une bonne nouvelle. En effet, il n'est pas rare qu'on écrive des conjonctions dont le premier opérande « protège » (dans l'esprit du programmeur) le second ; si cette protection ne fait pas partie de la sémantique de l'opérateur pour le langage utilisé, le programme résultant peut être faux. Considérons l'exemple suivant : un certain tableau `table` est formé de `nombre` chaînes de caractères ; soit `ch` une variable chaîne. L'opération « recherche de `ch` dans `table` » peut s'écrire :

```
...
i = 0;
while (i < nombre && strcmp(table[i], ch) != 0)
    i++;
...
```

Ce programme est juste parce que la condition `strcmp(table[i], ch) != 0` n'est évaluée qu'après avoir vérifié que la condition `i < nombre` est vraie (un appel de `strcmp(table[i], ch)` avec un premier argument `table[i]` invalide peut avoir des conséquences tout à fait désastreuses).

Une autre conséquence de cette manière de concevoir `&&` et `||` est stylistique. En C la conjonction et la disjonction s'évaluent dans l'esprit des instructions plus que dans celui des opérations. L'application pratique de cela est la possibilité d'écrire sous une forme fonctionnelle des algorithmes qui dans d'autres langages seraient

séquentiels. Voici un exemple : le prédicat qui caractérise la présence d'un élément dans une liste chaînée. Version (récursive) habituelle :

```
int present(INFO x, LISTE L) {   /* l'information x est-elle dans la liste L ? */
    if (L == NULL)
        return 0;
    else if (L->info == x)
        return 1;
    else
        return present(x, L->suivant);
}
```

Version dans un style fonctionnel, permise par la sémantique des opérateurs `&&` et `||` :

```
int existe(INFO x, LISTE L) {   /* l'information x est-elle dans la liste L ? */
    return L != NULL && (x == L->info || existe(x, L->suivant));
}
```

2.2.18 Expression conditionnelle `? :`

Opération : sorte de *if...then...else...* présenté sous forme d'expression, c'est-à-dire renvoyant une valeur. Format :

$$exp_0 \texttt{ ? } exp_1 \texttt{ : } exp_2$$

exp_0 est d'un type quelconque. exp_1 et exp_2 doivent être de types compatibles. Cette expression est évaluée de la manière suivante :

La condition exp_0 est évaluée d'abord
- si sa valeur est non nulle, exp_1 est évaluée et définit la valeur de l'expression conditionnelle. Dans ce cas, exp_2 n'est pas évaluée ;
- sinon, exp_2 est évaluée et définit la valeur de l'expression conditionnelle. Dans ce cas, exp_1 n'est pas évaluée.

L'expression exp_0 `?`exp_1`:`exp_2 n'est pas une *lvalue*.

EXEMPLE. L'opérateur conditionnel n'est pas forcément plus facile à lire que l'instruction conditionnelle, mais permet quelquefois de réels allégements du code. Imaginons un programme devant afficher un des textes *non* ou *oui* selon que la valeur d'une variable `reponse` est nulle ou non. Solutions classiques de ce micro-problème :

```
if (reponse)
    printf("la réponse est oui");
else
    printf("la réponse est non");
```

ou bien, avec une variable auxiliaire :

```
char *texte;
...
if (reponse)
    texte = "oui";
else
    texte = "non";
printf("la réponse est %s", texte);
```

Avec l'opérateur conditionnel c'est bien plus compact :

```
printf("la réponse est %s", reponse ? "oui" : "non");
```

2.2.19 Affectation `=`

Opération : affectation, considérée comme une expression. Format :

$$exp_1 \texttt{ = } exp_2$$

exp_1 doit être une *lvalue*. Soit $type_1$ le type de exp_1 ; l'affectation ci-dessus représente le même objet que

$$\texttt{(} type_1 \texttt{) } exp_2$$

(la valeur de exp_2 convertie dans le type de exp_1), avec pour effet de bord le rangement de cette valeur dans l'emplacement de la mémoire déterminé par exp_1.

L'expression exp_1 `=` exp_2 n'est pas une *lvalue*.

Contrairement à ce qui se passe dans d'autres langages, une affectation est donc considérée en C comme une expression : elle « fait » quelque chose, mais aussi elle « vaut » une certaine valeur et, à ce titre, elle peut figurer comme opérande dans une sur-expression. On en déduit la possibilité des affectations multiples, comme dans l'expression :

```
a = b = c = 0;
```

comprise comme `a = (b = (c = 0))`. Elle aura donc le même effet que les trois affectations `a = 0 ; b = 0 ; c = 0 ;` Autre exemple, lecture et traitement d'une suite de caractères dont la fin est indiquée par un point :

```
...
while ((c = getchar()) != '.')
    exploitation de c
...
```

Des contraintes pèsent sur les types des deux opérandes d'une affectation exp_1 `=` exp_2. Lorsqu'elles sont satisfaites on dit que exp_1 et exp_2 sont *compatibles pour l'affectation*. Essentiellement :

- deux types numériques sont toujours compatibles pour l'affectation. La valeur de exp_2 subit éventuellement une conversion avant d'être rangée dans exp_1. La manière de faire cette conversion est la même que dans le cas de l'opérateur de conversion (cf. section 2.2.12) ;
- si exp_1 et exp_2 sont de types adresses distincts, certains compilateurs (dont ceux conformes à la norme ANSI) les considéreront comme incompatibles tandis que d'autres compilateurs se limiteront à donner un message d'avertissement lors de l'affectation de exp_2 à exp_1 ;
- dans les autres cas, exp_1 et exp_2 sont compatibles pour l'affectation si et seulement si elles sont de même type.

D'autre part, de la signification du nom d'une variable de type tableau (cf. 5.1.1) et de celle d'une variable de type structure (cf. 5.2.1) on déduit que :

- on ne peut affecter un tableau à un autre, même s'ils sont définis de manière rigoureusement identique (un nom de tableau n'est pas une *lvalue*) ;
- on peut affecter le contenu d'une variable de type structure ou union à une autre, à la condition qu'elles aient été explicitement déclarées comme ayant exactement le même type.

2.2.20 Autres opérateurs d'affectation `+=` `*=` *etc.*

Opération binaire vue comme une modification du premier opérande. Format :

$$exp_1 \left\{ \begin{array}{c} \texttt{+=} \\ \texttt{-=} \\ \texttt{*=} \\ \texttt{/=} \\ \texttt{\%=} \\ \texttt{>>=} \\ \texttt{<<=} \\ \texttt{\&=} \\ \texttt{\^{}=} \\ \texttt{|=} \end{array} \right\} exp_2$$

exp_1 doit être une *lvalue*. Cela fonctionne de la manière suivante : si $\bullet$ représente l'un des opérateurs `+ - * / % >> << & ^ |`, alors

$$exp_1 \bullet = exp_2$$

peut être vue comme ayant la même valeur et le même effet que

$$exp_1 = exp_1 \bullet exp_2$$

mais *avec une seule évaluation* de exp_1. L'expression résultante n'est pas une *lvalue*.

Exemple. Écrivons la version itérative usuelle de la fonction qui calcule x^n avec x flottant et n entier non négatif :


```
double puissance(double x, int n) {
    double p = 1;
    while (n != 0) {
        if (n % 2 != 0)     /* n est-il impair ? */
            p *= x;
        x *= x;
        n /= 2;
    }
    return p;
}
```

REMARQUE. En examinant ce programme, on peut faire les mêmes commentaires qu'à l'occasion de plusieurs autres éléments du langage C :

- l'emploi de ces opérateurs constitue une certaine optimisation du programme. En langage machine, la suite d'instructions qui traduit textuellement `a += c` est plus courte que celle qui correspond à `a = b + c`. Or, un compilateur rustique traitera `a = a + c` comme un cas particulier de `a = b + c`, sans voir l'équivalence avec la première forme.
- hélas, l'emploi de ces opérateurs rend les programmes plus denses et moins faciles à lire, ce qui favorise l'apparition d'erreurs de programmation.
- de nos jours les compilateurs de C sont devenus assez perfectionnés pour déceler automatiquement la possibilité de telles optimisations. Par conséquent, l'argument de l'efficacité ne justifie plus qu'on obscurcisse un programme par l'emploi de tels opérateurs.

Il faut savoir cependant qu'il existe des situations où l'emploi de ces opérateurs n'est pas qu'une question d'efficacité. En effet, si exp_1 est une expression sans effet de bord, alors les expressions $exp_1 \bullet = exp_2$ et $exp_1 = exp_1 \bullet exp_2$ sont réellement équivalentes. Mais ce n'est plus le cas si exp_1 a un effet de bord. Il est clair, par exemple, que les deux expressions suivantes ne sont pas équivalentes (la première est tout simplement erronée)[20] :

```
nombre[rand() % 100] = nombre[rand() % 100] + 1;  /* ERRONE ! */

nombre[rand() % 100] += 1;                        /* CORRECT */
```

2.2.21 L'opérateur virgule ,

Opération : évaluation en séquence. Format :

$$exp_1 \ , \ exp_2$$

exp_1 et exp_2 sont quelconques. L'évaluation de exp_1 , exp_2 consiste en l'évaluation de exp_1 suivie de l'évaluation de exp_2. L'expression exp_1 , exp_2 possède le type et la valeur de exp_2 ; le résultat de l'évaluation de exp_1 est « oublié », mais non son éventuel effet de bord (cet opérateur n'est utile que si exp_1 a un effet de bord).

L'expression exp_1 , exp_2 n'est pas une *lvalue*.

EXEMPLE. Un cas fréquent d'utilisation de cet opérateur concerne la boucle *for* (cf. section 3.2.6), dont la syntaxe requiert exactement trois expressions à trois endroits bien précis. Parfois, certaines de ces expressions doivent être doublées :

```
...
for (pr = NULL, p = liste; p != NULL; pr = p, p = p->suivant)
    if (p->valeur == x)
        break;
...
```

REMARQUE SYNTAXIQUE. Dans des contextes où des virgules apparaissent normalement, par exemple lors d'un appel de fonction, des parenthèses sont requises afin de forcer le compilateur à reconnaître l'opérateur virgule. Par exemple, l'expression

`uneFonction(`exp_1`, (`exp_2`, `exp_3`), `exp_4`);`

représente un appel de `uneFonction` avec trois arguments effectifs : les valeurs de exp_1, exp_3 et exp_4. Au passage, l'expression exp_2 aura été évaluée. Il est certain que exp_2 aura été évaluée avant exp_3, mais les spécifications du langage ne permettent pas de placer les évaluations de exp_1 et exp_4 par rapport à celles de exp_2 et exp_3.

[20] La fonction `rand()` renvoie un entier aléatoire distinct chaque fois qu'elle est appelée.

2.3 Autres remarques

2.3.1 Les conversions usuelles

Les règles suivantes s'appliquent aux expressions construites à l'aide d'un des opérateurs `*`, `/`, `%`, `+`, `-`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&` et `||`. *Mutatis mutandis*, elles s'appliquent aussi à celles construites avec les opérateurs `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=` et `|=`.

Dans ces expressions, les opérandes subissent certaines conversions avant que l'expression ne soit évaluée. En C ANSI ces conversions, dans l' ordre « logique » où elles sont faites, sont les suivantes :

- Si un des opérandes est de type `long double`, convertir l'autre dans le type `long double` ; le type de l'expression sera `long double`.
- Sinon, si un des opérandes est de type `double`, convertir l'autre dans le type `double` ; le type de l'expression sera `double`.
- Sinon, si un des opérandes est de type `float`, convertir l'autre dans le type `float` ; le type de l'expression sera `float`[21].
- Effectuer la *promotion entière* : convertir les `char`, les `short`, les énumérations et les champs de bits en des `int`. Si l'une des valeurs ne peut pas être représentée dans le type `int`, les convertir *toutes* dans le type `unsigned int`.
- Ensuite, si un des opérandes est `unsigned long`, convertir l'autre en `unsigned long` ; le type de l'expression sera `unsigned long`.
- Sinon, si un des opérandes est `long` et l'autre `unsigned int`[22] :
  - si un `long` peut représenter toutes les valeurs `unsigned int`, alors convertir l'opérande de type `unsigned int` en `long`. Le type de l'expression sera `long` ;
  - sinon, convertir les deux opérandes en `unsigned long`. Le type de l'expression sera `unsigned long`.
- Sinon, si un des opérandes est de type `long`, convertir l'autre en `long` ; le type de l'expression sera `long`.
- Sinon, si un des opérandes est de type unsigned int, convertir l'autre en `unsigned int` ; le type de l'expression sera `unsigned int`.
- Sinon, et si l'expression est correcte, c'est que les deux opérandes sont de type `int` ; le type de l'expression sera `int`.

2.3.2 L'ordre d'évaluation des expressions

Les seules expressions pour lesquelles l'ordre (chronologique) d'évaluation des opérandes est spécifié sont les suivantes :

- « exp_1 `&&` exp_2 » et « exp_1 `||` exp_2 » : exp_1 est évaluée d'abord ; exp_2 n'est évaluée que si la valeur de exp_1 ne permet pas de conclure ;
- « exp_0 `?` exp_1 `:` exp_2 » exp_0 est évaluée d'abord. Une seule des expressions exp_1 ou exp_2 est évaluée ensuite ;
- « exp_0 `,` exp_0 » : exp_1 est évaluée d'abord, exp_2 est évaluée ensuite.

Dans tous les autres cas, C ne garantit pas l'ordre chronologique dans lequel les opérandes intervenant dans une expression ou les arguments effectifs d'un appel de fonction sont évalués ; il ne faut donc pas faire d'hypothèse à ce sujet. La question ne se pose que lorsque ces opérandes et arguments sont à leur tour des expressions complexes ; elle est importante dans la mesure où C favorise la programmation avec des effets de bord. Par exemple, si `i` vaut 1, l'expression `a[i] + b[i++]` peut aussi bien additionner `a[1]` et `b[1]` que `a[2]` et `b[1]`.

L'ordre d'évaluation des opérandes d'une affectation n'est pas fixé non plus. Pour évaluer exp_1 `=` exp_2, on peut évaluer d'abord l'adresse de exp_1 et ensuite la valeur de exp_2, ou bien faire l'inverse. Ainsi, le résultat de l'affectation `a[i] = b[i++]` est imprévisible.

2.3.3 Les opérations non abstraites

Beaucoup d'opérateurs étudiés dans cette section (opérateurs arithmétiques, comparaisons, logiques, etc.) représentent des opérations *abstraites*, c'est-à-dire possédant une définition formelle qui ne fait pas intervenir les particularités de l'implantation du langage. Bien sûr, les opérandes sont représentés dans la machine par des

[21] Cette règle est apparue avec le C ANSI : le compilateur accepte de faire des calculs sur des flottants en simple précision. Dans le C original, elle s'énonce plus simplement : « sinon, si l'un des opérandes est de type `float`, convertir les deux opérandes dans le type `double` ; le type de l'expression sera `double` ».

[22] Cette règle compliquée est apparue avec le C ANSI. En C original, le type unsigned « tire vers lui » les autres types.

configurations de bits, mais seule leur interprétation comme des entités de niveau supérieur (nombres entiers, flottants...) est utile pour définir l'effet de l'opération en question ou les contraintes qu'elle subit.

A l'opposé, un petit nombre d'opérateurs, le complément à un (`~`), les décalages (`<<` et `>>`) et les opérations bit-à-bit (`&`, `^` et `|`), n'ont pas forcément de signification abstraite. Les transformations qu'ils effectuent sont définies au niveau des bits constituant le codage des opérandes, non au niveau des nombres que ces opérandes représentent. De telles opérations sont réservées aux programmes qui remplissent des fonctions de très bas niveau, c'est-à-dire qui sont aux points de contact entre la composante logicielle et la composante matérielle d'un système informatique.

L'aspect de cette question qui nous intéresse le plus ici est celui-ci : *la portabilité des programmes contenant des opérations non abstraites n'est pas assurée.* Ce défaut, qui n'est pas rédhibitoire dans l'écriture de fonctions de bas niveau (ces fonctions ne sont pas destinées à être portées), doit rendre le programmeur très précautionneux dès qu'il s'agit d'utiliser ces opérateurs dans des programmes de niveau supérieur, et le pousser à :

- isoler les opérations non abstraites dans des fonctions de petite taille bien repérées ;
- documenter soigneusement ces fonctions ;
- constituer des jeux de tests validant chacune de ces fonctions.

3 Instructions

3.1 Syntaxe

Dans les descriptions syntaxiques suivantes le suffixe « $_{opt}$ » indique que la formule qu'il qualifie est optionnelle. Une formule avec des points de suspension, comme « *élément ... élément* », indique un élément pouvant apparaître un nombre quelconque, éventuellement nul, de fois.

instruction
→ *instruction-bloc*
→ *instruction-expression*
→ *instruction-goto*
→ *instruction-if*
→ *instruction-while*
→ *instruction-do*
→ *instruction-for*
→ *instruction-break*
→ *instruction-continue*
→ *instruction-switch*
→ *instruction-return*
→ *instruction-vide*
→ *identificateur* `:` *instruction*

instruction-bloc
→ `{` *déclaration ... déclaration*
instruction ... instruction `}`

instruction-expression
→ *expression* `;`

instruction-goto
→ `goto` *identif* `;`

instruction-if
→ `if (` *expression* `)` *instruction* `else` *instruction*
→ `if (` *expression* `)` *instruction*

instruction-while
→ `while (` *expression* `)` *instruction*

instruction-do
→ `do` *instuction* `while (` *expression* `);`

instruction-for
→ `for (` $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *instruction*

instruction-break
→ `break;`

instruction-continue
→ `continue;`

instruction-switch
→ `switch (` *expression* `)` `{` *instruction-ou-case ... instruction-ou-case* `}`

instruction-ou-case
→ `case` *expression-constante* `:` $instruction_{opt}$
→ `default :` *instruction*
→ *instruction*

instruction-return
→ `return` $expression_{opt}$ `;`

instruction-vide
→ `;`

C ET LE POINT-VIRGULE. Comme l'indique la syntaxe de l'*instruction-bloc*, en C le point-virgule n'est pas

un *séparateur* d'instructions mais un *terminateur* de certaines instructions. Autrement dit, il appartient à la syntaxe de chaque instruction de préciser si elle doit ou non être terminée par un point-virgule, indépendamment de ce par quoi l'instruction est suivie dans le programme.

L'oubli du point-virgule à la fin d'une instruction qui en requiert un est toujours une erreur, quelle que soit la situation de cette instruction. Un surnombre de points-virgules crée des instructions vides.

3.2 Présentation détaillée des instructions

3.2.1 Blocs

Un bloc est une suite de déclarations et d'instructions encadrée par les deux accolades { et }. Du point de vue de la syntaxe il se comporte comme une instruction unique et peut figurer en tout endroit où une instruction simple est permise.

Le bloc le plus extérieur d'une fonction et les autres blocs plus profonds ont le même statut. En particulier, quelle que soit sa position dans le programme, un bloc peut comporter ses propres déclarations de variables. Sauf si elle est déclarée `extern`, une variable définie dans un bloc est locale à ce bloc et donc inconnue à l'extérieur. Dans le bloc où elle est définie, une telle variable masque, sans le détruire, tout autre objet de même nom connu à l'extérieur du bloc.

Sauf si elles sont qualifiées `static` (ou `extern`), de telles variables sont créées et éventuellement initialisées lors de l'activation du bloc ; elles sont détruites dès que le contrôle quitte le bloc. Il ne faut donc pas espérer qu'une telle variable conserve sa valeur entre deux passages dans le bloc.

Les variables locales aux blocs permettent d'optimiser la gestion de l'espace local. Par exemple, dans un programme tel que

```
if (...) {
    type1 n;
    ...
}
else {
    type2 x;
    ...
}
```

les variables `n` et `x` n'existeront jamais simultanément ; le compilateur peut donc leur allouer le même emplacement de la mémoire.

3.2.2 Instruction-expression

Format :

expression ;

Mais oui, il suffit d'écrire un point-virgule derrière n'importe quelle expression pour en faire une instruction. Exemples :

```
123;                          a
i++;                          b
x = 2 * x + 3;                c
printf("%d\n", n);            d
```

Intuitivement, une instruction-expression représente l'ordre « évaluez cette expression, ensuite oubliez le résultat ». Il est clair que si l'expression n'a pas d'effet de bord, cela n'aura servi à rien (exemple *a*). L'aspect utile de cette notion est : toute expression avec effet de bord pourra être évaluée *uniquement pour son effet de bord* (exemple *b*). Avec deux cas particuliers très intéressants :

- puisque l'affectation est une expression, on retrouve bien l'instruction d'affectation, fondamentale dans tous les langages de programmation (exemple *c*) ;
- toute fonction peut être appelée « comme une procédure » (exemple[23] *d*), c'est-à-dire en ignorant la valeur qu'elle rend.

[23] On verra le moment venu (cf. section 7.2.5 *d*) que `printf` rend un résultat parfois utile.

REMARQUE. Une conséquence regrettable de tout cela est un piège assez vicieux tendu aux pascaliens. Imaginons que `lirecaractere` soit le nom d'une fonction sans argument. L'appel correct de cette fonction s'écrit :

```
lirecaractere();
```

Cependant, puisque le nom d'une fonction est une expression (une constante valant l'adresse de la fonctio) et que toute expression suivie d'un point-virgule est une instruction correcte, l'énoncé

```
lirecaractere;
```

sera trouvé légal par le compilateur. Or cette expression (tout à fait analogue à l'exemple *a* ci-dessus) ne produit pas l'appel de la fonction et ne traduit donc probablement pas la pensée du programmeur.

3.2.3 Etiquettes et instruction `goto`

Format :

étiquette `:` *instruction*

Une étiquette est un identificateur ; elle doit être placée devant une fonction, séparée de celle-ci par un caractère deux points. Elle n'a pas à faire l'objet d'une déclaration explicite ; il suffit de l'écrire devant une instruction pour qu'elle soit automatiquement connue comme un nom à portée locale. Elle est alors utilisable partout dans la fonction où elle apparaît (avant et après l'instruction qu'elle préfixe) et elle reste inconnue en dehors de la fonction.

L'instruction

`goto` *étiquette* `;`

transfère le contrôle à l'instruction préfixée par l'étiquette en question.

Théoriquement, tout algorithme peut être programmé sans utiliser l'instruction `goto`. Dans certains langages comme Pascal, elle est utilisée pour obtenir l'abandon d'une structure de contrôle (exemple : une boucle) depuis l'intérieur de la structure. Un tel emploi de `goto` est avantageusement remplacé en C par l'utilisation des instructions `return`, `break` et `continue`.

Il est donc rare que l'on ait besoin de l'instruction `goto` en C. Elle ne se révèle utile que lorsqu'il faut abandonner plusieurs structures de contrôle (`if`, `while`, `for`...) imbriquées les unes dans les autres. Exemple :

```
for (i = 0; i < N1; i++) {
    for (j = 0; j <= N2; j++)
        for (k = 0; k <= N2; k++) {
            ...
            if (...)
                goto grande_boucle;
            ...
        }
    ...
grande_boucle:          /* ici on a quitté les deux boucles internes (sur j et k) */
    ...                 /* mais on est toujours dans la boucle la plus externe (sur i) */
}
```

3.2.4 Instruction `if...else...`

Formats :

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

et

```
if (expression)
    instruction1
```

Dans la première forme, *expression* est évaluée : si elle est vraie (i.e. non nulle) $instruction_1$ est exécutée ; si elle est fausse (nulle) $instruction_2$ est exécutée. Dans la deuxième forme, *expression* est évaluée : si elle est vraie $instruction_1$ est exécutée ; sinon, rien n'est exécuté.

On notera que l'expression conditionnelle doit figurer entre parenthèses. Celles-ci font partie de la syntaxe du if, non de celle de l'expression.

Lorsque plusieurs instructions `if` sont imbriquées, il est convenu que chaque `else` se rapporte au *dernier* `if` pour lequel le compilateur a rencontré une condition suivie d'*exactement une* instruction. Le listing du programme peut (et doit !) traduire cela par une *indentation* (marge blanche) expressive, mais il ne faut pas oublier que le compilateur ne tient pas compte des marges. Par exemple, le programme suivant est probablement incorrect ; en tout cas, son indentation ne traduit pas convenablement les rapports entre les `if` et les `else` :

```
if (nombrePersonnes != 0)
    if (nombrePersonnes != nombreAdultes)
        printf("Il y a des enfants!");
else
    printf("Il n'y a personne!");
```

Ce problème se pose dans tous les langages qui offrent deux variétés d'instruction conditionnelle. On le résout soit par l'utilisation d'instructions vides :

```
if (nombrePersonnes != 0)
    if (nombrePersonnes != nombreAdultes)
        printf("Il y a des enfants!");
    else
        ;
else
    printf("Il n'y a personne!");
```

soit, plus simplement, en utilisant des blocs :

```
if (nombrePersonnes != 0) {
    if (nombrePersonnes != nombreAdultes)
        printf("Il y a des enfants!");
}
else
    printf("Il n'y a personne!");
```

REMARQUE. La syntaxe prévoit exactement *une* instruction entre la condition et le `else`. Par conséquent, un excès de points-virgules à la suite de $instruction_1$ constitue une erreur. Voici une faute qu'on peut faire quand on débute :

```
if (nombrePersonnes != 0) {
    if (nombrePersonnes != nombreAdultes)
        printf("Il y a des enfants!");
};
else
    printf("Il n'y a personne!");
```

Il y a maintenant *deux* instructions entre la ligne du `if` et celle du `else` : une instruction-bloc { ... } et une instruction vide « `;` ». Le compilateur signalera donc une erreur sur le `else`.

3.2.5 Instructions `while` et `do...while`

Ces instructions correspondent respectivement aux instructions `while...do...` et `repeat...until...` du langage Pascal. Notez que la syntaxe exige que la condition figure entre parenthèses. Formats :

`while (`*expression*`)`
    instruction

et

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

Le fonctionnement de ces instructions est décrit par les organigrammes de la figure 4. Fondamentalement, il s'agit de réitérer l'exécution d'une certaine instruction tant qu'une certaine instruction, vue comme une condition, reste vraie. Dans la structure `while` on vérifie la condition *avant* d'exécuter l'instruction, tandis que dans la structure `do...while` on la vérifie après.

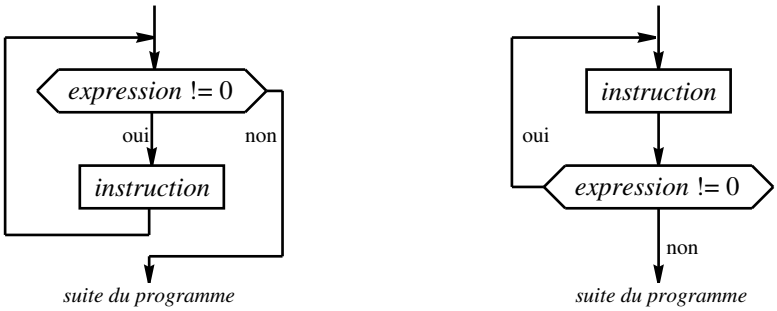

FIG. 4 – Instruction `while` (à gauche) et `do...while` (à droite)

L'instruction `do...while...` exécute donc au moins une fois l'instruction qui constitue son corps avant d'évaluer la condition de continuation. C'est en cela qu'elle est l'analogue de l'instruction `repeat...until` du Pascal. Mais on notera que la condition figurant dans une instruction `do...while` (« faire tant que... ») et celle qui figurerait dans une instruction `repeat...until` équivalente (« répéter jusqu'à ce que... ») sont inverses l'une de l'autre.

3.2.6 Instruction `for`

Format :

```
for ( expr1 ; expr2 ; expr3 )
    instruction
```

Par définition, cette construction équivaut *strictement* à celle-ci :

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

Ainsi, dans la construction `for(`$expr_1$ `;` $expr_2$ `;` $expr_3$`)` :
- $expr_1$ est l'expression qui effectue les initialisations nécessaires avant l'entrée dans la boucle ;
- $expr_2$ est le test de continuation de la boucle ; il est évalué avant l'exécution du corps de la boucle ;
- $expr_3$ est une expression (par exemple une incrémentation) évaluée à la fin du corps de la boucle.

Par exemple, l'instruction Pascal :

```
for i:=0 to 9 do
    t[i]:=0
```

se traduit tout naturellement en C

```
for (i = 0; i < 10; i++)
    t[i] = 0;
```

Les expressions $expr_1$ et $expr_3$ peuvent être absentes (les points-virgules doivent cependant apparaître). Par exemple

```
for ( ; expr2 ; )
    instruction
```

équivaut à

```
while (expr2)
    instruction
```

La condition de continuation $expr_2$ peut elle aussi être absente. On considère alors qu'elle est toujours vraie. Ainsi, la boucle « indéfinie » peut se programmer :

```
for ( ; ; )
    instruction
```

Bien entendu, il faut dans ce cas que l'abandon de la boucle soit programmé à l'intérieur de son corps (sinon cette boucle indéfinie devient infinie !). Exemple :


```
for (;;) {
    printf("donne un nombre (0 pour sortir): ");
    scanf("%d", &n);
    if (n == 0)
        break;
    ...
    exploitation de la donnée n
    ...
}
```

Lorsque l'abandon de la boucle correspond aussi à l'abandon de la procédure courante, `break` est avantageusement remplacée par `return` comme dans :

```
char *adresse_de_fin(char *ch) {
        /* renvoie adresse du caractère qui suit la chaîne ch */
    for (;;)
        if (*ch++ == 0)
            return ch;
}
```

3.2.7 Instruction `switch`

Format :

`switch (` *expression* `)`
corps

Le corps de l'instruction `switch` prend la forme d'un bloc {...} renfermant une suite d'instructions entre lesquelles se trouvent des constructions de la forme

`case` *expression-constante* `:`

ou bien

`default :`

Le fonctionnement de cette instruction est le suivant : expression est évaluée ;
- s'il existe un énoncé `case` avec une constante qui égale la valeur de expression, le contrôle est transféré à l'instruction qui suit cet énoncé ;
- si un tel `case` n'existe pas, et si l'énoncé `default` existe, alors le contrôle est transféré à l'instruction qui suit l'énoncé `default` ;
- si la valeur de expression ne correspond à aucun `case` et s'il n'y a pas d'énoncé `default`, alors aucune instruction n'est exécutée.

ATTENTION. Lorsqu'il y a branchement réussi à un énoncé `case`, *toutes les instructions qui le suivent* sont exécutées, jusqu'à la fin du bloc ou jusqu'à une instruction de rupture (`break`). Autrement dit, l'instruction `switch` s'apparente beaucoup plus à une sorte de `goto` « paramétré » (par la valeur de l'expression) qu'à l'instruction `case...of...` de Pascal.

Exemple (idiot) :

```
j = 0;
switch (i) {
    case 3:
        j++;
    case 2:
        j++;
    case 1:
        j++;
}
```

Si on suppose que `i` ne peut prendre que les valeurs 0 ... 3, alors l'instruction ci-dessus a le même effet que l'affectation `j = i`.

Pour obtenir un comportement similaire à celui du `case...of...` de Pascal, on doit utiliser l'instruction `break`, comme dans l'exemple suivant, dans lequel on compte la fréquence de chaque chiffre et des caractères blancs dans un texte :

```
nb_blancs = nb_autres = 0;
for (i = 0; i < 10; )
    nb_chiffre[i++] = 0;
while ((c = getchar()) != EOF)
    switch (c) {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            nb_chiffre[c - '0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nb_blancs++;
            break;
        default:
            nb_autres++;
    }
```

3.2.8 Instructions `break` et `continue`

Formats :

```
break;
continue;
```

Dans la portée d'une structure de contrôle (instructions `for`, `while`, `do` et `switch`), l'instruction `break` provoque l'abandon de la structure et le passage à l'instruction écrite immédiatement derrière. L'utilisation de `break` ailleurs qu'à l'intérieur d'une de ces quatre instructions constitue une erreur (que le compilateur signale).

Par exemple la construction

```
for (expr1 ; expr2 ; expr3) {
    ...
    break;
    ...
}
```

équivaut à

```
{
    for (expr1 ; expr2 ; expr3) {
        ...
        goto sortie;
        ...
    }
sortie: ;
}
```

L'instruction `continue` est moins souvent utilisée. Dans la portée d'une structure de contrôle de type boucle (`while`, `do` ou `for`), elle produit l'abandon de l'itération courante et, le cas échéant, le démarrage de l'itération suivante. Elle agit comme si les instructions qui se trouvent entre l'instruction `continue` et la fin du corps de la boucle avaient été supprimées *pour l'itération en cours* : l'exécution continue par le test de la boucle (précédé, dans le cas de `for`, par l'exécution de l'expression d'incrémentation).

Lorsque plusieurs boucles sont imbriquées, c'est la plus profonde qui est concernée par les instructions `break` et `continue`. Dans ce cas, l'emploi de ces instructions ne nous semble pas œuvrer pour la clarté des programmes.

3.2.9 Instruction `return`

Formats :

`return` *expression* `;`

et

`return ;`

Dans un cas comme dans l'autre l'instruction `return` provoque l'abandon de la fonction en cours et le retour à la fonction appelante.

Dans la première forme *expression* est évaluée ; son résultat est la valeur que la fonction renvoie à la fonction appelante ; c'est donc la valeur que l'appel de la fonction représente dans l'expression où il figure. Si nécessaire, la valeur de *expression* est convertie dans le type de la fonction (déclaré dans l'en-tête), les conversions autorisées étant les mêmes que celles faites à l'occasion d'une affectation.

Dans la deuxième forme, la valeur retournée par la fonction reste indéterminée. On suppose dans ce cas que la fonction appelante n'utilise pas cette valeur ; il est donc prudent de déclarer `void` cette fonction.

Absence d'instruction `return` dans une fonction. Lorsque la dernière instruction (dans l'ordre de l'exécution) d'une fonction est terminée, le contrôle est rendu également à la procédure appelante. Tout se passe comme si l'accolade fermante qui termine le corps de la fonction était en réalité écrite sous la forme

```
    ...
    return;
}
```

4 Fonctions

Beaucoup de langages distinguent deux sortes de sous-programmes[24] : les *fonctions* et les *procédures*. L'appel d'une fonction est une expression, tandis que l'appel d'une procédure est une instruction. Ou, si on préfère, l'appel d'une fonction renvoie un résultat, alors que l'appel d'une procédure ne renvoie rien.

En C on retrouve ces deux manières d'appeler les sous-programmes, mais du point de la syntaxe le langage ne connaît que les fonctions. Autrement dit, un sous-programme est toujours supposé renvoyer une valeur, même lorsque celle-ci n'a pas de sens ou n'a pas été spécifiée. C'est pourquoi on ne parlera ici que de fonctions.

C'est dans la syntaxe et la sémantique de la définition et de l'appel des fonctions que résident les principales différences entre le C original et le C ANSI. Nous expliquons principalement le C ANSI, rassemblant dans une section spécifique (cf. section 4.2) la manière de faire du C original.

4.1 Syntaxe ANSI ou “avec prototype”

4.1.1 Définition

Une fonction se définit par la construction :

type$_{opt}$ *ident* `(` *déclaration-un-ident* `,` ... *déclaration-un-ident* `)`
instruction-bloc

Notez qu'il n'y a pas de point-virgule derrière le “`)`” de la première ligne. La présence d'un point-virgule à cet endroit provoquerait des erreurs bien bizarres, car la définition serait prise pour une déclaration (cf. section 4.1.4).

La syntaxe indiquée ici est incomplète ; elle ne convient qu'aux fonctions dont le type est défini simplement, par un identificateur. On verra ultérieurement (cf. section 5.4.1) la manière de déclarer des fonctions rendant un résultat d'un type plus complexe.

La première ligne de la définition d'une fonction s'appelle l'*en-tête*, ou parfois le *prototype*, de la fonction. Chaque formule *déclaration-un-ident* possède la même syntaxe qu'une déclaration de variable[25].

Exemple.

```
int extract(char *dest, char *srce, int combien) {
                    /* copie dans dest les combien premiers caractères de srce */
                    /* renvoie le nombre de caractères effectivement copiés    */
    int compteur;
    for (compteur = 0; compteur < combien && *srce != '\0'; compteur++)
        *dest++ = *srce++;
    *dest = '\0';
    return compteur;
}
```

Contrairement à d'autres langages, en C on ne peut pas définir une fonction à l'intérieur d'une autre : toutes les fonctions sont au même niveau, c'est-à-dire globales. C'est le cas notamment de `main`, qui est une fonction comme les autres ayant pour seule particularité un nom convenu.

4.1.2 Type de la fonction et des arguments

L'en-tête de la fonction définit le type des objets qu'elle renvoie. Ce peut être :
– tout type numérique ;
– tout type pointeur ;
– tout type `struct` ou `union`.

Si le type de la fonction n'est pas indiqué, le compilateur suppose qu'il s'agit d'une fonction à résultat entier. Ainsi, l'exemple précédent aurait aussi pu être écrit de manière équivalente (mais cette pratique n'est pas conseillée) :

[24] La notion de sous-programme (comme les procedures de Pascal, les subroutines de Fortran, etc.) est supposée ici connue du lecteur.

[25] Restriction : on n'a pas le droit de « mettre en facteur » un type commun à plusieurs arguments. Ainsi, l'en-tête de la fonction extract donnée en exemple *ne peut pas* s'écrire sous la forme `char *extract(char *dest, *srce, int n)`.

```
extract(char *dest, char *srce, int combien)
    etc.
```

FONCTIONS SANS RÉSULTAT. Lorsqu'une fonction ne renvoie pas une valeur, c'est-à-dire lorsqu'elle correspond plus à une procédure qu'à une vraie fonction, il est prudent de la déclarer comme rendant un objet de type `voidvoid`. Ce type est garanti *incompatible avec tous les autres types* : une tentative d'utilisation du résultat de la fonction provoquera donc une erreur à la compilation. Le programmeur se trouve ainsi à l'abri d'une utilisation intempestive du résultat de la fonction. Exemple :

```
void extract(char *dest, char *srce, int combien) {
                    /* copie dans dest les combien premiers caractères de srce */
                    /* maintenant cette fonction ne renvoie rien               */
    int compteur;
    for (compteur = 0; compteur < combien && *srce != '\0'; compteur++)
        *dest++ = *srce++;
    *dest = '\0';
}
```

FONCTIONS SANS ARGUMENTS. Lorsqu'une fonction n'a pas d'arguments, sa définition prend la forme

type$_{opt}$ *ident* `( void )`
    instruction-bloc

On notera que, sauf cas exceptionnel, on ne doit pas écrire une paire de parenthèses vide dans la déclaration ou la définition d'une fonction. En effet, un en-tête de la forme

type ident`()`

ne signifie pas que la fonction n'a pas d'arguments, mais (cf. section 4.2.2) que le programmeur ne souhaite pas que ses appels soient contrôlés par le compilateur.

4.1.3 Appel des fonctions

L'appel d'une fonction se fait en écrivant son nom, suivi d'une paire de parenthèses contenant éventuellement une liste d'arguments effectifs.

Notez bien que les parenthèses doivent toujours apparaître, même si la liste d'arguments est vide : si un nom de fonction apparaît dans une expression sans les parenthèses, alors il a la valeur d'une constante adresse (l'adresse de la fonction) et aucun appel de la fonction n'est effectué.

PASSAGE DES ARGUMENTS. En C, le passage des arguments se fait *par valeur*. Cela signifie que les arguments formels[26] de la fonction représentent d'authentiques variables locales initialisées, lors de l'appel de la fonction, par les *valeurs* des arguments effectifs[27] correspondants.

Supposons qu'une fonction ait été déclarée ainsi

type fonction `(` *type*$_1$ *arg_formel*$_1$ `,` ... *type*$_k$ *arg_formel*$_k$ `)`
    etc.

alors, lors d'un appel tel que

fonction `(` *arg_effectif*$_1$ `,` ... *arg_effectif*$_k$ `)`

la transmission des valeurs des arguments se fait comme si on exécutait les affectations :

arg_formel$_1$ `=` *arg_effectif*$_1$
...
arg_formel$_k$ `=` *arg_effectif*$_k$

Cela a des conséquences très importantes :
- les erreurs dans le nombre des arguments effectifs sont détectées et signalées,
- si le type d'un argument effectif n'est pas compatible avec celui de l'argument formel correspondant, une erreur est signalée,
- les valeurs des arguments effectifs subissent les conversions nécessaires avant d'être rangées dans les arguments formels correspondants (exactement les mêmes conversions qui sont faites lors des affectations).

[26] Les *arguments formels* d'une fonction sont les identificateurs qui apparaissent dans la définition de la fonction, déclarés à l'intérieur de la paire de parenthèses.

[27] Les *arguments effectifs* d'un appel de fonction sont les expressions qui apparaissent dans l'expression d'appel, écrites à l'intérieur de la paire de parenthèses caractéristiques de l'appel.

REMARQUE. Le langage ne spécifie pas l'ordre chronologique des évaluations des arguments effectifs d'un appel de fonction. Ces derniers doivent donc être sans effets de bord les uns sur les autres. Par exemple, il est impossible de prévoir quelles sont les valeurs effectivement passées à une_fonction lors de l'appel :

```
x = une_fonction(t[i++], t[i++]);     /* ERREUR !!! */
```

Si i_0 est la valeur de `i` juste avant l'exécution de l'instruction ci-dessus, alors cet appel peut aussi bien se traduire par `une_fonction(t[`i_0`], t[`$i_0 + 1$`])` que par `une_fonction(t[`$i_0 + 1$`], t[`i_0`])` ou même par `une_fonction(t[`i_0`], t[`i_0`])`. Ce n'est sûrement pas indifférent !

APPEL D'UNE FONCTION INCONNUE. En C une fonction peut être appelée alors qu'elle n'a pas été définie (sous-entendu : entre le début du fichier et l'endroit où l'appel figure). Le compilateur suppose alors
- que la fonction renvoie un `int`,
- que le nombre et les types des arguments formels de la fonction sont ceux qui correspondent aux arguments effectifs de l'appel (ce qui, en particulier, empêche les contrôles et conversions mentionnés plus haut).

De plus, ces hypothèses sur la fonction constituent pour le compilateur une première déclaration de la fonction. Toute définition ou déclaration ultérieure tant soit peu différente sera qualifiée de « redéclaration illégale »[28].

Lorsque les hypothèses ci-dessus ne sont pas justes, en particulier lorsque la fonction ne renvoie pas un `int`, il faut :
- soit écrire la définition de la fonction appelée avant celle de la fonction appelante,
- soit écrire, avant l'appel de la fonction, une déclaration externe de cette dernière, comme expliqué à la section 4.1.4.

4.1.4 Déclaration “externe” d'une fonction

Une déclaration externe d'une fonction est une déclaration qui n'est pas en même temps une définition. On « annonce » l'existence de la fonction (définie plus loin, ou dans un autre fichier) tout en précisant le type de son résultat et le nombre et les types de ses arguments, mais on ne donne pas son corps, c'est-à-dire les instructions qui la composent.

Cela se fait en écrivant
- soit un en-tête identique à celui qui figure dans la définition de la fonction (avec les noms des arguments formels), suivi d'un point-virgule ;
- soit la formule obtenue à partir de l'en-tête précédent, en y supprimant les noms des arguments formels.

Par exemple, si une fonction a été définie ainsi

```
void truc(char dest[80], char *srce, unsigned long n, float x) {
    corps de la fonction
}
```

alors des déclarations externes correctes sont :

```
extern void truc(char dest[80], char *srce, unsigned long n, float x);
```

ou

```
ou extern void machin(char [80], char *, unsigned long, float);
```

Ces expressions sont appelées des *prototypes* de la fonction. Le mot `extern` est facultatif, mais le point-virgule est essentiel. Dans la première forme, les noms des arguments formels sont des identificateurs sans aucune portée.

4.2 Syntaxe originale ou “sans prototype”

4.2.1 Déclaration et définition

DÉFINITION. En syntaxe originale la définition d'une fonction prend la forme

type$_{opt}$ *ident* `(` *ident* `,` ... *ident* `)`
déclaration ... *déclaration*
instruction-bloc

[28] C'est une erreur surprenante, car le programmeur, oubliant que l'appel de la fonction a entraîné une déclaration implicite, conçoit sa définition ou déclaration ultérieure comme étant la *première* déclaration de la fonction.

Les parenthèses de l'en-tête ne contiennent ici que la liste des noms des arguments formels. Les déclarations de ces arguments se trouvent immédiatement *après l'en-tête, avant l'accolade* qui commence le corps de la fonction.

Exemple :

```
int extract(dest, srce, combien)
                    /* copie dans dest les combien premiers caractères de srce */
                    /* renvoie le nombre de caractères effectivement copiés    */
    char *dest, char *srce;
    int combien;
{
    int compteur;
    for (compteur = 0; compteur < combien && *srce != '\0'; compteur++)
        *dest++ = *srce++;
    *dest = '\0';
    return compteur;
}
```

Déclaration externe. En syntaxe originale, la déclaration externe de la fonction précédente prend une des formes suivantes :

```
extern int extract();
```

ou

```
int extract();
```

Comme on le voit, ni les types, ni même les noms, des arguments formels n'apparaissent dans une déclaration externe[29].

4.2.2 Appel

En syntaxe originale, lors d'un appel de la fonction le compilateur *ne tient aucun compte ni du nombre ni des types des arguments formels*[30], indiqués lors de la définition de la fonction. Chaque argument est évalué séparément, puis

- les valeurs des arguments effectifs de type `char` ou `short` sont converties dans le type `int` ;
- les valeurs des arguments effectifs de type `float` sont converties dans le type `double` ;
- les valeurs des arguments effectifs d'autres types sont laissées telles quelles.

Juste avant le transfert du contrôle à la fonction, ces valeurs sont copiées dans les arguments formels correspondants. Ou plus exactement, dans ce que la fonction appelante croit être les emplacements des arguments formels de la fonction appelée : il n'y a aucune vérification de la concordance, ni en nombre ni en type, des arguments formels et des arguments effectifs. Cela est une source d'erreurs très importante. Par exemple, la fonction `carre` ayant été ainsi définie

```
double carre(x)
    double x;
{
    return x * x;
}
```

l'appel

```
x = carre(2);
```

est erroné, car la fonction appelante dépose la valeur entière 2 à l'adresse du premier argument formel (qu'elle « croit » entier). Or la fonction appelée croit recevoir le codage d'un nombre flottant double. Le résultat n'a aucun sens. Des appels corrects auraient été

```
x = carre((double) 2);
```

ou

```
x = carre(2.0);
```

[29] Par conséquent, les déclarations externes montrées ici sont sans aucune utilité, puisque `int` est le type supposé des fonctions non déclarées.

[30] Mais oui, vous avez bien lu ! C'est là la principale caractéristique de la sémantique d'appel des fonctions du C original.

4.2.3 Coexistence des deux syntaxes

A côté de ce que nous avons appelé la syntaxe ANSI, la syntaxe originale pour la définition et la déclaration externe des fonctions fait partie elle aussi du C ANSI ; de nos jours les programmeurs ont donc le choix entre l'une ou l'autre forme. Lorsque la fonction a été définie ou déclarée sous la syntaxe originale, les appels de fonction sont faits sans vérification ni conversion des arguments effectifs. Au contraire, lorsque la fonction a été spécifiée sous la syntaxe ANSI, ces contrôles et conversions sont effectués.

A cause de cette coexistence, si une fonction n'a pas d'argument, la définition de son en-tête en C ANSI doit s'écrire sous la forme bien peu heureuse :

type$_{opt}$ *ident* `( void )`

car une paire de parenthèses sans rien dedans signifierait non pas que la fonction n'a pas d'arguments, mais qu'elle est déclarée sans prototype, c'est-à-dire que ses appels doivent être traités sans contrôle des arguments. Exemples :

```
int getchar(void);      /* une fonction sans arguments dont les appels seront contrôlés */

double moyenne();       /* une fonction avec ou sans arguments, aux appels incontrôlés */
```

4.3 Arguments des fonctions

4.3.1 Passage des arguments

L'idée maîtresse est qu'en C le passage des arguments des fonctions se fait *toujours par valeur*. Après avoir fait les conversions opportunes la valeur de chaque argument effectif est affectée à l'argument formel correspondant.

Si l'argument effectif est d'un type simple (nombre, pointeur) ou d'un type `struct` ou `union`, sa valeur est recopiée dans l'argument formel correspondant, quelle que soit sa taille, c'est-à-dire quel que soit le nombre d'octets qu'il faut recopier.

4.3.2 Arguments de type tableau

Apparaissant dans la partie exécutable d'un programme, le nom d'un tableau, et plus généralement toute expression déclarée « tableau de T », est considérée comme ayant pour type « adresse d'un T » et pour valeur l'adresse du premier élément du tableau. Cela ne fait pas intervenir la taille effective du tableau[31]. Ainsi lorsqu'un argument effectif est un tableau, l'objet effectivement passé à la fonction est uniquement l'adresse du premier élément et il suffit que l'espace réservé dans la fonction pour un tel argument ait la taille, fixe, d'un pointeur. D'autre part, en C il n'est jamais vérifié que les indices des tableaux appartiennent bien à l'intervalle $0...N-1$ déterminé par le nombre d'éléments indiqué dans la déclaration. Il en découle la propriété suivante :

Dans la déclaration d'un argument formel `t` de type « tableau de T » :
– l'indication du nombre d'éléments de `t` est sans utilité[32] ;
– les formules « *type* `t[ ]` » et « *type* `*t` » sont tout à fait équivalentes ;
– la fonction pourra être appelée indifféremment avec un tableau ou un pointeur pour argument effectif.

Exemple. L'en-tête « vague » de la fonction suivante

```
int strlen(chaîne de caractères s) {
    int i = 0;
    while (s[i] != 0)
        i++;
    return i;
}
```

peut indifféremment être concrétisée de l'une des trois manières suivantes :

```
int strlen(char s[80])

int strlen(char s[])

int strlen(char *s)
```

[31] Il n'en est pas de même lors de la définition, où la taille du tableau ne peut pas être ignorée (elle est indispensable pour l'allocation de l'espace mémoire).
[32] Attention, cette question se complique notablement dans le cas des tableaux multidimensionnels. Cf. 6.2.3

Dans les trois cas, si `t` est un tableau de `char` et `p` l'adresse d'un `char`, les trois appels suivants sont corrects :

```
l1 = strlen(t);

l2 = strlen(p);

l3 = strlen("Bonjour");
```

A retenir : parce que les arguments effectifs sont passés par valeur, les tableaux sont passés par adresse.

Passage par valeur des tableaux. Cependant, puisque les structures sont passées par valeur, elles fournissent un moyen pour obtenir le passage par valeur d'un tableau, bien que les occasions où cela est nécessaire semblent assez rares. Il suffit de déclarer le tableau comme le champ unique d'une structure « enveloppe » ne servant qu'à cela. Exemple :

```
struct enveloppe {
    int t[100];
};
void possible_modification(struct enveloppe x) {
    x.t[50] = 2;
}
main() {
    struct enveloppe x;
    x.t[50] = 1;
    possible_modification(x);
    printf("%d\n", x.t[50]);
}
```

La valeur affichée est 1, ce qui prouve que la fonction appelée n'a modifié qu'une copie locale du *tableau enveloppé*, c'est-à-dire que celui-ci a bien été passé par valeur (les structures seront vues à la section 5.2.1).

4.3.3 Arguments par adresse

Les arguments qui sont d'un type simple (`char`, `int`, pointeur...) ou bien des `struc` ou des `union` sont toujours passés par valeur. Lorsque l'argument effectif est une variable, ce mécanisme empêche qu'elle puisse être modifiée par la fonction appelée. Or une telle modification est parfois souhaitable ; elle requiert que la fonction puisse accéder non seulement à la valeur de la variable, mais aussi à son adresse.

C'est ce qui s'appelle le *passage par adresse* des arguments. Alors que certains langages le prennent en charge, C ne fournit aucun mécanisme spécifique : le passage de l'adresse de l'argument doit être programmé explicitement en utilisant comme argument un pointeur vers la variable.

Par exemple, supposons avoir à écrire une fonction

```
int quotient(a, b)
```

censée renvoyer le quotient de la division euclidienne de l'entier `a` par l'entier `b` et devant en plus remplacer `a` par le reste de cette division. On devra la définir :

```
int quotient(int *a, int b) {
    int r = *a / b;
    *a = *a % b;
    return r;
}
```

Ci-dessus, `b` représente un entier ; `a` l'adresse d'un entier. La notation `*a` fait référence à une variable (ou, plus généralement, une *lvalue*) appartenant à la procédure appelante.

L'argument effectif correspondant à un tel argument formel doit être une adresse. Souvent, cela est obtenu par l'utilisation de l'opérateur `&`. Par exemple, si `x`, `y` et `z` sont des variables entières, un appel correct de la fonction précédente sera

```
z = quotient(&x, y);
```

Bien entendu, l'opérateur `&` ne doit pas être utilisé si l'argument effectif est déjà une adresse. Le schéma suivant résume les principales situations possibles. Soient les fonctions

```
f(int a) {
    ...
}
g(int *b) {
    ...
}
h(int c, int *d) {
    ...
}
```

A l'intérieur de la fonction `h`, des appels corrects de `f` et `g` sont :

```
f(c);
f(*d);
g(&c);
g(d);
```

cette dernière expression correspondant, bien sûr, à `g(&*d)`.

4.3.4 Arguments en nombre variable

Le langage C permet de définir des fonctions dont le nombre d'arguments n'est pas fixé et peut varier d'un appel à un autre. Nous exposons ici la manière dont cette facilité est réalisée dans le C ANSI. Dans le C original elle existe aussi ; elle consiste à utiliser, sans garde-fou, une faiblesse du langage (la non-vérification du nombre d'arguments effectifs) et une caractéristique du système sous-jacent (le sens d'empilement des arguments effectifs). Mais tout cela est suffisamment casse-cou pour que nous préférions n'expliquer que la manière ANSI de faire, un peu plus fiable.

Une fonction avec un nombre variable d'arguments est déclarée en explicitant quelques arguments fixes, au moins un, suivis d'une virgule et de trois points : « `, ...` » . Exemple :

```
int max(short n, int x0, ...)
```

Une telle fonction peut être appelée avec un nombre quelconque d'arguments effectifs, mais il faut qu'il y en ait au moins autant qu'il y a d'arguments formels nommés, ici deux. Exemple d'appel :

```
m = max(3, p, q, r);
```

Pour les arguments nommés, l'affectation des valeurs des arguments effectifs aux arguments formels est faite comme d'ordinaire. Pour les arguments anonymes elle s'effectue comme pour une fonction sans prototype : les `char` et les `short` sont convertis en `int`, les `float` en `double` et il n'y a aucune autre conversion.

A l'intérieur de la fonction on accède aux arguments anonymes au moyen des macros `va_start` et `va_arg` définies dans le fichier `stdarg.h`, comme le montre l'exemple suivant :

```
#include <stdarg.h>

int max(short n, int x1, ...) {
    va_list magik;
    int x, i, m;

    m = x1;
    va_start(magik, x1);
    for (i = 2; i <= n; i++)
        if ((x = va_arg(magik, int)) > m)
            m = x;
    return m;
}
```

Des appels corrects de cette fonction seraient :

```
a = max(3, p, q, r);
b = max(8, x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]);
c = max(2, u, 0);
```

Les éléments définis dans le fichier `stdarg.h` sont :

`va_list` *pointeur* Déclaration de la variable *pointeur*, qui sera automatiquement gérée par le dispositif d'accès aux arguments. Le programmeur choisit le nom de cette variable, mais il n'en fait rien d'autre que la mettre comme argument dans les appels des macros `va_start` et `va_arg`.

`va_start(`*pointeur*`,` *dernier_argument*`)` Initialisation de la variable *pointeur* ; *dernier_argument* doit être le dernier des arguments explicitement nommés dans l'en-tête de la fonction.

`va_arg(`*pointeur*`,` *type*`)` Parcours des arguments anonymes : le premier appel de cette macro donne le premier argument anonyme ; chaque appel suivant donne l'argument suivant le dernier déjà obtenu. Chaque fois, *type* doit décrire le type de l'argument qui est en train d'être référencé.

Comme l'exemple le montre, le principe des fonctions avec un nombre variable d'arguments est de déclarer effectivement au moins un argument, dont la fonction peut déduire l'adresse des autres. Il faut aussi choisir le moyen d'indiquer à la fonction le nombre de valeurs réellement fournies. Dans l'exemple ci-dessus ce nombre est passé comme premier argument ; une autre technique consiste à utiliser une valeur « intruse » (ex : le pointeur `NULL` parmi des pointeurs valides). La fonction `printf` est un autre exemple de fonction avec un nombre variable d'arguments. Elle est déclarée (dans le fichier `stdio.h`) :

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

Ici encore, le premier argument (le format) renseigne sur le nombre et la nature des autres arguments.

La question des fonctions formelles (fonctions arguments d'autres fonctions) et d'autres remarques sur les fonctions et leurs adresses sont traitées à la section 6.3.1.

5 Objets structurés

5.1 Tableaux

5.1.1 Cas général

Dans le cas le plus simple la déclaration d'un tableau se fait par une formule comme :

type-de-base nom `[` $expression_{opt}$ `];`

Exemple :

```
unsigned long table[10];
```

L'expression optionnelle qui figure entre les crochets spécifie le nombre d'éléments du tableau. Le premier élément possède toujours l'indice 0. Par conséquent, un tableau t déclaré de taille N possède les éléments $t_0, t_1, ... t_{N-1}$. Ainsi la définition précédente alloue un tableau de 10 éléments, nommés respectivement `table[0]`, `table[1]`, ... `table[9]`.

L'expression optionnelle qui figure entre les crochets doit être de type entier et constante au sens de la section 1.3.4, c'est-à-dire une expression dont tous les éléments sont connus au moment de la compilation. De cette manière le compilateur peut l'évaluer et connaître la quantité d'espace nécessaire pour loger le tableau. Elle est obligatoire lorsqu'il s'agit d'une définition, car il y a alors allocation effective du tableau. Elle est facultative dans le cas des déclarations qui ne sont pas des définitions, c'est-à-dire :
- lors de la déclaration d'un tableau qui est un argument formel d'une fonction ;
- lors de la déclaration d'un tableau externe (défini dans un autre fichier).

Sémantique du nom d'un tableau. En général lorsque le nom d'un tableau apparaît dans une expression il y joue le même rôle qu'une constante de type adresse ayant pour valeur l'adresse du premier élément du tableau. Autrement dit, si `t` est de type tableau, les deux expressions

`t` `&t[0]`

sont équivalentes.

Les exceptions à cette règle, c'est-à-dire les expressions de type tableau qui ne sont pas équivalentes à l'adresse du premier élément du tableau, sont
- l'occurrence du nom du tableau dans sa propre déclaration ;
- l'apparition d'un nom du tableau comme argument de `sizeof`.

Ce sont les seules occasions où le compilateur veut bien se souvenir qu'un nom de tableau représente aussi un espace mémoire *possédant une certaine taille*.

De tout cela, il faut retenir en tout cas que le nom d'un tableau n'est pas une *lvalue* et donc qu'il n'est pas possible d'affecter un tableau à un autre. Si `a` et `b` sont deux tableaux, même ayant des types identiques, l'affectation `b = a` sera
- comprise comme l'affectation d'une adresse à une autre, et
- rejetée, car `b` n'est pas modifiable.

Tableaux multidimensionnels. C ne prévoit que les tableaux à un seul indice, mais les éléments des tableaux peuvent à leur tour être des tableaux. La déclaration d'un tableau avec un nombre quelconque d'indices suit la syntaxe :

type nom `[` $expression_{opt}$ `]` `[` $expression_{opt}$ `]` ... `[` $expression_{opt}$ `]`

qui est un cas particulier de formules plus générales expliquées à la section 5.4.1. Par exemple, la déclaration

```
double matrice[10][20];
```

introduit `matrice` comme étant le nom d'un tableau de 10 éléments[33] qui, chacun à son tour, est un tableau de 20 éléments de type `double`.

5.1.2 Initialisation des tableaux

Une variable de type tableau peut être initialisée lors de sa déclaration. Cela se fait par une expression ayant la syntaxe :

`{` *expression* `,` *expression* `,` ... *expression* `}`

[33] Dans le cas d'une matrice, la coutume est d'appeler ces éléments les *lignes* de la matrice. Cela permet de dire alors : « en C les matrices sont rangées par lignes ».

Exemple :

```
int t[5] = { 10, 20, 30, 40, 50 };
```

Les expressions indiquées doivent être des expressions constantes au sens de la section 1.3.4. S'il y a moins d'expressions que le tableau a d'éléments, les éléments restants sont remplis de zéros (cas des variables globales) ou bien restent indéterminés (cas des variables locales). S'il y a plus d'expressions d'initialisation que d'éléments dans le tableau, une erreur est signalée.

D'autre part, la présence d'un initialisateur dispense d'indiquer la taille du tableau. Il est convenu que ce dernier doit avoir alors pour taille le nombre effectif de valeurs initiales. Par exemple, la définition

```
int t[] = { 10, 20, 30, 40, 50 };
```

alloue un tableau à 5 composantes garni avec les valeurs indiquées.

Lorsque le tableau comporte des sous-tableaux, c'est-à-dire lorsque c'est un tableau à plusieurs indices, la liste des expressions d'initialisation peut comporter des sous-listes indiquées à leur tour avec des accolades, mais ce n'est pas une obligation. Le cas échéant, la règle de complétion par des zéros s'applique aussi aux sous-listes. Par exemple, les déclarations

```
int t1[4][4] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
int t2[4][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

allouent et garnissent les deux tableaux de la figure 5.

FIG. 5 – Tableaux initialisés

REMARQUE. Rappelons que ces représentations rectangulaires sont très conventionnelles. Dans la mémoire de l'ordinateur ces deux tableaux sont plutôt arrangés comme sur la figure .

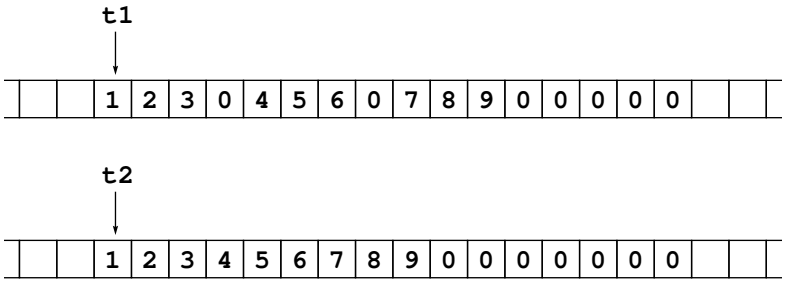

FIG. 6 – Tableaux « à plat »

Si la taille d'une composante est un diviseur de la taille d'un entier, les tableaux sont « tassés » : chaque composante occupe le moins de place possible. En particulier, dans un tableau de `char` (resp. de `short`), chaque composante occupe un (resp. deux) octet(s).

5.1.3 Chaînes de caractères

Les chaînes de caractères sont représentées comme des tableaux de caractères. Un caractère nul suit le dernier caractère utile de la chaîne et en indique la fin. Cette convention est suivie par le compilateur (qui range les chaînes constantes en leur ajoutant un caractère nul), ainsi que par les fonctions de la librairie standard qui construisent des chaînes. Elle est supposée vérifiée par les fonctions de la librairie standard qui exploitent des chaînes. Par conséquent, avant de passer une chaîne à une telle fonction, il faut s'assurer qu'elle comporte bien un caractère nul à la fin.

Donnons un exemple classique de traitement de chaînes : la fonction `strlen` (extraite de la bibliothèque standard) qui calcule le nombre de caractères utiles d'une chaîne :

```
int strlen(char s[]) {
    register int i = 0;
    while (s[i++] != '\0')
        ;
    return i - 1;
}
```

Une constante-chaîne de caractères apparaissant dans une expression a le même type qu'un tableau de caractères c'est-à-dire, pour l'essentiel, le type « adresse d'un caractère ». Par exemple, si `p` a été déclaré

```
char *p;
```

l'affectation suivante est tout à fait légitime :

```
p = "Bonjour. Comment allez-vous?";
```

Elle affecte à `p` l'adresse de la zone de la mémoire où le compilateur a rangé le texte en question (l'adresse du `'B'` de `"Bonjour..."`). Dans ce cas, bien que l'expression `*p` soit une *lvalue* (ce qui est une propriété syntaxique), elle ne doit pas être considérée comme telle car `*p` représente un objet mémorisé dans la zone des constantes, et toute occurrence de `*p` comme opérande gauche d'une affectation :

`*p =` *expression*`;`

constituerait une tentative de modification d'un objet constant. Cette erreur est impossible à déceler à la compilation. Selon le système sous-jacent, soit une erreur sera signalée à l'exécution, provoquant l'abandon immédiat du programme, soit aucune erreur ne sera signalée mais la validité de la suite du traitement sera compromise.

Initialisation. Comme tous les tableaux, les variables-chaînes de caractères peuvent être initialisées lors de leur déclaration. De ce qui a été dit pour les tableaux en général il découle que les expressions suivantes sont correctes :

```
char message1[80] = { 'S', 'a', 'l', 'u', 't', '\0' };
char message2[]   = { 'S', 'a', 'l', 'u', 't', '\0' };
```

(la deuxième formule définit un tableau de taille 6). Heureusement, C offre un raccourci : les deux expressions précédentes peuvent s'écrire de manière tout à fait équivalente :

```
char message1[80] = "Salut";
char message2[]   = "Salut";
```

Il faut bien noter cependant que l'expression `"Salut"` qui apparaît dans ces deux exemples n'est pas du tout traité par le compilateur comme les autres chaînes de caractères constantes rencontrées dans les programmes. Ici, il ne s'agit que d'un moyen commode pour indiquer une collection de valeurs initiales à ranger dans un tableau. A ce propos, voir aussi la remarque 2 de la section 6.2.1.

5.2 Structures et unions

5.2.1 Structures

Les structures sont des variables composées de champs de types différents. Elles se déclarent sous la syntaxe suivante :

$$\texttt{struct}\ nom_{opt} \left\{ \begin{array}{c} \texttt{'\{'}\ declaration\ ...\ declaration\ \texttt{'\}'} \\ rien \end{array} \right\} \left\{ \begin{array}{c} nom\ \texttt{','}\ ...\ nom \\ rien \end{array} \right\} \texttt{';'}$$

A la suite du mot-clé `struct` on indique :
- facultativement, le nom que l'on souhaite donner à la structure. Par la suite, l'expression « `struct` *nom* » pourra être employée comme une sorte de nom de type ;
- facultativement, entre accolades, la liste des déclarations des champs de la structure. Chaque champ peut avoir un type quelconque (y compris `struct`, bien sûr) ;
- facultativement, la liste des variables que l'on définit ou déclare comme possédant la structure ici définie.

Exemple.

```
struct fiche {
    int numero;
    char nom[32], prenom[32];
} a, b, c;
```


Cette déclaration introduit trois variables `a`, `b` et `c`, chacune constituée des trois champs `numero`, `nom` et `prenom` ; en même temps elle donne à cette structure le nom `fiche`. Plus loin, on pourra déclarer de nouvelles variables analogues à `a`, `b` et `c` en écrivant simplement

```
struct fiche x, y;
```

(remarquez qu'il faut écrire « `struct fiche` » et non pas « `fiche` »). Nous aurions pu aussi bien faire les déclarations :

```
struct fiche {
    int numero;
    char nom[32], prenom[32];
};
struct fiche a, b, c, x, y;
```

ou encore

```
struct {
    int numero;
    char nom[32], prenom[32];
} a, b, c, x, y;
```

mais cette dernière forme nous aurait empêché par la suite de déclarer aussi facilement d'autres variables de même type.

Comme dans beaucoup de langages, on accède aux champs d'une variable de type structure au moyen de l'opérateur « `.` ». Exemple :

```
a.numero = 1234;
```

Les structures supportent les manipulations « globales »[34] :
- on peut affecter une expression d'un type structure à une variable de type structure (pourvu que ces deux types soient identiques) ;
- les structures sont passées *par valeur* lors de l'appel des fonctions ;
- le résultat d'une fonction peut être une structure.

La possibilité pour une fonction de rendre une structure comme résultat est officielle dans le C ANSI ; sur ce point les compilateurs plus anciens présentent des différences de comportement. D'autre part il est facile de voir que -sauf dans le cas de structures vraiment très simples- les transmettre comme résultats des fonctions est en général peu efficace.

Voici quelques informations sur la disposition des champs des structures. Elles sont sans importance lorsque les structures sont exploitées par le programme qui les crée, mais deviennent indispensables lorsque les structures sont partagées entre sous-programmes écrits dans divers langages ou lorsqu'elles servent à décrire les articles d'un fichier existant en dehors du programme.

Position. C garantit que les champs d'une structure sont alloués dans l'ordre où ils apparaissent dans la déclaration. Ainsi, avec la définition

```
struct modele {
    type1 a, b;
    type2 c, d, e;
    type3 f;
}
```

on trouve, dans le sens des adresses croissantes : `x.a`, `x.b`, `x.c`, `x.d`, `x.e` et enfin `x.f`[35].

Contiguïté. Les champs des structures subissent en général des contraintes d'alignement dépendant du système sous-jacent, qui engendrent des trous anonymes entre les champs. Il s'agit souvent des mêmes règles que celles qui pèsent sur les variables simples. Deux cas relativement fréquents sont les suivants :
- les champs de type `char` sont à n'importe quelle adresse, tous les autres champs devant commencer à une adresse paire ;
- les champs d'un type simple doivent commencer à une adresse multiple de leur taille (les `short` à une adresse paire, les `long` à une adresse multiple de quatre, etc.).

Initialisation. Une variable de type structure peut être initialisée au moment de sa déclaration, du moins dans le cas d'une variable globale. La syntaxe est la même que pour un tableau. Exemple :

[34] L'expression correcte de cette propriété consisterait à dire qu'une structure, comme une expression d'un type primitif, bénéficie de la *sémantique des valeurs*, par opposition à un tableaux qui, lui, est assujetti à la *sémantique des valeurs*.

[35] Alors que, avec la déclaration correspondante en Pascal (facile à imaginer), beaucoup de compilateurs alloueraient les champs dans l'ordre : `x.b`, `x.a`, `x.e`, `x.d`, `x.c` et enfin `x.f`.

```
struct fiche {
    int numero;
    char nom[32], prenom[32];
} u = { 1234, "DURAND", "Pierre" };
```

Comme pour les tableaux, si on indique moins de valeurs que la structure ne comporte de champs, alors les champs restants sont initialisés par des zéros. Donner plus de valeurs qu'il n'y a de champs constitue une erreur.

5.2.2 Unions

Tandis que les champs des structures se suivent sans se chevaucher, les champs d'une union commencent tous au même endroit et, donc, se superposent. Ainsi, une variable de type union peut contenir, à des moments différents, des objets de types et de tailles différents. La taille d'une union est celle du plus volumineux de ses champs.

La syntaxe de la déclaration et de l'utilisation d'une union est exactement la même que pour les structures, avec le mot `union` à la place du mot `struct`. Exemple (la `struct adresse_virt` est définie à la section 5.2.3) :

```
union mot_machine {
    unsigned long mot;
    char *ptr;
    struct adresse_virt adrv;
} mix;
```

La variable `mix` pourra être vue tantôt comme un entier long, tantôt comme l'adresse d'un caractère, tantôt comme la structure bizarre définie ci-après. Par exemple, le programme suivant teste si les deux bits hauts d'une certaine variable `m`, de type `unsigned long`, sont nuls :

```
mix.mot = m;
if (mix.adrv.tp == 0) etc.
```

Remarque 1. Dans ce genre de problèmes, les opérateurs binaires de bits (`&`, `|`, `^`) s'avèrent également très utiles. L'instruction précédente pourrait aussi s'écrire (sur une machine 32 bits, où `int` est synonyme de `long`) :

```
if (m & 0xC0000000 == 0) etc.
```

(l'écriture binaire du chiffre hexa `C` est `1100`). Bien sûr, les programmes de cette sorte n'étant pas portables, ces deux manières de faire peuvent être équivalentes sur une machine et ne pas l'être sur une autre.

Remarque 2. Les unions permettent de considérer un même objet comme possédant plusieurs types ; elles semblent donc faire double emploi avec l'opérateur de conversion de type. Ce n'est pas le cas. D'une part, cet opérateur ne supporte pas les structures, tandis qu'un champ d'une union peut en être une.

D'autre part, lorsque l'on change le type d'un objet au moyen de l'opérateur de changement de type, C *convertit* la donnée, afin que l'expression ait, sous le nouveau type, une valeur vraisemblable et utile. Par exemple, si `x` est une variable réelle (`float`), l'expression « `(int) x` » convertit `x` en l'entier dont la valeur est la plus voisine de la valeur qu'avait le réel `x`. Cela implique une transformation, éventuellement assez coûteuse, de la valeur de `x`. A l'opposé, lorsqu'on fait référence à un champ d'une union, la donnée ne subit aucune transformation, même si elle a été affectée en faisant référence à un autre champ. Par exemple, si l'on déclare

```
union {
    float reel;
    int entier;
} x;
```

et qu'ensuite on exécute les affectations :

```
x.reel = 12.5;
i = x.entier;
```

l'entier `i` obtenu sera celui dont la représentation binaire coïncide avec la représentation binaire du nombre flottant `12.5` ; cet entier n'a aucune raison d'être voisin de `12`.

5.2.3 Champs de bits

Le langage C permet aussi d'utiliser des structures et des unions dont les champs, tous ou seulement certains, sont faits d'un nombre quelconque de bits. De tels champs doivent être ainsi déclarés :

$$\left\{ \begin{array}{l} \texttt{unsigned}_{opt}\ \texttt{int}_{opt}\ \textit{identificateur} \\ \qquad\qquad\qquad \textit{rien} \end{array} \right\} \texttt{':'}\ \textit{constante-entiere}$$

Chaque constante précise le nombre de bits qu'occupe le champ. Le type doit obligatoirement être entier. Lorsque le type et le nom du champ sont absents, le nombre indiqué de bits est quand même réservé : on suppose qu'il s'agit de bits de « rembourrage » entre deux champs nommés.

Chacun de ces champs est logé immédiatement après le champ précédent, sauf si cela le met à cheval sur deux mots (de type `int`) ; dans ce cas, le champ est logé au début du mot suivant.

Exemple : la structure

```
struct adresse_virt {
    unsigned depl     :  9;
    unsigned numpagv  : 16;
                      :  5;
    unsigned tp       :  2;
};
```

découpe un mot de 32 bits en en quatre champs, comme indiqué[36] sur la figure 7.

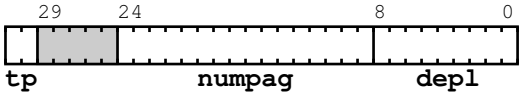

FIG. 7 – Champs de bits

ATTENTION. Dans tous les cas, les champs de bits sont très dépendants de l'implantation. Par exemple, certaines machines rangent les champs de gauche à droite, d'autres de droite à gauche. Ainsi, la portabilité des structures contenant des champs de bits n'est pas assurée ; cela limite leur usage à une certaine catégorie de programmes, comme ceux qui communiquent avec leur machine-hôte à un niveau très bas (noyau d'un système d'exploitation, gestion des périphériques, etc.). Ces programmes sont rarement destinés à être portés d'un système à un autre.

5.3 Enumérations

Les énumérations ne constituent pas un type structuré. Si elles figurent ici c'est uniquement parce que la syntaxe de leur déclaration possède des points communs avec celle des structures. Dans les cas simples, cette syntaxe est la suivante :

$$\texttt{enum}\ \textit{nom}_{opt} \left\{ \begin{array}{c} \texttt{'\{'}\ \textit{id-valeur}\ \texttt{','}\ \ldots\ \textit{id-valeur}\ \texttt{'\}'} \\ \textit{rien} \end{array} \right\} \left\{ \begin{array}{c} \textit{nom}\ \texttt{','}\ \ldots\ \textit{nom} \\ \textit{rien} \end{array} \right\}\ \texttt{';'}$$

où chaque id-valeur est à son tour de la forme

$$\textit{identificateur} \left\{ \begin{array}{c} \texttt{'='}\ \textit{expression-constante} \\ \textit{rien} \end{array} \right\}$$

EXEMPLE :

```
enum jourouvrable { lundi, mardi, mercredi, jeudi, vendredi } x, y;
```

A la suite d'une telle déclaration, le type `enum jourouvrable` est connu ; les variables `x` et `y` le possèdent. Ce type est formé d'une famille finie de symboles qui représentent des constantes entières : `lundi` (égale à 0), `mardi` (égale à 1), etc. Dans ce programme on pourra écrire :

```
enum jourouvrable a, b, c;
```

On aurait aussi bien pu introduire toutes ces variables par les déclarations

```
enum jourouvrable { lundi, mardi, mercredi, jeudi, vendredi };
...
enum jourouvrable x, y, a, b, c;
```

ou encore

```
enum { lundi, mardi, mercredi, jeudi, vendredi } x, y, a, b, c;
```

[36] Sur un système où un mot est fait de 32 bits (sinon un champ serait à cheval sur deux mots, ce qui est interdit).

mais cette dernière forme nous aurait empêché par la suite de déclarer aussi facilement d'autres variables du même type. Bien sûr, les énumérations peuvent se combiner avec la déclaration `typedef` (cf. section 5.4.3) : après la déclaration

```
typedef enum jourouvrable { lundi, mardi, mercredi, jeudi, vendredi } JOUROUVRABLE;
```

les expressions « `enum jourouvrable` » et « `JOUROUVRABLE` » sont équivalentes.

Les nombres entiers et les valeurs de tous les types énumérés sont totalement compatibles entre eux. Par conséquent, l'affectation d'un entier à une variable d'un type énuméré ou l'affectation réciproque ne provoquent en principe aucun avertissement de la part du compilateur . Ainsi, en C, les types énumérés ne sont qu'une deuxième manière de donner des noms à des nombres entiers (la première manière étant l'emploi de la directive `#define`, voir section 8.1.2).

Par défaut, la valeur de chacune de ces constantes est égale au rang de celle-ci dans l'énumération (`lundi` vaut 0, `mardi` vaut 1, etc.). On peut altérer cette séquence en associant explicitement des valeurs à certains éléments :

```
enum mois_en_r { janvier = 1, fevrier, mars, avril,
                 septembre = 9, octobre, novembre, decembre };
```

(`fevrier` vaut 2, `mars` vaut 3, `octobre` vaut 10, etc.)

5.4 Déclarateurs complexes

Jusqu'ici nous avons utilisé des formes simplifiées des déclarations. Tous les objets étaient soit simples, soit des tableaux, des fonctions ou des adresses d'objets simples. Il nous reste à voir comment déclarer des « tableaux de pointeurs » , des « tableaux d'adresses de fonctions », des « fonctions rendant des adresses de tableaux », etc. Autrement dit, nous devons expliquer la syntaxe des formules qui permettent de décrire des types de *complexité quelconque*.

La question concerne au premier chef les descripteurs de types, qui apparaissent dans trois sortes d'énoncés :
- les définitions de variables (globales et locales), les déclarations des paramètres formels des fonctions, les déclarations des variables externes et les déclarations des champs des structures et des unions ;
- les définitions des fonctions et les déclarations des fonctions externes ;
- les déclarations de types (`typedef`)

La même question réapparaît dans des constructions qui décrivent des types sans les appliquer à des identificateurs. Appelons cela des types *désincarnés*. Ils sont utilisés
- par l'opérateur de changement de type lorsque le type de destination n'est pas défini par un identificateur : « `(char *)` *expression* » ;
- comme argument de `sizeof` (rare) : « `sizeof(int [100])` » ;
- dans les prototypes des fonctions : « `strcpy(char *, char *) ;` »

5.4.1 Cas des déclarations

Un déclarateur complexe se compose de trois sortes d'ingrédients :
- un type « terminal » qui est un type de base (donc numérique), `enum`, `struct` ou `union` ou bien un type auquel on a donné un nom par une déclaration `typedef` ;
- un certain nombre d'occurrences des opérateurs `()`, `[]` et `*` qui représentent des procédés récursifs de construction de types complexes ;
- l'identificateur qu'il s'agit de déclarer.

Ces formules ont mauvaise réputation. Si elles sont faciles à lire et à traiter par le compilateur, elles s'avèrent difficiles à composer et à lire par le programmeur, car les symboles qui représentent les procédés récursifs de construction sont peu expressifs et se retrouvent *placés à l'envers* de ce qui aurait semblé naturel.

Pour cette raison nous allons donner un procédé mécanique qui, à partir d'une sorte de description « à l'endroit », facile à concevoir, fabrique la déclaration C souhaitée. Appelons *description naturelle* d'un type une expression de l'un des types suivants :
- la description correcte en C d'un type de base, d'un type `enum`, d'une `struct` ou `union` ou le nom d'un type baptisé par `typedef` ;
- la formule « tableau de T », où T est la description naturelle d'un type ;
- la formule « adresse d'un T » (ou « pointeur sur un T »), où T est la description naturelle d'un type ;
- la formule « fonction rendant un T », où T est la description naturelle d'un type.

Dans ces conditions, nous pouvons énoncer la « recette de cuisine » du tableau 3.

Pour obtenir la déclaration d'une variable ou d'une fonction :

1. Ecrivez :
   – à gauche, la description naturelle du type,
   – à droite, l'identificateur à déclarer (un seul).

2. Tant que l'expression à gauche n'est pas un type de base, `enum`, `struct` ou `union`, transformez les deux expressions de la manière suivante :

(a)	*fonction rendant un* exp_g	exp_d	$\rightarrow$	exp_g	`(`exp_d`)()`
(b)	*tableau de* exp_g	exp_d	$\rightarrow$	exp_g	`(`exp_d`)[]`
(c)	*pointeur sur* exp_g	exp_d	$\rightarrow$	exp_g	`*`exp_d

De plus, dans les règles *(a)* et *(b)*, si exp_d n'est pas de la forme `*`*exp*, alors les parenthèses autour de exp_d peuvent être omises.

TAB. 3 – Construction d'un déclarateur complexe

EXEMPLE 1. Soit à déclarer `tabPtr` comme un tableau (de dimension indéterminée) de pointeurs d'entiers. Appliquant les règles du tableau 3, nous écrirons successivement :

tableau de pointeur sur `int`	`tabPtr`
pointeur sur `int`	`(tabPtr)[]`
pointeur sur `int`	`tabPtr[]`
`int`	`*tabPtr[]`

La partie gauche de la ligne ci-dessus se réduisant à un type simple, cette ligne constitue la déclaration C cherchée, comme il faudra l'écrire dans un programme : « `int *tabPtr[] ;` ».

EXEMPLE 2. Pour la comparer à la précédente, cherchons à écrire maintenant la déclaration de `ptrTab`, un pointeur sur un tableau[37] d'entiers :

pointeur sur un tableau de `int`	`ptrTab`
tableau de `int`	`*ptrTab`
`int`	`(*ptrTab)[]`

Les parenthèses ci-dessus ne sont pas facultatives, car elles encadrent une expression qui commence par `*` (dit autrement, elles sont nécessaires, car `[]` a une priorité supérieure à `*`).

Les déclarations pour lesquelles le membre gauche est le même (à la fin des transformations) peuvent être regroupées. Ainsi, les déclarations de `tabPtr` et `ptrTab` peuvent être écrites ensemble :

```
int *tabPtr[], (*ptrTab)[];
```

En toute rigueur les règles *a* et *b* de ce tableau devraient être plus complexes, car la description naturelle du type d'une fonction inclut en général la spécification des types des arguments (le prototype de la fonction), et la description naturelle d'un type tableau comporte l'indication du nombre de composantes de celui-ci.

En réalité ces indications s'ajoutent aux règles données ici, presque sans modifier le processus que ces règles établissent : tout simplement, des informations annexes sont écrites à l'intérieur des crochets des tableaux et des parenthèses des fonctions.

EXEMPLE 3. Soit à déclarer une matrice de `10` lignes et `20` colonnes. Nos pouvons écrire successivement :

tableau de `10` *tableau de* `20 double`	`matrice`
tableau de `20 double`	`(matrice)[10]`
tableau de `20 double`	`matrice[10]`
`double`	`(matrice[10])[20]`
`double`	`matrice[10][20]`

EXEMPLE 4. Quand on écrit des programmes relevant du calcul numérique on a souvent besoin de variables de type « adresse d'une fonction $\mathbb{R} \rightarrow \mathbb{R}$ ». Voici comment écrire un tel type

[37] La notion de *pointeur sur un tableau* est assez académique, puisqu'une variable de type tableau représente déjà l'adresse de celui-ci (dans la notion d'« adresse d'un tableau » il y a une double indirection). En pratique on n'en a presque jamais besoin.

adresse de fonction (avec argument `double`*) rendant* `double`	`adrFon`
fonction (avec argument `double`*) rendant* `double`	`*adrFon`
`double`	`(*adrFon)(double)`

La déclaration cherchée est donc « `double (*adrFon)(double)` ».

EXEMPLE 5. Cet exemple abominable est extrait de la bibliothèque UNIX. La fonction `signal` renvoie l'adresse d'une « procédure », c'est-à-dire d'une fonction rendant `void`. Voici comment il faudrait la déclarer *en syntaxe originale*, c'est-à-dire sans spécifier les arguments de la fonction `signal` (cette déclaration est fournie dans le fichier `<signal.h>`) :

fonction rendant un pointeur sur fonction rendant `void`	`signal`
pointeur sur fonction rendant un `void`	`(signal)()`
pointeur sur fonction rendant un `void`	`signal()`
fonction rendant un `void`	`*signal()`
`void`	`(*signal())()`

En syntaxe ANSI il faut en plus donner les types des arguments des fonctions. On nous dit que les arguments de la fonction `signal` sont un `int` et un pointeur sur une fonction prenant un `int` et rendant un `void`. En outre, `signal` rend un pointeur vers une telle fonction. Nous savons déjà écrire le type « pointeur sur une fonction prenant un `int` et rendant un `void` » (ce n'est pas très différent de l'exemple précédent) :

```
void (*fon)(int);
```

Attaquons-nous à `signal` :

fonction (avec arguments `int sig` *et* `int (*fon)(int)`*)*	
rendant un pointeur sur fonction	
(avec argument `int`*) rendant* `void`	`signal`
pointeur sur fonction (avec argument `int`*) rendant* `void`	`(signal)(int sig, int (*fon)(int))`
pointeur sur fonction (avec argument `int`*) rendant* `void`	`signal(int sig, int (*fon)(int))`
fonction (avec argument `int s`*) rendant* `void`	`*signal(int sig, int (*fon)(int))`
`void`	`(*signal(int sig, int (*fon)(int)))(int)`

Finalement, la déclaration cherchée est

```
void (*signal(int sig, int (*fon)(int)))(int) ;
```

Comme annoncé, les expressions finales obtenues sont assez lourdes à digérer pour un humain. A leur décharge on peut tout de même dire qu'une fois écrites elles se révèlent bien utiles puisqu'elles font apparaître les types de leurs éléments terminaux comme ils seront employés dans la partie exécutable des programmes. Ainsi, à la vue de l'expression

```
float matrice[10][20];
```

on comprend sans effort supplémentaire qu'un élément de la matrice s'écrira `matrice[i][j]` et que cette formule représentera un objet de type `float`. De même, l'énoncé

```
void (*signal(sig, func))();
```

nous fera reconnaître immédiatement l'expression `(*signal(2,f))(n)` comme un appel correct de la « procédure » rendue par l'appel de `signal`.

5.4.2 Pointeurs et tableaux constants et volatils

Les qualifieurs `const` et `volatile` peuvent aussi apparaître dans les déclarateurs complexes. Plutôt que de rendre encore plus lourdes des explications qui le sont déjà beaucoup, contentons-nous de montrer des exemples significatifs. Les cas les plus utiles sont les suivants :

1. La formule :

   `const` *type* `*`*nom*

déclare *nom* comme un *pointeur vers un objet constant* ayant le type indiqué (la variable pointée, `*`*nom*, ne doit pas être modifiée).

2. La formule

   type `*const` *nom*

déclare *nom* comme un *pointeur constant vers un objet* ayant le type indiqué (c'est la variable *nom* qui ne doit pas être modifiée).

3. Plus simplement, la formule

`const` *type* *nom*`[`*expr*$_{opt}$`]`

déclare *nom* comme un tableau d'objets constants[38] (*nom*`[i]` ne doit pas être modifié).

Ainsi par exemple les déclarations

```
int e, *pe, *const pce = &e, te[100];
const int ec = 0, *pec, *const pcec = &ec, tec[100];
```

déclarent successivement :

- `e` : un entier
- `pe` : un pointeur vers un entier
- `pce` : un pointeur constant vers un entier (qui doit être obligatoirement initialisé, puisqu'il est constant)
- `te` : un tableau de 100 entiers
- `ec` : un entier constant
- `pec` : un pointeur vers un entier constant
- `pcec` : un pointeur constant vers un entier constant
- `tec` : un tableau de 100 entiers constants

Toute affectation avec pour membre gauche `pce`, `ec`, `*pec`, `pcec`, `*pcec` ou `tec[i]` est illégale.

En résumé, la seule vraie nouveauté introduite ici est la notion de *pointeur constant* et la syntaxe de sa déclaration « *type-pointé* `*const` *pointeur* ». Signalons enfin que toute cette explication reste valable en remplaçant le mot `const` et la notion d'objet constant par le mot `volatile` et le concept d' *objet volatil* (cf. section 1.5.7). On a donc la possibilité de déclarer des tableaux d'objets volatils, des pointeurs volatils vers des objets, volatils ou non, etc.

5.4.3 La déclaration `typedef`

Cette construction est l'homologue de la déclaration type du langage Pascal. Elle permet de donner un nom à un type dans le but d'alléger par la suite les expressions où il figure. Syntaxe :

`typedef` *déclaration*

Pour déclarer un nom de type, on fait suivre le mot réservé `typedef` d'une expression identique à une déclaration de variable, dans laquelle le rôle du nom de la variable est joué par le nom du type qu'on veut définir. Exemple : les déclarations

```
typedef float MATRICE[10][20];
...
MATRICE mat;
```

sont équivalentes à

```
float mat[10][20];
```

Voici un autre exemple :

```
typedef struct noeud {
    int info;
    struct noeud *fils_gauche, *fils_droit;
} NOEUD;
```

Cette expression déclare l'identificateur `NOEUD` comme le nom du type « la structure `noeud` » . Des variables `a`, `b`, `c` de ce type pourront par la suite être déclarées aussi bien par l'expression

```
struct noeud a, b, c;
```

que par

```
NOEUD a, b, c;
```

Remarquez l'utilisation différente d'un nom de structure (précédé du mot `struct`) et d'un nom de type. Par tradition, les noms des types définis à l'aide de la déclaration `typedef` sont écrits en majuscules.

L'ensemble des règles de syntaxe qui s'appliquent aux déclarations de variables s'appliquent également ici. Par exemple, on peut introduire plusieurs noms de types dans un seul énoncé `typedef`. Ainsi, à la suite de

[38] Notez qu'il n'y a pas besoin d'une notation spéciale pour déclarer un tableau constant d'objets (variables) puisque la signification ordinaire des tableaux est exactement celle-là.

```
typedef struct noeud {
    int info;
    struct noeud *fils_gauche, *fils_droit;
} NOEUD, *ARBRE;
```

les identificateurs `NOEUD` et `ARBRE` sont des synonymes pour « `struct noeud` » et « `struct noeud *` ».

Dans la construction des déclarateurs complexes, le nom d'un type qui a fait l'objet d'une déclaration `typedef` peut être utilisé comme type de base. Par exemple :

```
typedef float LIGNE[20];
typedef LIGNE MATRICE[10];
```

est une autre manière d'introduire le type MATRICE déjà vu. De même

```
typedef void PROCEDURE(int);
PROCEDURE *signal(int sig, PROCEDURE *func);
```

est une autre manière (nettement plus facile à lire, n'est-ce pas ?) de déclarer la fonction `signal` vue précédemment.

5.4.4 Cas des types désincarnés

Tout cela devient encore plus ésotérique lorsqu'on doit construire un type sans l'appliquer à un identificateur. Le principe est alors le suivant : *construisez la déclaration correcte d'un nom quelconque X, puis effacez X.*

De telles expressions sont utiles dans trois circonstances :

1. Pour construire un opérateur de changement de type. Exemple : la fonction `malloc` alloue un bloc de mémoire de taille donnée, dont elle renvoie l'adresse. Dans les bibliothèques du C original[39], cette fonction est déclarée comme rendant l'adresse d'un char, et l'utilisateur doit convertir le résultat de `malloc` dans le type qui l'intéresse. Supposons avoir besoin d'espace pour ranger un tableau de `N` réels en double précision. Déclarations[40] :

```
char *malloc(size_t);
...
double *p;
```

Utilisation :

```
p = (double *) malloc(N * sizeof(double));      /* DÉCONSEILLÉ EN C ANSI */
```

Le type désincarné « `double *` » apparaissant ci-dessus a été déduit d'une déclaration de pointeur, « `double *ptr` » en effaçant le nom déclaré, `ptr`.

Rappelons qu'en C ANSI l'expression ci-dessus s'écrit plus simplement (voyez à la section 2.2.12, notamment à la page 26, des indications et mises en garde à propos de l'opérateur de changement de type) :

```
p = malloc(N * sizeof(double));
```

2. Comme argument de l'opérateur `sizeof`. Essayons par exemple d'écrire autrement l'affectation précédente. Déclarons un tableau `T` de `N` doubles :

```
double T[N]
```

Supprimons `T`, il reste « `double [N]` ». Par conséquent, l'appel de la fonction `malloc` montré ci-dessus peut s'écrire aussi[41] :

```
p = (double *) malloc(sizeof(double [N]));
```

3. Dans les prototypes des fonctions, en C ANSI. Par exemple, la bibliothèque standard C comporte la fonction `qsort`, qui implémente une méthode de tri rapide d'un tableau (algorithme *Quicksort*) programmé en toute généralité. Les arguments de cette fonction sont le tableau à trier, le nombre de ses éléments, la taille de chacun et une fonction qui représente la relation d'ordre utilisée (le « critère de tri ») : elle reçoit deux adresses d'éléments du tableau et rend un nombre négatif, nul ou positif exprimant la comparaison.

La fonction `qsort` est donc ainsi déclarée dans le fichier `stdlib.h` :

```
void qsort(const void*, size_t, size_t, int (*)(const void*, const void*));
```

[39] Dans la bibliothèque ANSI (fichier `stdlib.h`) la fonction `malloc` est déclarée comme rendant une valeur de type « `void *` ». Ce type étant compatible avec tous les autres types pointeur, le problème exposé ici ne se pose pas.

[40] Le type `size_t` (défini dans `<stddef.h>`) représente un entier sans signe

[41] L'exemple montré ici, « `sizeof(double [N])` », est franchement ésotérique. La notation « `N * sizeof(double)` » est bien plus raisonnable.

NOTE. Ce n'est pas requis par la syntaxe mais, lorsqu'ils sont bien choisis, les noms des arguments sont une aide pour le programmeur. Exemple :

```
void qsort(const void *tableau, size_t nombre, size_t taille,
                        int (*compare)(const void *adr1, const void *adr2));
```

6 Pointeurs

6.1 Généralités

Un pointeur est une variable dont la valeur est l'adresse d'une cellule de la mémoire.

Les premiers langages évolués ne permettaient pas les manipulations de pointeurs, de telles opérations étant obligatoirement confinées dans les programmes écrits en assembleur. Le langage Pascal a introduit les pointeurs parmi les types de données disponibles dans des langages généralistes. Mais ils y sont présentés comme des entités fort mystérieuses, dont l'utilisateur n'est censé connaître ni la structure ni le comportement. Cela n'en facilite pas le maniement aux programmeurs débutants. Beaucoup des difficultés à comprendre les pointeurs proviennent plus du secret qui les entoure que de leur réelle difficulté d'emploi.

Un pointeur n'est pas un concept unique, sans ressemblance avec les autres types de données, car une adresse n'est en fin de compte rien d'autre qu'un nombre entier (l'indice d'un élément du tableau qu'est la mémoire de l'ordinateur). Un pointeur devrait donc supporter toutes les opérations qu'on peut faire subir à un nombre et qui gardent un sens lorsque ce nombre exprime un *rang*.

Ce qui est tout à fait spécifique d'un pointeur p est la possibilité de mentionner *la variable dont la valeur de* `p` *est l'adresse*. Un pointeur correctement initialisé est toujours le renvoi à une deuxième variable, *la variable pointée*, accessible à partir du contenu de `p`.

Dans tous les langages où ils existent, le principal intérêt des pointeurs réside dans la possibilité de réaliser des structures de données *récursives* (listes et arbres) : des structures dans lesquelles un champ est virtuellement aussi complexe que la structure elle-même. En plus de cet aspect, en C les pointeurs sont le moyen d'améliorer l'accès aux éléments des tableaux ordinaires, en incorporant au langage des techniques d'adressage indirect couramment utilisées en assembleur.

6.1.1 Déclaration et initialisation des pointeurs

Nous avons étudié à la section précédente la syntaxe des déclarations des pointeurs. Dans le cas le plus simple, cette syntaxe est

type `*`*variable*

Si par la suite on affecte à une telle variable l'adresse d'un certain objet, on fait référence à ce dernier par l'expression :

`*`*variable*

Par exemple, l'expression

```
char *p;
```

définit la variable `p` comme « pointeur vers un char ». Lorsqu'elle aura été correctement initialisée, elle contiendra l'adresse d'un caractère, auquel on pourra faire référence par l'expression

```
*p
```

Initialisation des pointeurs. Un pointeur contient en principe une adresse valide. Il existe trois manières sûres de donner une telle valeur à un pointeur :

1. Prendre l'adresse d'une variable existant par ailleurs. Exemple, déclaration :

   type `x, *p;`

Initialisation :

```
p = &x;
```

`p` contient maintenant une adresse valide, l'adresse de la variable `x`. Par la suite, les expressions `x` et `*p` désigneront le même objet (le contenu de `x`).

2. Provoquer l'allocation d'un nouvel espace, c'est-à-dire allouer dynamiquement une variable anonyme. Exemple, déclaration :

   type `*p;`

Initialisation :

```
p = malloc(sizeof(type));
```

Après cette instruction, une variable tout à fait analogue à la variable `x` de l'exemple précédent existe. Comme dans l'exemple précédent, `*p` en désigne le contenu. La différence est qu'ici elle n'a pas de nom propre :

si la valeur de `p` venait à être altérée (sans avoir été auparavant sauvegardée dans un autre pointeur), la variable pointée deviendrait inaccessible et serait définitivement perdue.

3. Obtenir une valeur par des calculs légitimes sur des pointeurs. Par exemple, comme on le verra bientôt, avec les déclarations

```
type t[10], *p, *q;
```

suivies de l'initialisation

```
p = &t[0];
```
(ou tout simplement `p = t;`)

l'expression

```
q = p + 3;
```

affecte à `q` l'adresse du quatrième élément du tableau `t`, c'est-à-dire `&t[3]`.

Il existe aussi une manière risquée, en tout cas non portable, de donner une valeur à un pointeur : lui affecter une constante ou une expression numérique, dans le style de :

```
p = (struct machin *) 0x2A3E;
```

De telles expressions, indispensables dans certaines fonctions de bas niveau (contrôle d'organes matériels, etc.), rendent non portables les programmes où elles figurent. Elles doivent donc être évitées chaque fois que cela est possible.

A propos de la taille des pointeurs et celle des entiers. Une expression comme l'affectation ci-dessus soulève un autre problème : existe-t-il une relation entre la taille d'un nombre entier (types `int` et `unsigned int`) et celle d'un pointeur ? Le langage ne prescrit rien à ce sujet ; il est donc parfois erroné et *toujours dangereux de supposer qu'un entier puisse contenir un pointeur.* Cette question peut sembler très secondaire, l'utilisation de variables entières pour conserver ou transmettre des pointeurs paraissant bien marginale. Or c'est ce qu'on fait, peut-être sans s'en rendre compte, lorsqu'on appelle une fonction qui renvoie un pointeur sans avoir préalablement déclaré la fonction. Par exemple, supposons que sans avoir déclaré la fonction `malloc` (ni inclus le fichier `<stdlib.h>`) on écrive l'affectation

```
p = (struct machin *) malloc(sizeof(struct machin));
```

Le compilateur rencontrant `malloc` pour la première fois, il postule que cette fonction renvoie un `int` et insère à cet endroit le code qui convertit un entier en un pointeur. Cette conversion peut fournir un résultat tout à fait erroné. Notamment, sur un système où les entiers occupent deux octets et les pointeurs quatre, les deux octets hauts du résultat de `malloc` seront purement et simplement remplacés par zéro. La question est d'autant plus vicieuse que le programme ci-dessus se comportera correctement sur tous les systèmes dans lesquels les entiers et les pointeurs ont la même taille.

6.1.2 Les pointeurs génériques et le pointeur `NULL`

Certaines fonctions de la bibliothèque, comme `malloc`, rendent comme résultat une « adresse quelconque », à charge pour l'utilisateur de la convertir en l'adresse spécifique qui lui convient. La question est : comment déclarer de tels pointeurs peu typés ou pas typés du tout ?

Le C original n'ayant rien prévu à ce sujet, la tradition s'était créée de prendre le type `char *` pour « pointeur vers n'importe quoi ». En effet, sur beaucoup de systèmes les adresses des objets de taille non unitaire supportent des contraintes d'alignement (les `short` aux adresse paires, les `long` aux adresses multiples de quatre, etc.), alors que les adresses des caractères sont les seules à ne subir aucune restriction.

Le C ANSI introduit un type spécifique pour représenter de telles adresses génériques : le type `void *`. Ce type est compatible avec tout autre type pointeur : n'importe quel pointeur peut être affecté à une variable de type `void *` et réciproquement[42]. Par exemple, le fichier standard `<stdlib.h>` contient la déclaration suivante de la fonction `malloc` :

```
void *malloc(size_t size);
```

Appel :

```
p = malloc(sizeof(struct machin));
```

Note. A la section 2.2.12 (page 26) nous expliquons pourquoi l'emploi d'un opérateur de changement de type en association avec `malloc` est dangereux et déconseillé, en C ANSI.

[42] Avec une protection supplémentaire : le type `void` étant incompatible avec tous les autres types, l'objet pointé par une variable déclarée de type `void *` ne pourra pas être utilisé tant que la variable n'aura pas été convertie en un type pointeur précis.

Le pointeur NULL. Une des principales applications des pointeurs est la mise en œuvre de structures récursives de taille variable : listes chaînées de longueur inconnue, arbres de profondeur quelconque, etc. Pour réaliser ces structures il est nécessaire de posséder une valeur de pointeur, distincte de toute adresse valide, représentant la fin de liste, la structure vide, etc.

C garantit qu'aucun objet valide ne se trouve à l'adresse 0 et l'entier 0 est compatible avec tout type pointeur. Cette valeur peut donc être utilisée comme la valeur conventionnelle du « pointeur qui ne pointe vers rien » ; c'est l'équivalent en C de la constante *nil* de Pascal. En pratique on utilise plutôt la constante `NULL` qui est définie dans le fichier `<stddef.h>` :

```
#define NULL ((void *) 0)
```

6.2 Les pointeurs et les tableaux

6.2.1 Arithmétique des adresses, indirection et indexation

Arithmétique. Certaines des opérations arithmétiques définies sur les nombres gardent un sens lorsque l'un des opérandes ou les deux sont des adresses :
- l'addition et la soustraction d'un nombre entier à une adresse ;
- la soustraction de deux adresses (de types rigoureusement identiques).

L'idée de base de ces extensions est la suivante : si *exp* exprime l'adresse d'un certain objet O_1 alors $exp{+}1$ exprime l'adresse de l'objet de même type que O_1 qui se trouverait dans la mémoire immédiatement après O_1 et $exp{-}1$ l'adresse de celui qui se trouverait immédiatement avant.

De manière analogue, si exp_1 et exp_2 sont les adresses respectives de deux objets O_1 et O_2 de même type et contigus (O_2 après O_1), il semble naturel de donner à $exp_2{-}exp_1$ la valeur 1.

L'arithmétique des adresses découle de ces remarques. D'un point de vue technique, ajouter 1 à un pointeur c'est le considérer momentanément comme un entier et lui ajouter une fois la taille de l'objet pointé. On peut aussi dire cela de la manière suivante : exp_1 et exp_2 étant deux expressions de type « adresse d'un objet de type T » et n une expression entière, nous avons :

$$exp_1 + n \equiv \texttt{(}T\texttt{ *) ((unsigned long) }exp_1 + n\times\texttt{sizeof(}T\texttt{))}$$

et

$$exp_2 - exp_1 \equiv \frac{\texttt{(unsigned long)}\,exp_2 - \texttt{(unsigned long)}\,exp_1}{\texttt{sizeof(}T\texttt{)}}$$

L'indexation. Il découle des explications précédentes que l'arithmétique des adresses ne se justifie que si on peut considérer un pointeur p comme l'adresse d'un élément $t\texttt{[}i_0\texttt{]}$ d'un tableau, éventuellement fictif. On peut alors interpréter $p+i$ comme l'adresse de $t\texttt{[}i_0+i\texttt{]}$ et $q-p$ comme l'écart entre les indices correspondant à ces deux éléments. Les pointeurs avec leur arithmétique et l'accès aux éléments des tableaux sont donc des concepts très voisins. En réalité il y a bien plus qu'un simple lien de voisinage entre ces deux concepts, car l'un est la définition de l'autre :

Soit *exp* une expression quelconque de type adresse et *ind* une expression entière. *Par définition*, l'expression :

$$\texttt{*(}exp\texttt{ + }ind\texttt{)}$$

se note :

$$exp\texttt{[}ind\texttt{]}$$

On en déduit que si on a l'une ou l'autre des déclarations

```
type t[100];
```

ou

```
type *t;
```

alors dans un cas comme dans l'autre on peut écrire indifféremment

`t[0]` *ou* `*t`

ou

`t[i]` *ou* `*(t + i)`

Bien entendu, un programme est plus facile à lire si on montre une certaine constance dans le choix des notations.

Exemple. Le programme suivant parcourt un tableau en utilisant un pointeur :

```
char s[10] = "Hello";

main() {
    char *p = s;
    while (*p != '\0')
        fputc(*p++, stdout);
    }
```

REMARQUE 1. La similitude entre un tableau et un pointeur est grande, notamment dans la partie exécutable des fonctions. Mais il subsiste des différences, surtout dans les déclarations. Il ne faut pas oublier que les déclarations

```
int *p, t[100];
```

introduisent `p` comme une variable pointeur et `t` comme une constante pointeur. Ainsi, les expressions `p = t` et `p++` sont légitims, mais `t = p` et `t++` sont illégales, car elles tentent de modifier une « non variable » (un nom de tableau n'est pas une lvalue).

REMARQUE 2. Dans ce même esprit, rappelons la différence qu'il peut y avoir entre les deux déclarations suivantes :

```
char s[] = "Bonsoir";
```

et

```
char *t = "Bonsoir";
```

`s` est (l'adresse d') un tableau de caractères (figure 8)

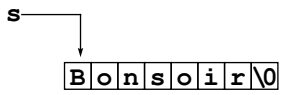

FIG. 8 – Chaîne de caractères « variable »

tandis que t est l'adresse d'un (tableau de) caractère(s) (figure 9)

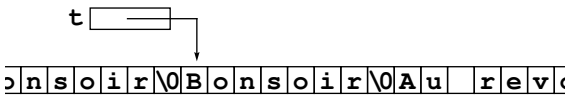

FIG. 9 – Constante chaîne de caractères

On pourra référencer ces caractères indifféremment par `s[i]`, `*(s + i)`, `t[i]` ou `*(t + i)`. Cependant, dans le premier cas le compilateur a alloué un vrai tableau de 8 caractères, et y a copié la chaîne donnée ; dans le second cas, la chaîne a été rangée avec les autres constantes littérales et le compilateur n'a alloué qu'une variable de type pointeur dans laquelle il a rangé l'adresse de cette constante. Par suite, `s` n'est pas une variable mais `*s`, `s[i]` et `*(s + i)` en sont, tandis que `t` est une variable mais `*t`, `t[i]` et `*(t + i)` n'en sont pas[43]. A ce propos voir aussi la section 5.1.3.

L'arithmétique des adresses permet d'améliorer l'efficacité de certains traitements de tableaux[44], en mettant à profit le fait que l'indirection est plus rapide que l'indexation. Autrement dit, l'accès à une donnée à travers une expression de la forme

```
*p
```

est réputée (légèrement) plus rapide que l'accès à la même donnée à travers l'expression

`*(q + i)` (ou, ce qui est la même chose, `q[i]`)

Développons un exemple classique : la fonction `strcpy(dest, srce)` copie la chaîne de caractères `srce` dans l'espace pointé par `dest`. Première forme :

[43] C'est uniquement d'un point de vue technique que ces expressions ne sont pas des variables car du point de vue de la syntaxe elles sont des *lvalue* tout à fait légitimes et le compilateur ne nous interdira pas de les faire apparaître à gauche d'une affectation.

[44] Bien sûr, il s'agit de petites économies dont le bénéfice ne se fait sentir que lorsqu'elles s'appliquent à des parties des programmes qui sont vraiment critiques, c'est-à-dire exécutées de manière très fréquente.

```
char *strcpy(char dest[], char srce[]) {
    register int i = 0;
    while ((dest[i] = srce[i]) != '\0')
        i++;
    return dest;
}
```

Deuxième forme :

```
char *strcpy(char *dest, char *srce) {
    register char *d = dest, *s = srce;
    while ((*d++ = *s++) != '\0')
        ;
    return dest;
}
```

La deuxième forme est plus efficace, car on y a supprimé l'indexation. Pour s'en convaincre, il suffit de considérer que la boucle qui figure dans la première forme équivaut à :

```
while ((*(dest + i) = *(srce + i)) != '\0')
    i++;
```

6.2.2 Tableaux dynamiques

L'équivalence entre les tableaux et les pointeurs permet de réaliser des *tableaux dynamiques*, c'est-à-dire des tableaux dont la taille n'est pas connue au moment de la compilation mais uniquement lors de l'exécution, lorsque le tableau commence à exister. Pour cela il suffit de

- remplacer la déclaration du tableau par celle d'un pointeur ;
- allouer l'espace à l'exécution, avant toute utilisation du tableau, par un appel de `malloc` ;
- dans le cas d'un tableau local, libérer l'espace à la fin de l'utilisation.

Pour tout le reste, l'utilisation de ces tableaux dynamiques est identique à celle des tableaux normaux (ce qui, pour le programmeur, est une très bonne nouvelle !). Exemple : la fonction ci-dessous calcule dans un tableau la N^{eme} ligne du triangle de Pascal[45] et s'en sert dans des calculs auxquels nous ne nous intéressons pas ici. On notera qu'une seule ligne diffère entre la version à tableau statique et celle avec tableau dynamique.

Programmation avec un tableau ordinaire (statique) :

```
void Blaise(int N) {
    int n, p, i;
    int tab[N_MAX + 1];        /* N_MAX est une constante connue à la compilation */

    for (n = 0; n <= N; n++) {
        tab[0] = tab[n] = 1;
        for (p = n - 1; p > 0; p--)
            tab[p] += + tab[p - 1];
                               /* exploitation de tab ; par exemple, affichage : */
        for (i = 0; i <= n; i++)
            printf("%5d", tab[i]);
        printf("\n");
    }
}
```

Programmation avec un tableau dynamique :

```
void Blaise(int N) {
    int n, p, i;
    int *tab;                  /* ici, pas besoin de constante connue à la compilation */

    tab = malloc((N + 1) * sizeof(int));
    if (tab == NULL)
        return;
```

[45] Rappelons que les coefficients de la n^{eme} ligne du triangle de Pascal sont définis récursivement à partir de ceux de la $n-1^{eme}$ ligne par : $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$.

```
    for (n = 0; n <= N; n++) {
        tab[0] = tab[n] = 1;
        for (p = n - 1; p > 0; p--)
            tab[p] += + tab[p - 1];
                                /* exploitation de tab ; par exemple, affichage : */
        for (i = 0; i <= n; i++)
            printf("%5d", tab[i]);
        printf("\n");
    }
    free(tab);                  /* à ne pas oublier (tab est une variable locale) */
}
```

REMARQUE. L'emploi de tableaux dynamiques à plusieurs indices est beaucoup plus compliqué que celui des tableaux à un indice comme ci-dessus, car il faut gérer soi-même les calculs d'indices, comme le montre la section suivante.

6.2.3 Tableaux multidimensionnels

RÉFLEXIONS SUR LA DOUBLE INDEXATION. Supposons que nous ayons à écrire un programme qui opère sur des matrices à NL lignes et NC colonnes. La déclaration d'une telle matrice est facile à construire (cf. section 5.4.1) :

tableau de `NL` tableau de `NC float`	`m1`
tableau de `NC float`	`(m1)[NL]` ou `m1[NL]`
`float`	`(m1[NL])[NC]` ou `m1[NL][NC]`

Un élément particulier de cette matrice sera noté `m1[i][j]`. D'après ce que nous savons déjà, on peut l'interpréter de la manière suivante[46] :

```
m1[i][j]
    = *(m1[i] + j)
    = *(float *)((unsigned int) m1[i] + j × sizeof(float))
```

Le problème est : avec la déclaration que nous avons donnée, que signifie `m1[i]` ? Il est intéressant de constater que cette expression n'est pas une lvalue, puisqu'elle est de type « tableau de... », mais que, à part cela, elle est traitée comme un élément d'un tableau ordinaire. Autrement dit, si on supposait avoir la déclaration

```
typedef float LIGNE[NC];        /* LIGNE = tableau de NC float  */
```

alors `m1[i]` serait une expression de type `LIGNE` valant :

```
m1[i]
    = *(m1 + i)
    = *(LIGNE *)((unsigned) m1 + i × sizeof(LIGNE))
    = *(LIGNE *)((unsigned) m1 + i × NC × sizeof(float))
```

En oubliant pour un instant les types des pointeurs, il nous reste

```
m1[i][j] = *(m1 + (i × NC + j) × sizeof(float))
```

Nous retrouvons l'expression classique `i` × `NC` + `j` caractérisant l'accès à un élément $m_{i,j}$ d'une matrice $NL \times NC$. Cette expression traduit la disposition « par lignes » des tableaux rectangulaires, comme le montre la figure 10.

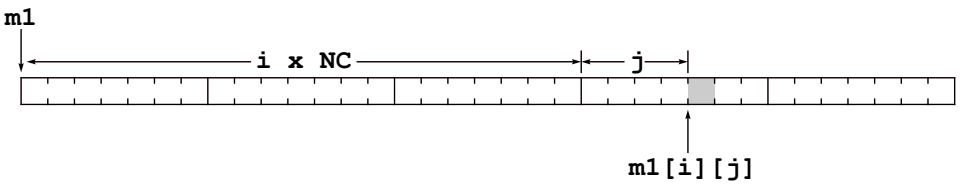

FIG. 10 – Tableau bidimensionnel statique

Etudions une autre manière d'accéder aux éléments de *cette même matrice*. Soient les déclarations

[46] Pour alléger les formules, nous supposons ici que la taille d'un `int`, et donc celle d'un `unsigned`, sont égales à celle d'un pointeur.

```
float m1[NL][NC], *m2[NL];
```

La deuxième se lit, compte tenu des priorités : `float *(m2[NL])`. La variable `m2` est donc déclarée comme un tableau de `NL` *pointeurs* vers des nombres flottants. Pour réaliser la matrice dont nous avons besoin, il nous suffit d'initialiser correctement ces pointeurs : chacun sera l'adresse d'une ligne de la matrice précédente. Nous aurons la configuration de la figure 11.

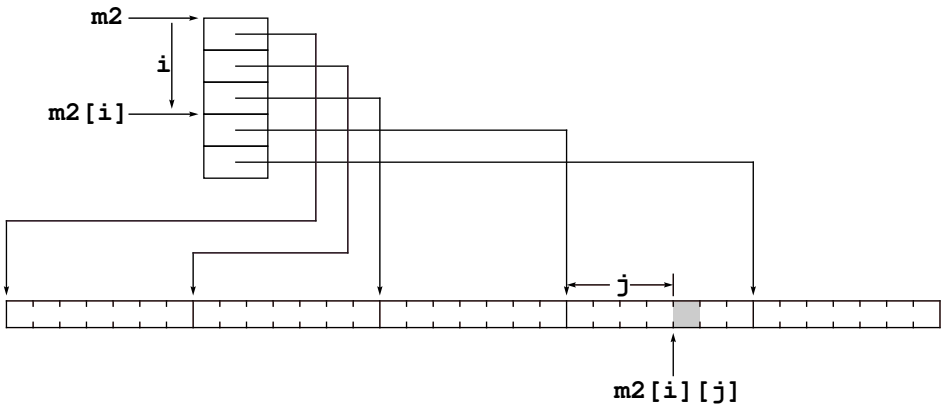

FIG. 11 – Tableau bidimensionnel réalisé avec un tableau de pointeurs

Il est remarquable qu'un élément de la nouvelle « matrice » ainsi déclarée se note encore

```
m2[i][j]
```

mais maintenant, cette expression se traduira (en continuant à négliger les conversions des types pointeurs) par :

```
m2[i][j]
    = *(m2[i] + j × sizeof(float))
    = *(*(m2 + i × sizeof(float *)) + j × sizeof(float))
```

Les multiplications par `sizeof(float)` et `sizeof(float *)` sont réalisées de manière très rapide par les calculateurs (ce sont des multiplications par des puissances de 2), nous pouvons les ignorer. Il reste que nous avons remplacé « `m1 + i × NC + j` » par « `*(m2 + i) + j` ». Nous avons donc bien éliminé une « vraie » multiplication : notre deuxième manière est plus efficace. En contrepartie nous occupons un peu plus d'espace (les `NL` pointeurs supplémentaires).

De plus, il nous faut initialiser le tableau de pointeurs `m2`. La matrice `m1` existant par ailleurs, cela se fait ici par une expression d'une étonnante simplicité :

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

6.2.4 Tableaux multidimensionnels dynamiques

Les considérations précédentes montrent pourquoi dans la réalisation d'une matrice dynamique on devra renoncer au mécanisme de double indexation ordinaire. En effet, `NC` doit être connu à la compilation pour que l'expression `i × NC + j` ait un sens.

Or nous sommes maintenant parfaitement équipés pour réaliser une telle matrice. Supposons que la fonction `void unCalcul(int nl, int nc)` implémente un algorithme qui requiert une matrice locale à `nl` lignes et `nc` colonnes.

Version statique :

```
#define NL 20
#define NC 30
...

void unCalcul(int nl, int nc) {
    double mat[NL][NC];     /* en espérant que nl ≤ NL et nc ≤ NC */
    int i, j;
```



```
                            /* utilisation de la matrice mat. Par exemple : */
    for (i = 0; i < nl; i++)
        for (j = 0; j < nc; j++)
            mat[i][j] = 0;
    etc.
}
```

Version dynamique (par souci de clarté nous y avons omis la détection des échecs de `malloc`) :

```
void unCalcul(int nl, int nc) {
    double **mat;
    int i, j;
                            /* initialisation des pointeurs */
    mat = malloc(nl * sizeof(double *));
    for (i = 0; i < nl; i++)
        mat[i] = malloc(nc * sizeof(double));

                            /* utilisation de la matrice mat. Par exemple : */
    for (i = 0; i < nl; i++)
        for (j = 0; j < nc; j++)
            mat[i][j] = 0;
    etc.

                            /* libération (indispensable dans le cas d'une variablelocale) */
    for (i = 0; i < nl; i++)
        free(mat[i]);
    free(mat);
}
```

Dans cette manière de procéder, les lignes de la matrice sont allouées à l'occasion de `nl` appels distincts de `malloc`. La matrice est réalisée par des morceaux de mémoire éparpillée, comme le montre la figure 12.

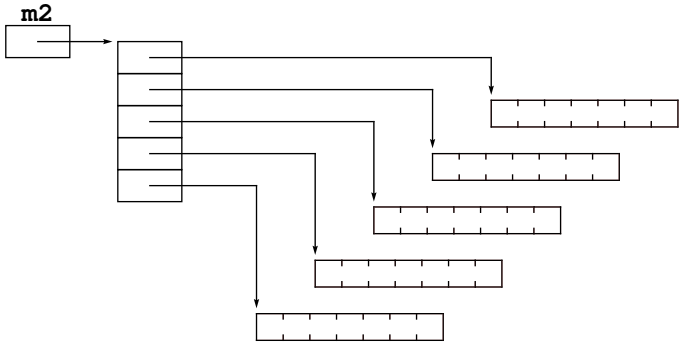

FIG. 12 – Matrice dynamique (lignes allouées séparément)

Une légère variante consiste à allouer d'un seul coup toute la mémoire nécessaire aux lignes de la matrice. La structure de la matrice ressemble alors à celle de la figure 11 :

```
int unCalcul(int nl, int nc) {
    double **mat, *espace;
    int i, j;
                            /* initialisation des pointeurs */
    mat = malloc(nl * sizeof(double *));
    if (mat == NULL)
        return 0;           /* échec */
```

```
    espace = malloc(nl * nc * sizeof(double));
    if (espace == NULL) {
        free(mat);
        return 0;           /* échec */
    }
    for (i = 0; i < nl; i++)
        mat[i] = espace + i * nc;

                            /* utilisation de la matrice mat. Par exemple : */
    for (i = 0; i < nl; i++)
        for (j = 0; j < nc; j++)
            mat[i][j] = 0;
    etc.

                            /* libération (indispensable dans le cas d'une variablelocale) */
    free(espace);
    free(mat);
    return 1;               /* succès */
}
```

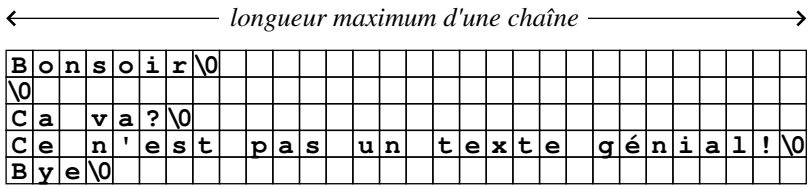

FIG. 13 – Matrice de caractères

6.2.5 Tableaux de chaînes de caractères

Le remplacement d'un tableau à deux indices par un tableau de pointeurs se révèle encore plus utile dans le traitement des tableaux de chaînes de caractères. Ici, cette technique entraîne une économie de place substantielle, puisqu'elle permet de remplacer un tableau rectangulaire dont la largeur est déterminée par la plus grande longueur possible d'une chaîne (avec un espace inoccupé très important) comme celui de la figure 13, par un espace linéaire dans lequel chaque chaîne n'occupe que la place qui lui est nécessaire, comme le montre la figure 14.

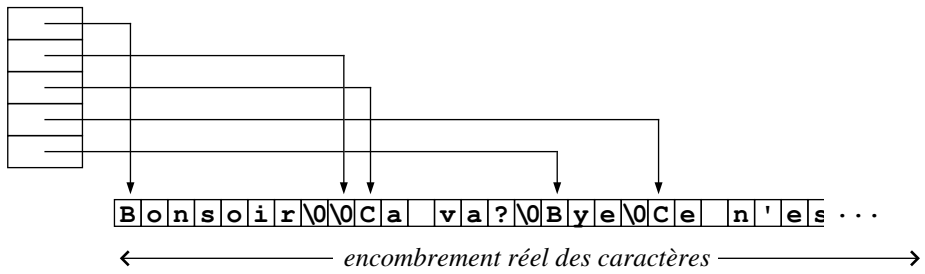

FIG. 14 – Tableau de chaînes de caractères

Nous allons illustrer cette méthodologie par un exemple très simple mais tout à fait typique. Notre programme lit des lignes de caractères au terminal et les range dans une table, jusqu'à la rencontre de la fin de fichier ou la lecture d'une ligne vide. Ensuite, il ordonne les lignes lues ; finalement il affiche les lignes ordonnées.

Les lignes lues sont rangées les unes à la suite des autres dans un tableau unique, nommé `espace`, dont le pointeur `libre` indique la première place disponible. La table des mots, notée mots, est une table de renvois dans espace. La figure 14 illustre ces structures de données. Notez que le tri se fait en ordonnant la table des pointeurs ; les caractères eux-mêmes ne sont nullement déplacés.

Les fonctions `gets` (lecture de chaîne), `puts` (écriture de chaîne), `strcmp` (comparaison de chaînes) et `strlen` (longueur d'une chaîne) appartiennent à la bibliothèque standard et sont expliquées dans les sections suivantes.

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

#define MAXMOTS  100
#define MAXCARS 3000

char espace[MAXCARS];
char *libre = espace;
char *mots[MAXMOTS];
int  nbr_mots = 0;

void tri(char *t[], int n) {
    for (;;) {
        int i, ok = 1;
        for (i = 1; i < n; i++)
            if (strcmp(t[i - 1], t[i]) > 0) {
                char *w = t[i - 1];
                t[i - 1] = t[i];
                t[i] = w;
                ok = 0;
            }
        if (ok)
            return;
        n--;
    }
}

void main(void) {
    int i;
    while(gets(libre) != NULL && *libre != '\0') {
        mots[nbr_mots++] = libre;
        libre += strlen(libre) + 1;
    }
    tri(mots, nbr_mots);
    for (i = 0; i < nbr_mots; i++)
        printf("%s\n", mots[i]);
}
```

Remarque. On peut signaler qu'une structure tout à fait analogue à notre table mots est créée par le compilateur lorsqu'on initialise un tableau de chaînes de caractères, comme dans :

```
char *messages[] = {
        "Fichier inexistant",
        "Volume ou repertoire inexistant",
        "Nom de fichier incorrect",
        "Erreur physique d'entree-sortie",
        "Autre erreur" };
```

La seule différence entre notre tableau `mots` et le tableau `messages` que créerait le compilateur par suite de la déclaration ci-dessus est la suivante : les éléments de `mots` sont des pointeurs vers un espace modifiable faisant partie de la mémoire attribuée pour les variables de notre programme, tandis que `messages` est fait de pointeurs vers des cellules de la « zone de constantes », espace immodifiable alloué et garni une fois pour toutes par le compilateur.

6.2.6 Tableaux multidimensionnels formels

Nous avons vu que si un argument formel d'une fonction ou une variable externe est un tableau à un seul indice, alors l'indication du nombre de ses éléments est sans utilité puisqu'il n'y a pas d'allocation d'espace. Mais que se passe-t-il pour les tableaux à plusieurs indices ? Si un tableau a été déclaré

T `table[`N_1`][`N_2`] ... [`N_{k-1}`][`N_k`] ;`

(N_1, N_2, etc. sont des constantes) alors un accès comme

`table`[i_1]`[`i_2`]` ... `[`i_{k-1}`]`[i_k]

se traduit par

`*( (`T`*)table` + $i_1 \times N_2 \times N_3 \times \cdots \times N_k + \cdots + i_{k-1} \times N_k + i_k$ `)`

Qu'il y ait ou non allocation d'espace, il faut que ce calcul puisse être effectué. D'où la règle générale : lorsqu'un argument formel d'une fonction est un tableau à plusieurs indices, la connaissance de la première dimension déclarée est sans utilité mais les autres dimensions déclarées doivent être connues du compilateur.

Pour un tableau à deux indices, cela donne : il est inutile d'indiquer combien il y a de lignes, mais il faut absolument que, en compilant la fonction, le compilateur connaisse la taille déclarée pour chaque ligne. Par exemple, la fonction suivante effectue la multiplication par un scalaire `k` de la matrice `m` déclarée `NL` × `NC` (deux constantes connues dans ce fichier) :

```
void k_fois(double k, double m[][NC]) {
    int i, j;
    for (i = 0; i < NL; i++)
        for (j = 0; j < NC; j++)
            m[i][j] *= k;
}
```

Remarque. Bien entendu, une autre solution aurait consisté à faire soi-même les calculs d'indice :

```
void k_fois(double k, void *m) {
    int i, j;
    for (i = 0; i < NL; i++)
        for (j = 0; j < NC; j++)
            *((double *) m + i * NC + j) *= k;
}
```

Tout ce qu'on y gagne est un programme bien plus difficile à lire !

6.2.7 Tableaux non nécessairement indexés à partir de zéro

En principe, les tableaux de C sont indexés à partir de zéro : sans qu'il soit nécessaire de le spécifier, un tableau T déclaré de taille N est fait des éléments $T_0, T_1 \ldots T_{N-1}$. Nous avons vu que, dans la grande majorité des applications, cette convention est commode et fiable.

Il existe cependant des situations dans lesquelles on souhaiterait pouvoir noter les éléments d'un tableau $T_1, T_2 \ldots T_N$ ou, plus généralement, $T_a, T_{a+1} \ldots T_b$, a et b étant deux entiers quelconques vérifiant $a \leq b$. Nous allons voir que l'allocation dynamique fournit un moyen simple et pratique[47] pour utiliser de tels tableaux.

Cas des tableaux à un indice. La fonction `dTable`[48] crée un tableau d'éléments de type `double` dont les indices sont les entiers `iMin`, `iMin`+1, ... `iMax` :

```
double *dTable(int iMin, int iMax) {
    double *res = malloc((iMax - iMin + 1) * sizeof(double));
    if (res == NULL) {
        erreurTable = 1;
        return NULL;
    }
    erreurTable = 0;
    return res - iMin;
}
```

(Si on avait besoin de tableaux de `float`, de `int`, etc. on écrirait des fonctions analogues `fTable`, `iTable`, etc.). La variable globale `erreurTable` est rendue nécessaire par le fait qu'un résultat `NULL` ne suffit pas ici à caractériser l'échec de l'allocation ; en effet, `res - iMin` peut être nul alors que l'allocation a réussi et `res` ≠ `NULL`. La figure 15 illustre le résultat rendu par la fonction précédente lors d'un appel comme `dTable(5, 10)`.

[47] Nous expliquons ici une technique massivement employée dans la bibliothèque de calcul scientifique *Numerical Recipes* (© Numerical Recipes Software) basée sur le livre *Numerical Recipes in C* (W. Press, S. Teukolsky, W. Vetterling et B. Flannery, Cambridge University Press, 1992).

[48] D'obscures contraintes techniques, ayant perdu tout intérêt de nos jours, font que cette technique peut ne pas fonctionner dans les implantations de C sur d'anciennes plate-formes, comme les PC à base de processeurs Intel antérieurs au 80286 (un lointain ancêtre du Pentium), dans lesquelles l'arithmétique des adresses subit des limitations drastiques.

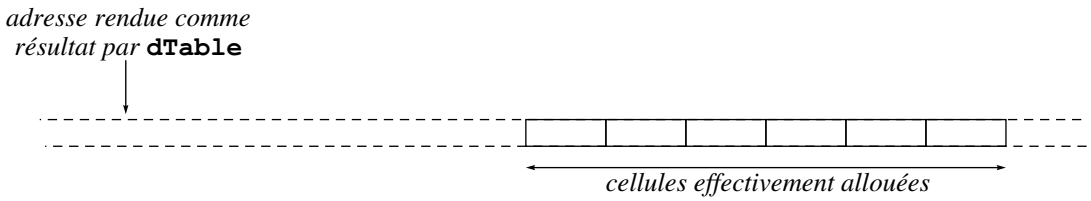

FIG. 15 – Adresse rendue par la fonction `dTable`

0n accède aux éléments d'un tel tableau à l'aide d'un indice compris entre les valeurs indiquées mais, à part cela, de manière tout à fait habituelle ; en particulier, sans aucune perte d'efficacité. Exemple d'utilisation :

```
...
double *t = dTable(5, 10);
...
for (i = 5; i <= 10; i++)
    t[i] = 1000 + i;
...
for (i = 5; i <= 10; i++)
    printf("%.1f\n", t[i]);
...
```

ATTENTION. Un tableau créé par un appel de la fonction `dTable` est dynamiquement alloué, cependant sa libération est un peu plus complexe que pour les tableaux dynamiques ordinaires. En particulier, à la suite du programme ci-dessus, un appel comme

```
free(t);
```

générera certainement une erreur à l'exécution. Il aurait fallu définir une structure plus complexe qu'un simple tableau, incluant soit la valeur de `iMin`, soit celle de `res` (actuellement des variables locales de la fonction `dTable`) pour gérer correctement la libération ultérieure d'un tel tableau.

CAS DES TABLEAUX À PLUSIEURS INDICES. Les remarques précédentes, avec celles faites à la section 6.2.4, suggèrent une manière de mettre en œuvre des matrices dont les lignes et les colonnes ne sont pas indexées à partir de zéro.

La fonction `dMatrice` crée une matrice dont les lignes sont indexées par les entiers `lMin`, `lMin`+1 ... `lMax` et les colonnes sont indexées par les entiers `cMin`, `cMin`+1 ... `cMax` :

```
double **dMatrice(int lMin, int lMax, int cMin, int cMax) {
    int nbLin, nbCol, i;
    double *espace, **lignes, *p;

    nbLin = lMax - lMin + 1;
    nbCol = cMax - cMin + 1;
    espace = malloc(nbLin * nbCol * sizeof(double));
    if (espace == NULL) {
        erreurTable = 1;
        return NULL;
    }
    lignes = malloc(nbLin * sizeof(double *));
    if (lignes == NULL) {
        erreurTable = 1;
        free(espace);
        return NULL;
    }
    p = espace - cMin;
    for (i = 0; i < nbLin; i++) {
        lignes[i] = p;
        p += nbCol;
    }
    erreurTable = 0;
    return lignes - lMin;
}
```

Comme précédemment, l'emploi de ces matrices ne s'accompagne d'aucune perte d'efficacité. Voici un exemple d'utilisation :

```
...
double **m = dMatrice(5, 10, -3, 3);
...
for (i = 5; i <= 10; i++)
    for (j = -3; j <= 3; j++)
        m[i][j] = 1000 * i + j;
...
for (i = 5; i <= 10; i++) {
    for (j = -3; j <= 3; j++)
        printf("%10.1f ", m[i][j]);
    printf("\n");
}
...
```

6.2.8 Matrices non dynamiques de taille inconnue

Un dernier cas qui se présente parfois est celui où on doit écrire une fonction opérant sur des matrices ordinaires (c.-à-d. non dynamiques) alors que leurs dimensions ne sont pas connues lors de la compilation.

Par exemple, voici la fonction qui effectue le produit de deux matrices `A` et `B` en déposant le résultat dans une troisième matrice `C`. Les arguments `nla` et `nca` représentent le nombre effectif de lignes et de colonnes de `A`, tandis que `ncb` est le nombre effectif de colonnes de `B`. Les arguments `dNca`, `dNcb` et `dNcc` représentent les nombres de colonnes avec lesquels on a déclaré les matrices `A`, `B` et `C` respectivement.

```
void produitMatrices(double *A, double *B, double *C,
            int nla, int nca, int ncb, int dNca, int dNcb, int dNcc) {
    int i, j, k;
    double s;

    for (i = 0; i < nla; i++)
        for (j = 0; j < ncb; j++) {
            s = 0;
            for (k = 0; k < nca; k++)                    /*1*/
                s += A[i * dNca + k] * B[k * dNcb + j];  /*1*/
            C[i * dNcc + j] = s;
        }
}
```

Exemple d'appel (`NLA`, `NCA`, etc., sont des constantes introduites par des `#define`, tandis que `nla`, `nca`, etc. sont des variables) :

```
double A[NLA][NCA], B[NLB][NCB], C[NLC][NCC];
...
obtention des valeurs des variables nla, nca, etc., et des matrices A et B
...
produitMatrices(&A[0][0], &B[0][0], &C[0][0], nla, nca, ncb, NCA, NCB, NCC);
...
```

On notera que la fonction `produitMatrices` peut être optimisée en supprimant des des multiplications qui sont faites dans la boucle la plus profonde (la seule qui doit nous intéresser, du point de vue de l'efficacité). Pour cela, il suffit de remplacer les deux lignes marquées `/*1*/` par ceci (`ia` et `ib` sont des variables locales nouvelles) :


```
...
ia = i * dNca;                                  /*2*/
ib = j;                                         /*2*/
for (k = 0; k < nca; k++) {                     /*2*/
    s += A[ia] * B[ib];                         /*2*/
    ia += 1;                                    /*2*/
    ib += dNcb;                                 /*2*/
}                                               /*2*/
...
```

6.3 Les adresses des fonctions

6.3.1 Les fonctions et leurs adresses

En étudiant les déclarateurs complexes (section 5.4) nous avons appris que la syntaxe de C permettait de manipuler le type « fonction rendant un [objet de type] T » (T représente un autre type), cette périphrase constituant elle-même un *déclarateur* que le programmeur peut composer librement avec les autres déclarateurs, « adresse d'un [objet de type] T » et « tableau de [objets de type] T ». Tout cela suggère qu'en C on aura le droit de travailler avec des variables qui représentent des fonctions, des tableaux de fonctions, des fonctions renvoyant comme résultat d'autres fonctions, etc. Qu'en est-il exactement ?

Le concept le plus simple qu'on peut souhaiter dans ce domaine est celui de *variable dont la valeur est une fonction*. Il va falloir y renoncer, pour une raison facile à comprendre : la taille d'une fonction ne peut pas se déduire de son type[49]. C'est un inconvénient rédhibitoire, car toute définition de variable doit entraîner la réservation d'un espace mémoire pour en contenir la valeur, ce qui demande de connaître combien d'octets sont nécessaires.

En revanche, le langage C permet d'utiliser un concept à peine plus complexe que le précédent : celui de *variable dont la valeur est l'adresse d'une fonction*. Il n'y a plus de problème de taille (la taille d'une adresse – ou pointeur – est constante), et la syntaxe pour déclarer une telle variable a déjà été expliquée dans la section sur les déclarateurs complexes (cf. section 5.4). Exemple :

```
double (*uneFonction)(int n, double x);
```

Cet énoncé déclare `uneFonction` comme une variable (un pointeur) destinée à contenir des adresses de fonctions ayant deux arguments, un `int` et un `double`, et rendant une valeur `double`. Cette déclaration n'initialise pas `uneFonction` (sauf, si c'est une variable globale, avec la valeur 0, peu utile ici) et, à part le type des arguments et du résultat, ne laisse rien deviner sur les caractéristiques des fonctions dont `uneFonction` contiendra les adresses.

Si, par la suite, `uneFonction` reçoit l'adresse d'une fonction ϕ pour valeur, alors l'expression suivante est correcte et représente un appel de ϕ (en supposant `k` de type `int` et `u` et `v` de type `double`) :

```
u = (*uneFonction)(k, v);
```

Comment le programmeur obtient-il des adresses de fonctions ? Il faut savoir que, de la même manière que le nom d'un tableau représente son adresse de base, le nom d'une fonction représente l'adresse de celle-ci. Ainsi, si on a la définition de fonction

```
double maFonction(int nbr, double val) {
    etc.
}
```

alors l'expression `maFonction` :

- possède le type « adresse d'une fonction ayant pour arguments un `int` et un `double` et rendant un `double` » ;
- a pour valeur l'adresse où commence la fonction `maFonction` ;
- n'est pas une *lvalue*.

[49] Notez que, malgré les apparences, il n'est pas délirant de rapprocher les fonctions et les données. Une fois compilée (traduite en langage machine), une fonction est une suite d'octets qui occupe une portion de mémoire exactement comme, par exemple, un tableau de nombres. La principale différence est dans la possibilité ou l'impossibilité de prédire le nombre d'octets occupés.

Pour un tableau, par exemple, ce nombre découle sans ambiguïté de la déclaration. Pour une fonction il ne dépend pas de la déclaration (qui ne précise que le type du résultat et le nombre et les types des arguments) mais du nombre d'instructions qui composent la fonction, c'est-à-dire de ce qu'elle fait et comment elle le fait.

Par conséquent, l'affectation suivante est tout à fait légitime :

```
uneFonction = maFonction;
```

A partir de cette affectation (et tant qu'on aura pas changé la valeur de `uneFonction`) les appels de `(*uneFonction)` seront en fait des appels de `maFonction`.

Examinons deux applications de cela parmi les plus importantes.

6.3.2 Fonctions formelles

Le problème des fonctions formelles, c'est-à-dire des fonctions arguments d'autres fonctions, est résolu en C par l'utilisation d'arguments de type « pointeur vers une fonction ».

EXEMPLE 1 : RÉSOLUTION DE $f(x) = 0$ PAR DICHOTOMIE. Si f est une fonction continue, définie sur un intervalle $[a \dots b]$ telle que $f(a)$ et $f(b)$ ne sont pas de même signe, la fonction `dicho` recherche $x_\varepsilon \in [a \dots b]$ telle qu'il existe $x_0 \in [a \dots b]$ vérifiant $|x_\varepsilon - x_0| \leq \varepsilon$ et $f(x_0) = 0$. En clair : `dicho` détermine x_ε, qui est une solution de l'équation $f(x) = 0$ si on tolère une erreur d'au plus ε.

Les arguments de la fonction `dicho` sont : les bornes a et b de l'intervalle de recherche, la précision ε voulue et surtout la fonction f dont on cherche un zéro. Voici cette fonction :

```
double dicho(double a, double b, double eps, double (*f)(double)) {
    double x;                                       /* hypothèse : f(a) * f(b) <= 0 */
    if ((*f)(a) > 0) {
        x = a; a = b; b = x;
    }
    while (fabs(b - a) > eps) {
        x = (a + b) / 2;
        if ((*f)(x) < 0)
            a = x;
        else
            b = x;
    }
    return x;
}
```

L'argument `f` est déclaré comme un pointeur vers une fonction ; par conséquent `*f` est une fonction et `(*f)(x)` un appel de cette fonction. Si `une_fonction` est le nom d'une fonction précédemment définie, changeant de signe sur l'intervalle $[0, 1]$, un appel correct de `dicho` serait

```
x = dicho(0.0, 1.0, 1E-8, une_fonction);
```

Autre exemple du même tonneau

```
y = dicho(0.0, 3.0, 1E-10, cos);
```

REMARQUE. En syntaxe originale, ou « sans proptotypes », la fonction `dicho` s'écrirait

```
double dicho(a, b, eps, f)
    double a, b, eps, (*f)();
{
    etc. (la suite est la même)
}
```

EXEMPLE 2 : UTILISATION DE QSORT. Reprenons notre programme de la section 6.2.5 (tri d'un tableau de chaînes de caractères) en remplaçant la fonction de tri naïf par la fonction `qsort` de la bibliothèque standard. Cette fonction est expliquée à la section 8.4.2, elle trie un tableau par rapport à un critère quelconque. Si elle nous intéresse ici c'est parce qu'elle prend ce critère, représenté par une fonction, pour argument :

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

#define MAXMOTS  100
#define MAXCARS 3000

char espace[MAXCARS], *libre = espace;
char *mots[MAXMOTS];
int nbr_mots = 0;
```



```
        /* La déclaration suivante se trouve aussi dans <stdlib.h> : */
void qsort(void *base, size_t nmemb, size_t size,
                                        int (*comp)(const void *, const void *));

int comparer(const void *a, const void *b) {
        /* Les arguments de cette fonction doivent être déclarés "void *" (voyez la  */
        /* déclaration de qsort). En réalité, dans les appels de cette fonction que  */
        /* nous faisons ici, a et b sont des adresses de "chaînes", c'est-à-dire des */
        /* adresses d'adresses de char         */
    return strcmp(*(char **)a, *(char **)b);
}

main() {
    int i;
    while(gets(libre) != NULL && *libre != '\0') {
        mots[nbr_mots++] = libre;
        libre += strlen(libre) + 1;
    }
    qsort(mots, nbr_mots, sizeof(char *), comparer);
    for (i = 0; i < nbr_mots; i++)
        printf("%s\n", mots[i]);
}
```

REMARQUE. La définition de `comparer` peut paraître bien compliquée. Pour la comprendre, il faut considérer les diverses contraintes auxquelles elle doit satisfaire :

- `comparer` figurant comme argument dans l'appel de `qsort`, son prototype doit correspondre exactement[50] à celui indiqué dans le prototype de `qsort` ;
- `a` et `b`, les arguments formels de `comparer`, étant de type `void *`, on ne peut accéder aux objets qu'ils pointent sans leur donner auparavant (à l'aide de l'opérateur de conversion de type) un type pointeur « utilisable » ;
- d'après la documentation de `qsort`, nous savons que `comparer` sera appelée avec deux adresses de chaînes pour arguments effectifs ; une chaîne étant de type `char *`, le type « adresse de chaîne » est `char **`, ce qui explique la façon dont `a` et `b` apparaissent dans l'appel de `strcmp`.

6.3.3 Tableaux de fonctions

Le programme suivant montre l'utilisation d'un tableau d'adresses de fonctions : chaque élément du tableau table est formé de deux champs :

- le nom d'une fonction standard, sous forme de chaîne de caractères ;
- l'adresse de la fonction correspondante.

Notre programme lit des lignes constituées d'un nom de fonction, suivi d'un ou plusieurs blancs, suivis d'un nombre réel ; il évalue et affiche alors la valeur de la fonction mentionnée appliquée au nombre donné. La frappe d'une ligne réduite au texte fin arrête le programme.

```
#include <stdio.h>

double sin(double), cos(double), exp(double), log(double);

struct {
    char *nom;
    double (*fon)(double);
} table[] = {
        "sin", sin,
        "cos", cos,
        "exp", exp,
        "log", log };

#define NBF (sizeof table / sizeof table[0])
```

[50] En syntaxe « sans prototype », aucun contrôle n'est fait sur le type des arguments de `comparer`, et toute cette question est bien plus simple. Mais attention : quelle que soit la syntaxe employée, on ne peut pas ignorer le fait que `comparer` sera appelée (par `qsort`) avec pour arguments effectifs les adresses des chaînes à comparer.

```
main() {
    char nom[80];
    double x;
    int i;
    for(;;) {
        scanf("%s", nom);
        if (strcmp(nom, "fin") == 0)
            break;
        scanf("%lf", &x);
        for(i = 0; i < NBF && strcmp(table[i].nom, nom) != 0; i++)
            ;
        if (i < NBF)
            printf("%f\n", (*table[i].fon)(x));
        else
            printf("%s ???\n", nom);
    }
}
```

Voici une exécution de ce programme :

```
sin 1
0.841471
cos 1
0.540302
atn 1
atn ???
fin
```

REMARQUE. La déclaration du tableau `table` présente une caractéristique peu fréquente : des fonctions sont référencées mais elles ne sont pas en même temps appelées. En effet, les noms des fonctions `sin`, `cos`, `exp`, etc. apparaissent dans cette déclaration sans être accompagnés d'une paire de parenthèses ; il n'y a aucun signe indiquant au compilateur qu'il s'agit de fonctions. C'est pourquoi ce programme sera trouvé erroné par le compilateur à moins qu'on lui donne les déclarations externes des fonctions, soit explicitement :

```
double sin(double), cos(double), exp(double), log(double);
```

soit en incluant un fichier dans lequel sont écrites ces déclarations :

```
#include <math.h>
```

6.3.4 Flou artistique

A ce point de notre exposé vous êtes en droit de trouver que les rôles respectifs des objets de type « fonction... » et des objets de type « adresse d'une fonction... » sont confus. Examinons une fonction bien ordinaire, *sinus*. La déclaration de cette fonction, extraite du fichier `<math.h>`, est :

```
double sin(double);
```

D'autre part, la définition d'une variable de type « adresse d'une fonction `double` à un argument `double` » est :

```
double (*f)(double);
```

Voici l'initialisation de `f` par l'adresse de la fonction `sin` et deux appels équivalents de cette fonction (`x`, `y` et `z` sont des variables `double`) :

```
f = sin;
y = (*f)(x);
z = sin(x);
```

Quelle que soit la valeur de `x`, `y` et `z` reçoivent la même valeur. Pourtant, les trois lignes ci-dessus ne sont pas cohérentes. La première montre que `f` et `sin` sont de même type (à savoir « adresse d'une fonction... ») ; or `f` doit être déréférencé pour appeler la fonction, tandis que `sin` n'a pas besoin de l'être.

On peut comprendre la raison de cette incohérence. Après avoir constaté l'impossibilité de déclarer des variables de type « fonction... » , car une fonction n'a pas de taille définie, et donc l'obligation de passer par des variables « adresse de fonction... », on a voulu faire cohabiter la rigueur (un pointeur doit être déréférencé pour accéder à l'objet qu'il pointe) et la tradition (le sinus de x s'est toujours écrit $sin(x)$). Cela a conduit à définir l'opérateur d'appel de fonction aussi bien pour une expression de type « fonction... » que pour une expression

de type « adresse d'une fonction... ». Ainsi, les deux expressions `(*f)(x)` et `sin(x)` sont correctes ; l'expérience montre que cette tolérance n'a pas de conséquences néfastes. Il n'en découle pas moins que les deux expressions

```
u = f(x);
v = (*sin)(x);
```

sont acceptées elles aussi (elles ont la même valeur que les précédentes).

6.4 Structures récursives

6.4.1 Déclaration

Les structures récursives font référence à elles-mêmes. Elles sont principalement utilisées pour implanter des structures de données dont la définition formelle est un énoncé récursif. Parmi les plus fréquemment rencontrées : les listes et les arbres, comme les arbres binaires.

Ainsi, par exemple, on peut donner la définition récursive suivante : un arbre binaire est
- soit un symbole conventionnel signifiant « la structure vide » ;
- soit un triplet (*valeur* , $fils_g$, $fils_d$) constitué d'une *information* dont la nature dépend du problème étudié et de deux *descendants* qui sont des arbres binaires.

Bien entendu, du point de vue technique il n'est pas possible de déclarer un champ d'une structure comme ayant le même type que la structure elle-même (une telle structure serait *ipso facto* de taille infinie !). On contourne cette difficulté en utilisant les pointeurs. Un arbre binaire est représenté par une adresse, qui est :
- soit l'adresse nulle ;
- soit l'adresse d'une structure constituée de trois champs : une information et deux descendants qui sont des adresses d'arbres binaires.

Typiquement, la déclaration d'une telle structure sera donc calquée sur la suivante :

```
struct noeud {
    type valeur;
    struct noeud *fils_gauche, *fils_droit;
};
```

Notez que le nom de la structure (`noeud`) est ici indispensable, pour indiquer le type des objets pointés par les champs `fils_gauche` et `fils_droit`.

6.4.2 Exemple

Le programme suivant effectue le même travail que celui du 6.2.5, mais à la place d'une table on utilise un arbre binaire de recherche (voir, par exemple, R. Sedgewick, *Algorithmes en langage C*, chapitre 14) pour ranger les mots. La structure de données mise en œuvre est représentée sur la figure 16.

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

#define MAXCARS 3000 char

espace[MAXCARS], *libre = espace;

typedef struct noeud {
    char *mot;
    struct noeud *fils_gauche, *fils_droit;
} NOEUD, *POINTEUR;

POINTEUR alloc(char *m, POINTEUR g, POINTEUR d) {
    POINTEUR p = malloc(sizeof(NOEUD));
    p->mot = m;
    p->fils_gauche = g;
    p->fils_droit = d;
    return p;
}
```

```
POINTEUR insertion(POINTEUR r, char *mot) {
    if (r == NULL)
        r = alloc(mot, NULL, NULL);
    else
        if (strcmp(mot, r->mot) <= 0)
            r->fils_gauche = insertion(r->fils_gauche, mot);
        else
            r->fils_droit = insertion(r->fils_droit, mot);
    return r;
}
```

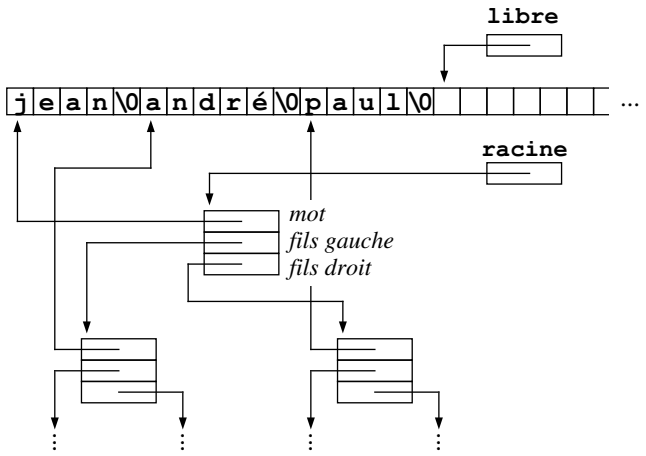

FIG. 16 – Arbre binaire de mots

```
void parcours(POINTEUR r) {
    if (r != NULL) {
        parcours(r->fils_gauche);
        printf("%s\n", r->mot);
        parcours(r->fils_droit);
    }
}

main() {
    POINTEUR racine = NULL;
    char *p;

    p = gets(libre);
    while(*p != '\0') {
        libre += strlen(p) + 1;
        racine = insertion(racine, p);
        p = gets(libre);
    }

    parcours(racine);
}
```

6.4.3 Structures mutuellement récursives

Donnons-nous le problème suivant : représenter une « liste de familles » ; chaque famille est une « liste d'individus », avec la particularité que chaque individu fait référence à sa famille. Les structures `famille` et `individu` se pointent mutuellement, chacune intervenant dans la définition de l'autre. Comment les déclarer ?

Pour résoudre ce problème le compilateur C tolère qu'on déclare un pointeur vers une structure non encore définie. Plus précisément, si la structure ayant le *nom* indiqué n'a pas encore été déclarée, l'expression « `struct`

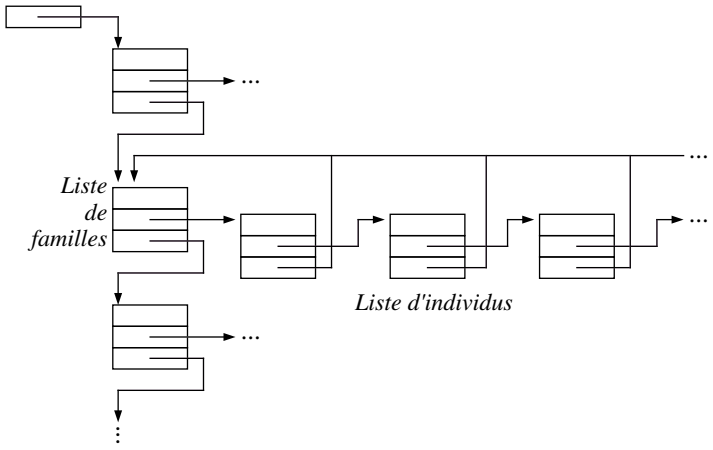

FIG. 17 – Structures mutuellement récursives

nom » est légitime. Elle identifie un type incomplet et on peut l'utiliser *à tout endroit où sa taille n'est pas requise*, notamment dans la déclaration d'un pointeur vers une telle structure. Cela nous permet de déclarer nos structures mutuellement récursives sous la forme :

```
typedef struct famille {
    char *nom;
    struct individu *membres;
    struct famille *famille_suivante;
} FAMILLE;

typedef struct individu {
    char *prenom;
    struct individu *membre_suivant;
    struct famille *famille;
} INDIVIDU;
```

Une autre solution aurait été :

```
typedef struct famille FAMILLE;
typedef struct individu INDIVIDU;

struct famille {
    char *nom;
    INDIVIDU *membres;
    FAMILLE *famille_suivante;
};

struct individu {
    char *prenom;
    INDIVIDU *membre_suivant;
    FAMILLE *famille;
};
```

7 Entrées-sorties

Strictement parlant, les opérations d'entrée-sortie ne font pas partie du langage C ; elles sont effectuées par des fonctions de la bibliothèque que chaque système offre avec le compilateur. Or, C est né et a vécu longtemps en milieu UNIX avant d'être transporté ailleurs ; la bibliothèque UNIX a donc souvent été prise pour référence, semant quelque confusion car la réalisation de certaines fonctions d'UNIX est inutile, voire impossible sur certains systèmes. La norme ANSI y a mis bon ordre ; en principe la liste des fonctions standard est maintenant clairement établie et un programme qui n'utilise que ces fonctions doit pouvoir être transporté d'un système à un autre sans modification. Rien n'empêche que telle ou telle réalisation particulière du langage propose des fonctions additionnelles ; il faudra alors vous référer à la documentation spécifique.

7.1 Flots

Quelles que soient les particularités du système d'exploitation sous-jacent, les fonctions de la bibliothèque standard des entrées-sorties font en sorte que les fichiers soient vus par le programmeur comme des *flots*, c'est-à-dire des suites d'octets qui représentent des données selon une des deux modalités suivantes :

- soit le fichier contient les caractères qui constituent l'expression écrite des données en question d'après une certaine syntaxe. Ces caractères sont eux-mêmes représentés selon une convention largement répandue, presque toujours le code ASCII. De tels fichiers sont appelés *fichiers de texte*. Leur principale qualité est d'être exploitables par un logiciel, un système ou un équipement différents de celui qui les a produits ; en particulier, ils peuvent être affichés, édités, imprimés, etc. ;
- soit le fichier contient des données enregistrées sous une forme qui est la copie exacte de leur codage dans la mémoire de l'ordinateur. On l'appelle alors *fichier binaire*. Les opérations de lecture ou d'écriture sur de tels fichiers sont très rapides, car elles ne requièrent pas un travail d'analyse ou de synthèse de l'expression écrite des données. En revanche, les fichiers binaires ne sont pas éditables ou imprimables ; ils ne sont pas censés être transportables d'un logiciel à un autre, encore moins d'un système à un autre.

Du point de vue du langage C (il n'en est peut être pas de même pour le système sous-jacent) être de texte ou binaire n'est pas une propriété d'un fichier, mais de la manière dont il est traité par un programme. Ainsi la distinction entre ces deux types de fichiers ne se fait pas lors de la déclaration ou de l'ouverture[51] du fichier, mais par le choix des fonctions de lecture ou d'écriture qui sont utilisées : certaines fonctions (`fgets` et `fputs`, lecture et écriture d'une ligne, et `fscanf` et `fprintf`, lecture et écriture de données formatées) n'ont de sens que sur un flot de texte. D'autres fonctions (`fread`, `fwrite`) effectuent la lecture et l'écriture de paquets d'octets sans interprétation aucune, et correspondent donc plutôt aux flots binaires[52].

Les flots binaires sont de simples suites d'octets. Les flots de texte ont une structure légèrement plus riche, puisqu'ils sont censés être organisés comme des suites de lignes. Chaque ligne est faite d'un nombre quelconque de caractères distincts de `'\n'` et se termine par le caractère `'\n'`. Même si sur un système particulier les fichiers de texte ne sont pas ainsi organisés, ou si le caractère qui indique la fin d'une ligne est différent, la bibliothèque standard fera en sorte qu'ils puissent être vus de cette manière par le programmeur[53].

Tampons. Un fichier est dit *tamponné*[54] lorsque les transferts physiques d'octets entre le fichier et un programme qui le lit ou l'écrit se font par paquets de taille fixe, la taille du tampon, même si les échanges logiques sont faits par paquets de taille variable ou même octet par octet. Par défaut un flot est toujours associé à un tampon de taille prédéterminée, mais le programmeur peut, à l'aide de la fonction `setvbuf` (cf. section 7.1.1), modifier la taille de ce dernier ou même le supprimer complètement.

Ce mode de gestion des fichiers réduit le nombre d'opérations physiques d'entrée - sortie (beaucoup plus lentes, quelles que soient les performances des disques durs actuels, que les transferts d'information purement internes) mais introduit un décalage dans leur chronologie. Ce que le programmeur croit être une opération d'écriture dans un fichier n'est en réalité qu'une « opération logique », c'est-à-dire une recopie dans un tampon situé en mémoire, dont le contenu sera transféré ultérieurement vers le fichier. Par conséquent, il est difficile ou impossible de savoir dans quel état se trouve le fichier à un moment donné ; de plus, si le programme vient à subir une terminaison anormale, une partie des informations logiquement écrites dans le fichier seront perdues,

[51] Il y a quand-même un point de détail qu'il fait fixer lors de l'ouverture (voyez la fonction `fopen`) : y a-t-il lieu de guetter les marques de fins-de-ligne pour les transformer dans le caractère `'\n'` et réciproquement ?

[52] Conséquence : sauf si le système sous-jacent l'interdit, un fichier écrit comme un flot de texte peut être lu comme un flot binaire. Cela revient à ignorer le fait que les octets dont il se compose représentent des données d'un ordre supérieur. La démarche inverse, lire comme un flot de texte un fichier binaire, n'a pas de signification et peut provoquer des erreurs fatales (les fins de ligne seront absentes ou incomprises, le tampon débordera, etc.).

[53] En particulier, le couple (*CR*,*LF*) utilisé par certains systèmes est traduit en lecture par l'unique caractère `'\n'`. Inversement, sur de tels systèmes, l'écriture « logique » du caractère `'\n'` se traduit par l'écriture effective du couple (*CR*,*LF*).

[54] En bon franglais, on dit plutôt « bufferisé ».

parce qu'elles n'ont jamais été physiquement transférées[55].

7.1.1 Fonctions générales sur les flots

Les flots sont représentés dans les programmes par des variables de type `FILE *`. La structure `FILE` est définie dans le fichier `<stdio.h>`. Pour utiliser un ou plusieurs flots il faut donc écrire en tête de son programme la directive

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

puis une déclaration de la forme

`FILE *`*fichier*`;`

Un fichier est donc représenté en C par une variable d'un type pointeur. Par conséquent, sa déclaration n'autorise aucune des opérations propres aux fichiers, aussi longtemps qu'on n'aura pas fait le nécessaire (voyez la fonction `fopen` ci-après) pour qu'il pointe sur une structure légitimement allouée et qui constitue la description d'un fichier ouvert.

Parmi les principales fonctions qui effectuent les opérations générales (ouverture, fermeture, etc.) sur les flots nous avons :

`FILE *fopen(const char *nom, const char *mode)`

Cette fonction ouvre le fichier dont le nom est indiqué par la chaîne nom et rend un pointeur sur le flot correspondant, ou `NULL` si l'opération a échoué (fichier absent, etc.). Le nom doit être correct pour le système d'exploitation sous-jacent ; cela ne regarde pas le langage C.

Les valeurs permises pour `mode` sont :

- `"r"` (*read*) ouverture d'un fichier. Le fichier doit exister ; son contenu n'est pas détruit. Le descripteur du flot créé est positionné en lecture et au début du fichier. En principe, seules les opérations de lecture sont permises sur ce flot.
- `"r+"` comme `"r"`, mais les opérations d'écriture sont permises aussi.
- `"w"` (write) création d'un fichier. Le fichier peut exister ou non ; s'il existe, son contenu est entièrement effacé. Le descripteur du flot créé est positionné en écriture et au début du fichier (qui est vide). En principe, seules les opérations d'écriture sont permises sur ce flot.
- `"w+"` comme `"w"`, mais les opérations de lecture sont permises aussi.
- `"a"` (append) allongement d'un fichier. Le fichier existe ou non ; s'il existe, son contenu n'est pas effacé. Le descripteur du flot créé est positionné en écriture et à la fin du fichier. Seules les opérations d'écriture sont permises.
- `"a+"` comme `"a"`, mais les opérations de lecture sont permises aussi.
- `"rb"`, `"r+b"`, `"wb"`, `"w+b"`, `"ab"`, `"a+b"` : si on envisage d'utiliser le fichier en mode binaire il faut ajouter la lettre `b` au mode (`"r+b"`, `"w+b"` et `"a+b"` peuvent se noter aussi `"rb+"`, `"wb+"` et `"ab+"`) .

REMARQUE 1. La seule différence que la lettre `b` dans le mode introduit dans le fonctionnement de *toutes* les fonctions d'entrée-sortie est la suivante : si le fichier est « de texte » (i.e. si la lettre `b` ne figure pas dans le mode) alors :
- en lecture, chaque occurrence d'une marque de fin de ligne est détectée et, si cette marque n'est pas le caractère `'\n'` lui-même, remplacée par l'unique caractère `'\n'`,
- toute demande d'écriture du caractère `'\n'` produit en fait l'envoi effectif au fichier de la marque de fin de ligne requise par le système sous-jacent.

Il n'est donc pas exclu que sur un système d'exploitation donné l'ajout de la lettre `b` au mode soit sans effet. C'est notamment le cas d'UNIX, où les fins de ligne sont indiquées par le caractère `'\n'`.

REMARQUE 2. Sur l'intérêt qu'il peut y avoir à effectuer des lectures et des écritures sur le même fichier, voir les remarques faites à la section 7.4.3 à propos des fichiers en accès relatif.

`int fflush(FILE *flot)`

Cette fonction provoque l'écriture physique immédiate du tampon en cours de remplissage pour le flot indiqué. Elle rend `EOF` en cas d'erreur, zéro dans les autres cas.

REMARQUE. Si le fichier physique qui correspond au flot indiqué est un *organe interactif*, par exemple si c'est l'écran d'un poste de travail, alors la fonction `fflush` est automatiquement appelée dans deux circonstances très fréquentes :

[55] S'il existe une probabilité non négligeable pour qu'un programme ait une fin anormale (par exemple à cause de conditions d'exploitation difficiles) il est prudent de programmer des appels réguliers de la fonction `fflush`, cf. section 7.1.1.

– l'écriture du caractère `'\n'` qui produit l'émission d'une marque de fin de ligne et, donc, la vidange effective du tampon,
– le début d'une opération de lecture sur l'unité d'entrée associée. En clair : toute lecture au clavier provoque la vidange du tampon d'écriture à l'écran ; cela permet d'afficher une question avant de lire la réponse correspondante).

`int fclose(FILE *flot)`

Cette fonction ferme le flot indiqué, produisant l'écriture physique des tampons, la restitution de l'espace alloué pour le tampon et les descripteurs du flot, etc. Elle rend zéro en cas de succès, une autre valeur en cas d'erreur.

A la suite d'un appel comme `fclose(flot)` l'emploi de la variable `flot` est illégitime.

Oublier de fermer, par un appel de `fclose`, un fichier qui a été ouvert en lecture n'est en général pas une erreur importante. Mais oublier de fermer un fichier qui a été ouvert en écriture (c'est-à-dire un fichier qui vient d'être créé) peut être fatal pour l'information qui y a été écrite, car
– le contenu du tampon d'écriture en cours de formation lors de la terminaison du programme ne sera probablement pas sauvegardé dans le fichier ;
– dans certains systèmes, les fichiers nouvellement créés ne subissent les opérations qui les rendent connus du système d'exploitation (comme l'inclusion de leur nom dans un répertoire) qu'au moment de leur fermeture. Un fichier qui n'a jamais été fermé a donc une existence très précaire.

`int feof(FILE *flot)`

Cette fonction n'a d'intérêt que si `flot` désigne un fichier en lecture. Elle renvoie une valeur non nulle si et seulement si l'indicateur de fin de fichier du `flot` indiqué est vrai. Cela se produit non pas lorsque le fichier est positionné *sur* sa fin mais lorsqu'il est placé *au-delà* de sa fin.

Autrement dit, `feof(fic)` devient vrai immédiatement *après* qu'une lecture sur `fic` ait échoué parce qu'il n'y avait plus [assez] d'informations dans `fic` pour la satisfaire.

En particulier, `feof` n'est jamais vrai après l'ouverture (en lecture) du fichier, même lorsque ce dernier est vide. Il faut au moins une [tentative de] lecture pour, éventuellement, le rendre vrai.

Cette spécification de la fonction `feof` fonde le schéma suivant, de *lecture séquentielle d'un fichier* :

```
FILE *fichier;
...
fichier = fopen(nom, "rb");
vérification du succès de l'ouverture de fichier
...
lecture de données sur fichier
while ( ! feof(fichier)) {
    traitement des données lues
    lecture de données sur fichier
}
...
```

`int ferror(FILE *flot)`

Cette fonction renvoie une valeur non nulle si et seulement si l'indicateur d'erreur du `flot` indiqué est « vrai ». Elle doit être appelée *immédiatement* après une entrée-sortie sur ce flot pour savoir si l'opération en question a échoué. La variable globale entière `errno` (déclarée dans `<errno.h>`) contient alors un code renseignant sur la nature de l'erreur.

`FILE *tmpfile(void)`

Cette fonction crée un fichier temporaire en mode `"w+b"`, sans nom externe, qui sera automatiquement détruit à la terminaison du programme. Elle renvoie le flot associé ou NULL en cas d'erreur.

`int setvbuf(FILE *flot, char *tampon, int mode, size_t taille)`

Cette fonction redéfinit le tampon associé à un `flot` ouvert. Elle doit être appelée après l'ouverture du flot mais avant toute lecture ou écriture. L'argument `mode` doit être une des trois constantes suivantes, définies dans `<stdio.h>` :

`IOFBF` Tampon de taille fixe. L'écriture ou la lecture physique a lieu lorsque le tampon contient `taille` caractères.

`IOLBF` Tampon égal à une ligne. L'écriture ou la lecture physique a lieu lors de la rencontre du caractère `'\n'` qui détermine la fin des lignes.

`IONBF` Pas de tampon. Une écriture ou une lecture physique a lieu à chaque lecture ou écriture logique.

L'argument `taille` indique la taille voulue pour le tampon. L'argument `tampon`, s'il n'est pas `NULL`, est censé pointer un espace (de taille au moins égale à `taille`) qui sera utilisé comme tampon. Si `tampon` est `NULL`, alors la fonction alloue un espace propre.

7.1.2 Les unités standard d'entrée-sortie

Sans que le programmeur n'ait à prendre aucune disposition particulière, l'exécution d'un programme commence avec trois flots de texte automatiquement ouverts par le système. Ils sont connectés (on dit *affectés*) aux organes d'entrée-sortie les plus « naturels » par rapport à la manière dont on utilise l'ordinateur. Ces fichiers sont déclarés dans `<stdio.h>`, de la manière suivante :

```
FILE *stdin, *stdout, *stderr;
```

`stdin` est l'unité standard d'entrée. Elle est habituellement affectée au clavier du poste de travail.

`stdout` est l'unité standard de sortie. Elle est habituellement affectée à l'écran du poste de travail.

`stderr` est l'unité standard d'affichage des erreurs. Elle est aussi affectée à l'écran du poste de travail.

Les fonctions de lecture (resp. d'écriture) qui ne spécifient pas un autre flot utilisent `stdin` (resp. `stdout`).

Remarque. L'utilisation des unités standard apparaît encore plus intéressante lorsque l'on sait que dans UNIX et plusieurs autres systèmes d'exploitation il est permis de réaffecter les unités standard lors d'une exécution du programme, sans qu'il faille recompiler ce dernier. En UNIX et MS-DOS cela fonctionne de la manière suivante : supposons que `prog1` soit un programme qui n'utilise que les unités standard `stdin` et `stdout`. Activé par la commande `prog1`, il lira ses données au clavier et affichera ses résultats à l'écran. Mais s'il est lancé par la commande

```
prog1 < fichier1
```
(resp. : `prog1 > fichier2`)

alors le fichier `fichier1` (resp. `fichier2`) remplacera le clavier (resp. l'écran). De manière similaire, supposons que `prog2` soit un autre programme qui fait ses entrées-sorties de la même manière. Alors la commande

```
prog1 | prog2
```

active en parallèle ces deux programmes avec la sortie standard de `prog1` connectée à l'entrée standard de `prog2` (les sorties de `prog1` sont lues et exploitées par `prog2` au fur et à mesure qu'elles sont produites par `prog1`). Bien entendu, tout cela se combine. Par exemple, la commande

```
prog1 < fichier1 | prog2 > fichier2
```

active `prog1` et `prog2`. Les données de `prog1` sont lues dans `fichier1`, ses résultats sont fournis à titre de données à `prog2`, dont les résultats sont écrits dans `fichier2`.

7.2 Lecture et écriture textuelles

7.2.1 Lecture et écriture de caractères et de chaînes

Les fonctions de lecture-écriture les plus simples effectuent la lecture ou l'écriture d'un caractère isolé ou bien la lecture ou l'écriture d'une ligne. Pour la lecture nous avons :

`int fgetc(FILE *flot)`
: Renvoie le caractère suivant sur le flot indiqué, ou `EOF` si la fin du fichier est atteinte ou si une erreur survient. C'est une vraie fonction.

`int getc(FILE *flot)`
: Fait la même chose que `fgetc` mais c'est peut être macro.

  L'intérêt d'utiliser une macro réside dans l'efficacité supérieure des (petites) macros par rapport aux (petites) fonctions[56]. Il ne faut pas oublier cependant qu'il y a deux cas où les macros ne peuvent pas être utilisées :
  – lorsque l'argument a un effet de bord. Une expression comme

    ```
    *s++ = fgetc(table_de_fichiers[i++])
    ```

    est correcte (mais peu utilisée), tandis que

[56] Il s'agit d'entrées-sorties tamponnées, c'est-à-dire de simples transferts entre mémoires. S'il s'était agi d'opérations physiques, le gain de temps obtenu en utilisant getc à la place de fgetc aurait été négligeable devant le temps requis par l'entrée-sortie elle-même.

```
*s++ = getc(table_de_fichiers[i++])
```

est certainement erronée, car ayant un effet indéfini (on ne peut pas dire combien de fois `i` sera incrémenté) ;
– lorsqu'il faut obtenir l'adresse de l'opération en question, pour la passer comme argument d'une autre fonction. Par exemple, si `getc` est une macro, un appel comme `appliquer(getc, fichier)` sera certainement trouvé incorrect par le compilateur.

`int getchar(void)`

`getchar()` équivaut à `getc(stdin)` ;

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

Lecture d'une ligne en veillant à ne pas déborder. Plus précisément, cette fonction lit des caractères sur le `flot` indiqué et les place dans l'espace pointé par `s`. Elle s'arrête lorsqu'elle a lu `n`−1 caractères ou lorsqu'elle a rencontré un caractère `'\n'` de fin de ligne (ce caractère est copié dans le tableau). Un caractère `'\0'` est ajouté à la fin du tableau. La fonction rend `s`, ou `NULL` si une erreur ou la fin du fichier a été rencontrée.

Lisez aussi la remarque faite ci-après à propos de la fonction `gets`.

`char *gets(char *s)`

Lecture d'une ligne sur le flot `stdin`. Les caractères lus sont rangés dans l'espace pointé par `s`. Elle renvoie le même résultat que `fgets` mais, contrairement à cette dernière
– il n'y a pas de test de débordement[57] ;
– le caractère de fin de ligne est remplacé par `'\0'`.

ATENTION. L'argument de `gets` doit être l'adresse d'un espace disponible (accessible en écriture) et de taille suffisante, dans lequel `gets` puisse ranger les caractères qui seront lus.

EXEMPLES. Pour illustrer ces fonctions, voici deux exercices d'école. D'une part, l'écriture d'une fonction analogue à `gets` en utilisant `getchar` :

```
char *myGets(char *s) {
    int c, i = 0;
    c = getchar();

    while (c != '\n') {
        if (c == EOF)
            return NULL;
        s[i++] = c;
        c = getchar();
    }
    s[i] = '\0';
    return s;
}
```

D'autre part, l'écriture d'une fonction analogue à `getchar` (version fonction) en utilisant `gets` :

```
int myGetchar(void) {
    static char tampon[256] = "";
    static char *pos = tampon;
    if (*pos == 0) {
        if (gets(tampon) == NULL)
            return EOF;
        strcat(tampon, "\n");
        pos = tampon;
    }
    return *pos++;
}
```

Voyons maintenant les fonctions d'écriture :

[57] Ainsi, `gets` lit des caractères jusqu'à la rencontre de `'\n'`, sans se préoccuper de savoir si l'espace dans lequel ces caractères sont rangés est de taille suffisante pour les recevoir ; autrement dit, tout appel de `gets` peut entraîner un débordement de mémoire. C'est la raison pour laquelle on dit que *tout programme comportant ne serait-ce qu'un appel de* `gets` *est bogué.*

`int fputc(int caractere, FILE *flot)`
Écrit le `caractere` indiqué sur le `flot` indiqué. Renvoie le caractère écrit ou `EOF` si une erreur survient. C'est une vraie fonction.

`int putc(int caractere, FILE *flot)`
Fait la même chose que `fputc` mais peut être une macro. Voir à ce propos les remarques faites à l'occasion de l'introduction de `getc`.

`int putchar(int caractere)`
`putchar(c)` équivaut à `putc(c, stdout)`.

`int fputs(const char *s, FILE *flot)`
Écrit la chaîne `s` sur le `flot` indiqué. La fonction renvoie `EOF` en cas d'erreur, une valeur ≥ 0 dans les autres cas.

`int puts(const char *s)`
Cette fonction écrit la chaîne `s` sur le flot `stdout`. Elle renvoie le même résultat que `fputs`. Contrairement à `fputs` un caractère `'\n'` est ajouté à la suite de la chaîne.

`int ungetc(int c, FILE *flot)`
« Délecture » du caractère `c`. Cette fonction remet le caractère `c` dans le `flot` indiqué, où il sera trouvé lors de la prochaine lecture. Sur chaque flot, seule la place pour *un* caractère « délu » est garantie. On ne peut pas « délire » le caractère `EOF`.

Cette opération incarne donc l'idée de restitution (à l'unité d'entrée, qui en est le fournisseur) d'un caractère lu à tort. C'est une manière de résoudre un problème que pose l'écriture d'analyseurs lexicaux, où la fin de certaines unités lexicales est indiquée par un caractère qui les suit sans en faire partie. Exemple : la lecture d'un nombre entier non négatif :

```
int lirentier(FILE *flot) {      /* lecture d'un entier (simpliste) */
    int c, x = 0;
    c = getc(flot);
    while ('0' <= c && c <= '9') {
        x = 10 * x + c - '0';
        c = getc(flot);
    }
                                 /* ici, c est le caractère suivant le nombre */
    ungetc(c, flot);
    return x;
}
```

7.2.2 Ecriture avec format `printf`

La fonction d'écriture sur l'unité de sortie standard (en principe l'écran du poste de travail) avec conversion et mise en forme des données est :

`printf(` *format* `,` $expr_1$ `,` $expr_2$ `, ...` $expr_n$ `)`

format est une chaîne qui joue un rôle particulier. Elle est formée de caractères qui seront écrits normalement, mélangés à des indications sur les types des expressions $expr_1$, $expr_2$, ... $expr_n$ et sur l'aspect sous lequel il faut écrire les valeurs de ces expressions.

Cela fonctionne de la manière suivante : la fonction `printf` parcourt la chaîne *format* et reconnaît certains groupes de caractères comme étant des *spécifications de format*. C'est le caractère `%` qui déclenche cette reconnaissance. La fonction `printf` recopie sur le flot de sortie tout caractère qui ne fait pas partie d'une telle spécification. La i^{eme} spécification détermine la manière dont la valeur de $expr_i$ sera écrite.

Les spécifications de format reconnues par la fonction `printf` sont expliquées dans la table 4.

Notez que seuls les éléments **1** (le caractère `%`) et **6** (la lettre qui indique le type de la conversion) sont obligatoires dans une spécification de format.

Exemple 1. Supposons qu'on ait donné la déclaration-initialisation

```
float x = 123.456;
```

Voici une liste d'appels de `printf`, avec l'affichage produit par chacun :

Appel	*Affichage obtenu*
`printf(">%f<", x);`	`>123.456001<`
`printf(">%12f<", x);`	`> 123.456001<`
`printf(">%12.2f<", x);`	`> 123.46<`
`printf(">%.2f<", x);`	`>123.46<`
`printf(">%-12.2f<", x);`	`>123.46 <`
`printf(">%+-12.2f<", x);`	`>+123.46 <`
`printf(">% -12.2f<", x);`	`> 123.46 <`
`printf(">%12.2f<", x);`	`> 123.46<`
`printf(">%012.2f<", x);`	`>000000123.46<`
`printf(">%.0f<", x);`	`>123<`
`printf(">%#.0f<", x);`	`>123.<`

EXEMPLE 2. La liste suivante illustre l'utilisation de la largeur et de la précision pour les chaînes. Supposons avoir donné les déclarations :

```
char *s = "ABCDEFG";
int l = -10, n = 5;
```

Voici une liste d'appels de printf, avec l'affichage produit par chacun :

Appel	*Affichage obtenu*
`printf("printf(">%5s<", s);`	`>ABCDEFG<`
`printf("printf(">%10s<", s);`	`> ABCDEFG<`
`printf("printf(">%-10s<", s);`	`>ABCDEFG <`
`printf("printf(">%10.5s<", s);`	`> ABCDE<`
`printf("printf(">%-10.5s<", s);`	`>ABCDE <`
`printf("printf(">%.5s<", s);`	`>ABCDE<`
`printf("printf(">%*.*s<", l, n, s);`	`>ABCDE <`

La largeur du champ (cf. tableau 4, n° 3) peut être indiquée par le caractère `*` à la place d'un nombre. Elle est alors définie par la valeur de l'argument dont c'est le tour, qui doit être de type entier. Il en est de même pour la spécification de la précision (cf. tableau 4, n° 4). Par exemple, `i` et `j` étant des variables entières et `x` une variable flottante, l'expression

```
printf("%*.*f", i, j, x);
```

écrit la valeur de `x` sur un nombre de caractères égal à la valeur de `i`, avec un nombre de chiffres après la virgule égal à la valeur de `j`.

Chaque spécification de format se compose, dans l'ordre indiqué, des éléments suivants :

1. Obligatoirement, le caractère `%`.

2. Facultativement, un ou plusieurs *modifieurs*, dans un ordre quelconque, parmi :

- `-` : cadrer à gauche du champ (par défaut le cadrage se fait à droite) ;
- `+` : imprimer le signe du nombre même lorsque celui-ci est positif ;
- ***espace*** : si le premier caractère n'est pas un signe, placer un espace au début d'un nombre ;
- `0` : compléter les nombres par des zéros (à la place de blancs) ;
- `#` : utiliser le format alternatif, lorsqu'il existe (voir ci-dessous).

3. Facultativement, un nombre qui indique la *largeur du champ*. Il s'agit d'une largeur minimum, automatiquement augmentée si elle se révèle insuffisante.

4. Facultativement, un point suivi d'un nombre qui indique la précision :
- dans les conversions `e`, `E` ou `f` : le nombre de chiffres à écrire après la virgule décimale ;
- dans les conversions `g` ou `G` : le nombre de chiffres significatifs ;
- pour une chaîne : le nombre maximum de caractères à afficher ;
- pour un entier (conversions `d` et `u`) : le nombre minimum de chiffres. Le cas échéant, des zéros seront ajoutés à gauche.

5. Facultativement, une lettre qui complète l'information sur la nature de l'argument et, le cas échéant, modifie en conséquence la largeur du champ :

- `h` (associée à `d` ou `u`) : l'argument est un `short`. Cette indication est sans effet si un `int` et un `short` sont la même chose ;
- `l` (associée à `d`, `u`, `x` ou `X`) : l'argument est un `long` ou un `unsigned long`. Cette indication est sans effet si un `int` et un `long` sont la même chose ;
- `L` (associée à `f`, `e`, `E`, `g` ou `G`) : l'argument est un `long double`.

6. Obligatoirement, un caractère de conversion, parmi

- `d` : argument : `int` ; impression : notation décimale signée.
- `o` : argument : `int` ; impression : notation octale non signée.
  Format alternatif (voir `#` au point **2**) : un 0 est imprimé au début du nombre.
- `x` : argument : `int` ; impression : notation hexadécimale non signée avec les chiffres `0...9abcdef`.
  Format alternatif : le nombre est précédé de `0x`.
- `X` : comme `x` mais avec `ABCDEF` et `0X` à la place de `abcdef` et `0x`.
- `u` : argument : `int` ; impression : notation décimale non signée.
- `c` : argument : `unsigned char` ; impression : un seul caractère.
- `s` : argument : `char *` ; impression : chaîne de caractères.
- `f` : argument : `double` ; impression : notation en « virgule fixe » [-]*zzz.fff*. Le nombre de *f* est donné par la précision. Précision par défaut : 6. Si la précision vaut zéro, le point est supprimé.
  Format alternatif : le point est toujours imprimé.
- `e` : argument : `double`. Impression : notation en « virgule flottante » [-]*z.fffff*`e`±*nn*. Le nombre de f est donné par la précision. Précision par défaut : 6. Si la précision vaut zéro, le point est supprimé.
  Format alternatif : le point est toujours imprimé.
- `E` : comme `%e` avec `E` à la place de `e`.
- `g` : argument : `double` ; impression : comme `%e` si l'exposant est inférieur à -4 ou supérieur à la précision, comme `%f` sinon.
  Format alternatif : le point est toujours imprimé.
- `G` : comme `%g` avec `E` à la place de `e`.
- `p` : argument : `void *` ; impression : dépend de l'implantation (par exemple `%X`).
- `n` : argument : `int *`. *Aucune impression.* Effet : l'argument correspondant doit être l'adresse d'une variable entière, qui recevra pour valeur le nombre de caractères effectivement écrits jusqu'à présent par l'appel en cours de la fonction `printf`.
- `%` : pas d'argument correspondant ; impression : le caractère `%`.

TAB. 4 – Spécifications de format pour `printf`, `fprintf` et `sprintf`

7.2.3 Lecture avec format `scanf`

La fonction de lecture avec format à l'unité d'entrée standard (en principe le clavier du poste de travail) est

`scanf(` *format* `,` $adresse_1$ `,` $adresse_2$ `, ...` $adresse_n$ `)`

L'argument *format* est une chaîne qui indique la manière de convertir les caractères qui seront lus ; $adresse_1$, $adresse_2$, ... $adresse_n$ indiquent les variables devant recevoir les données lues. Ce sont des « sorties » de la fonction `scanf`, par conséquent *il est essentiel que ces arguments soient des adresses* de variables à lire.

La fonction `scanf` constituant un vrai analyseur lexical, les données lues ont de nombreuses occasions d'être incorrectes et on peut s'attendre à ce que `scanf` soit d'un maniement difficile. L'expérience ne déçoit pas cette attente ! Voici un bon conseil : si un programme ne fonctionne pas comme il le devrait, commencez par vérifier les valeurs lues par la fonction `scanf` (par exemple avec des appels de `printf` suivant immédiatement les appels de `scanf`).

Fonctionnement : `scanf` parcourt le format. Elle rend la main lorsque la fin du format est atteinte ou sur une erreur. Jusque-là :

- Tout caractère ordinaire du format, c'est-à-dire qui n'est ni un caractère d'espacement (blanc, tabulation) ni un caractère faisant partie d'une spécification de format (commençant par `%`), doit s'identifier au caractère courant du flot d'entrée. Si cette identification a lieu, le caractère courant est lu, sans être rangé dans aucune variable, et le parcours du format se poursuit. Si l'identification n'a pas lieu, l'activation de `scanf` se termine.
- Les spécifications de format commencent par `%`. Elles indiquent la manière d'analyser les caractères lus sur le flot d'entrée et de ranger les valeurs ainsi obtenues. Voir le tableau 5.
- Dans le format, les caractères ordinaires et les spécifications peuvent être séparés entre eux par des caractères d'espacement. Le nombre de ces espacements est sans importance, mais leur présence indique que les données correspondantes peuvent être séparées dans le flot d'entrée par un nombre quelconque de caractères d'espacement ou de fin de ligne.
- S'il n'y a pas d'espacement, dans le format, entre les caractères ordinaires ou les spécifications, alors les données correspondantes dans le flot d'entrée doivent être adjacentes.
- Cependant, les espacements au début des nombres sont toujours sautés.

On peut mélanger des appels de `scanf` et d'autres fonctions de lecture. Chaque appel d'une telle fonction commence par lire le premier caractère que l'appel précédent n'a pas « consommé ».

EXEMPLE 1. Lecture d'un entier :

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

EXEMPLE 2. Lecture de deux entiers séparés par une virgule et encadrés par une paire de parenthèses :

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

Données correctes :

`(22,33)` ou `( 22, 33)`

(des blancs, mais uniquement devant les nombres). Données incorrectes :

`( 22 , 33 )`

(trop de blancs).

EXEMPLE 3. Comme ci-dessus mais avec une syntaxe plus souple (les blancs sont tolérés partout) :

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

Toutes les données de l'exemple précédent sont correctes. Exemple de donnée incorrecte :

`( 22 ; 33 )`

EXEMPLE 4. Lecture d'*un* caractère[58] :

```
char calu;
...
scanf("%c", &calu);
```

REMARQUE. Nous avons indiqué par ailleurs qu'en C on range souvent les caractères dans des variables de type `int` afin de permettre la mémorisation d'une valeur « intruse », `EOF`. Cependant, notez bien qu'ici nous sommes obligés de déclarer la variable `calu` de type `char` ; le programme

```
int calu;
...
scanf("%c", &calu);
```

[58] Rappelons que `fgetc`, `getc` et `getchar` font ce travail bien plus simplement.

Chaque spécification de format se compose, dans l'ordre indiqué, des éléments suivants :

1. Obligatoirement, le caractère `%`.

2. Facultativement, le caractère `*` qui indique une suppression d'affectation : lire une donnée comme indiqué et l'« oublier » (c'est-à-dire ne pas le ranger dans une variable).

3. Facultativement, un nombre donnant la largeur maximum du champ (utile, par exemple, pour la lecture de nombres collés les uns aux autres).

4. Facultativement, une lettre qui apporte des informations complémentaires sur la nature de l'argument correspondant :

`h` devant `d`, `i`, `o`, `u`, `x` : l'argument est l'adresse d'un `short` (et non pas l'adresse d'un `int`) ;

`l` devant `d`, `i`, `o`, `u`, `x` : l'argument est l'adresse d'un `long` (et non pas l'adresse d'un `int`).
Devant `e`, `f`, `g` : l'argument est l'adresse d'un `double` (et non pas d'un `float`) ;

`L` devant `e`, `f`, `g` : l'argument est l'adresse d'un `long double` (et non pas d'un `float`).

5. Obligatoirement, un caractère de conversion parmi

`d` – Argument : `int *`. Donnée : nombre entier en notation décimale.

`i` – Argument : `int *`. Donnée : entier en notation décimale, octale (précédé de `0`) ou hexadécimale (précédé de `0x` ou `0X`).

`o` – Argument : `int *`. Donnée : entier en octal (précédé ou non de `0`).

`u` – Argument : `unsigned int *`. Donnée : entier en notation décimale.

`x` – Argument : `int *`. Donnée : entier en notation hexadécimale (précédé ou non de `0x` ou `0X`).

`c` – Argument : `char *`. Donnée : autant de caractères que la largeur du champ l'indique (par défaut : 1). Ces caractères sont copiés dans l'espace dont l'adresse est donnée en argument. Les blancs et tabulations ne sont pas pris pour des séparateurs (ils sont copiés comme les autres caractères) et il n'y a pas de `'\0'` automatiquement ajouté à la fin de la chaîne.

`s` – Argument : `char *`. Donnée : chaîne de caractères terminée par un caractère d'espacement (blanc, tabulation, fin de ligne) qui n'en fait pas partie. Un `'\0'` sera automatiquement ajouté après les caractères lus.

`e` – Argument : `float *`. Donnée : nombre flottant, c'est-à-dire dans l'ordre : signe facultatif, suite de chiffres, point décimal et suite de chiffres facultatifs, exposant (`e` ou `E`, signe facultatif et suite de chiffres) facultatif.

`f` – Comme e.

`g` – Comme e.

`p` – Argument : `void **` (ou `void *`, pour ce que ça change...). Donnée : nombre exprimant un pointeur comme il serait imprimé par `printf` avec le format `%p`.

`n` – Argument : `int *`. Aucune lecture. Effet : l'argument correspondant doit être l'adresse d'une variable entière ; celle-ci reçoit pour valeur le nombre de caractères effectivement lus jusqu'à ce point par le présent appel de la fonction `scanf`.

`[`*caractère*`...`*caractère*`]` – Argument : `char *`. Donnée : la plus longue suite faite de caractères appartenant à l'ensemble[a] indiqué entre crochets. Un caractère au moins sera lu. Un `'\0'` est ajouté à la fin de la chaîne lue.

`[^`*caractère*`...`*caractère*`]` – Argument : `char *`. Donnée : comme ci-dessus mais les caractères permis sont maintenant ceux qui n'appartiennent pas à l'ensemble indiqué.

`%` Pas d'argument. Le caractère `%` doit se présenter sur le flot d'entrée.

[a] Pour spécifier le caractère] l'écrire immédiatement après le [initial.

TAB. 5 – Spécifications de format pour `scanf`, `fscanf` et `sscanf`

aurait été incorrect (bien qu'il puisse fonctionner correctement sur certains systèmes), car on y fournit à `scanf` l'adresse d'un entier (fait de plusieurs octets) pour y ranger un caractère, alors que `scanf`, d'après l'analyse du format, « croit » recevoir l'adresse d'un `char` (un octet). Or on ne doit pas faire d'hypothèse sur la manière dont sont organisés les octets qui composent un entier, le langage C ne précisant rien à ce sujet. Une autre solution correcte aurait été la suivante :

```
int calu;
char tmp;
...
scanf("%c", &tmp);
calu = tmp;
```

EXEMPLE 5. Lecture d'un nombre entier et du caractère le suivant immédiatement, quel qu'il soit :

```
int nblu;
char calu;
...
scanf("%d%c", &nblu, &calu);
```

EXEMPLE 6. Lecture d'un nombre entier et du premier caractère *non blanc* le suivant :

```
int nblu;
char calu;
...
scanf("%d  %c", &nblu, &calu);
```

7.2.4 A propos de la fonction `scanf` et des lectures interactives

A PROPOS DE « SAUTER LES BLANCS ». Nous avons dit que les spécifications qui constituent le format peuvent être séparées entre elles par un ou plusieurs blancs, ce qui indique que les données à lire correspondantes *peuvent* alors être séparées par un nombre quelconque de blancs.

Pour la fonction `scanf`, les blancs dans le format se traduisent donc par l'ordre « à présent, sautez tous les caractères blancs qui se présentent ». Or la seule manière pour l'ordinateur de s'assurer qu'il a bien sauté tous les caractères blancs consiste à lire (sans le consommer) le premier caractère non blanc qui les suit. Si la lecture se fait dans un fichier, cela ne pose pas de problème, les caractères à lire existant tous dès le départ. Mais cela peut créer des situations confuses dans le cas d'une lecture interactive mettant en œuvre le clavier et un opérateur *vivant*. Exemple :

```
int x, y = 33;
...
printf("donne x : ");
scanf(" %d ", &x);      /* aïe... */
printf("donne y : ");
scanf("%d", &y);
printf("x = %d, y = %d\n", x, y);
```

Parce qu'on a écrit, peut-être sans faire attention, des blancs *après* le format `%d`, le premier appel de `scanf` ci-dessus signifie « lisez un entier et tous les blancs qui le suivent ». L'opérateur constatera avec étonnement que pour que son premier nombre sont pris en compte il est obligé de composer le deuxième ! Exemple de dialogue (le signe ¶ représentant la frappe de la touche « Entrée ») :

```
donne x : 11¶
¶
¶            la machine ne réagit pas... !?
22¶
donne y : x = 11, y = 22
```

Notez que faire suivre le premier nombre d'un caractère non blanc conventionnel n'est pas une bonne solution car, ce caractère n'étant pas consommé, il fait échouer la lecture du second nombre :

```
donne x : 11¶
donne y : x = 11, y = 33
```

Leçon à retenir : dans le format de `scanf`, les blancs ne sont pas de la décoration, ils jouent un rôle bien précis.

A PROPOS DE LA LECTURE D'UN CARACTÈRE. Une autre situation génératrice de confusion est celle où des lectures de nombres sont suivies par des lectures de caractères ou de chaînes. Par exemple, le programme naïf suivant lit des nombres les uns à la suite des autres et affiche le carré de chacun :


```
int x, c;
...
do {
    printf("donne un nombre : ");
    scanf("%d", &x);                        ← lecture d'un nombre
    printf("%d ^ 2 = %d\n", x, x * x);
    printf("un autre calcul (O/N) ? ");
    c = getchar();                          ← lecture d'un caractère
}
while (c == 'O');
printf("au revoir...");\end{verbatim}
```

Voici un dialogue (la réplique de l'opérateur est soulignée, le signe ¶ représente la frappe de la touche « Entrée ») :

```
donne un nombre : 14¶
14 ^ 2 = 196
un autre calcul (O/N) ? au revoir...
```

La machine n'a pas attendu la réponse (`O` ou `N`) de l'opérateur... ! Que s'est-il passé ? L'opérateur a composé un nombre suivi immédiatement d'un caractère « retour-chariot » (produit par la touche « Entrée »). Ce retour-chariot a indiqué la fin de la lecture du nombre, mais il n'a pas été consommé et est resté disponible pour des lectures ultérieures. L'appel de `getchar` (qui, dans l'esprit du programmeur, aurait dû provoquer une nouvelle lecture physique) trouve ce retour-chariot et l'affecte à `c` à titre de réponse. On voit que ce programme ne s'exécutera correctement que si l'opérateur compose, à la suite de l'invite `donne un nombre`, un nombre *immédiatement* suivi d'un caractère `O` ou `N` en réponse à une question qui ne lui a pas encore été posée !

La solution de ce problème consiste à vider le tampon d'entrée avant de procéder à une lecture de caractère. Dans beaucoup de situations l'état du tampon est connu, et une manière simple de le vider consiste à le lire par `gets` :

```
int x, c;
char s[BUFSIZ];
...
do {
    printf("donne un nombre : ");
    scanf("%d", &x);
                                        ici, le tampon contient (au moins) un '\n'
    gets(s);
                                        maintenant le tampon est vide
    printf("%d ^ 2 = %d\n", x, x * x);
    printf("un autre calcul (O/N) ? ");
    c = getchar();
}
while (c == 'O');
printf("au revoir...");
```

7.2.5 Les variantes de `printf` et `scanf`

Voici la famille au complet, comme elle est déclarée (en syntaxe ANSI) dans le fichier `<stdio.h>` :

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

Nous avons décrit cette fonction. Ajoutons seulement qu'elle renvoie le nombre de caractères effectivement écrits (on utilise rarement cette information) ou un nombre négatif en cas d'erreur.

`int fprintf(FILE *flot, const char *format, ... )`

Comme `printf` mais sur le flot indiqué (à la place de `stdout`). Ainsi

`printf(` *format* `,` exp_1 `, ... ,` exp_k `)`

équivaut à

`fprintf(stdout,` *format* `,` exp_1 `, ... ,` exp_k `)`

`int sprintf(char *destination, const char *format, ... )`

Cette fonction effectue les mêmes mises en forme que ses deux sœurs, avec la différence que les caractères produits ne sont pas ajoutés à un flot, mais sont concaténés ensemble dans la chaîne destination. Elle

permet donc de dissocier les deux fonctionnalités de printf, transcodage et écriture des données, afin d'utiliser la première sans la deuxième.

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

Nous avons décrit cette fonction. Ajoutons qu'elle renvoie `EOF` si une fin de fichier ou une erreur a empêché toute lecture ; autrement elle rend le nombre de variables lues avec succès. Cette indication fournit une manière bien pratique pour détecter, dans les programmes simples, la fin d'une série de données. Exemple :

```
for(;;) {
    printf("donne un nombre : ");
    if (scanf("%d", &x) < 1)
        break;
    exploitation du nombre lu
}
printf("au revoir...");
```

La boucle précédente sera arrêtée par n'importe quel caractère non blanc ne pouvant pas faire partie d'un nombre.

`int fscanf(FILE *flot, const char *format, ... )`

Comme `scanf` mais sur le flot indiqué (à la place de `stdin`). Ainsi

`scanf(` *format* `,` exp_1 `, ... ,` exp_k `)`

équivaut à

`fscanf(stdin,` *format* `,` exp_1 `, ... ,` exp_k `)`

`int sscanf(const char *source, const char *format, ... )`

Cette fonction effectue la même analyse lexicale que ses sœurs, avec la particularité que le texte analysé n'est pas pris dans un flot, mais dans la chaîne source.

7.3 Opérations en mode binaire

7.3.1 Lecture-écriture

`size_t fread(void *destination, size_t taille, size_t nombre, FILE *flot)`

Cette fonction essaye de lire sur le `flot` indiqué `nombre` objets, chacun ayant la `taille` indiquée, et les copie les uns à la suite des autres dans l'espace pointé par `destination`. Elle renvoie le nombre d'objets effectivement lus, qui peut être inférieur au `nombre` demandé, à cause de la rencontre de la fin du fichier[59], d'une erreur, etc.

`size_t fwrite(const void *source, size_t taille, size_t nombre, FILE *flot)`

Cette fonction écrit les `nombre` objets, chacun ayant la `taille` indiquée, qui se trouvent les uns à la suite des autres à l'adresse indiquée par `source`. Elle renvoie le nombre d'objets écrits, qui peut être inférieur au nombre demandé (en cas d'erreur).

7.3.2 Positionnement dans les fichiers

`int fseek(FILE *flot, long deplacement, int origine)`

Cette fonction positionne le pointeur du fichier associé au `flot` indiqué. La première lecture ou écriture ultérieure se fera à partir de la nouvelle position. Celle-ci est obtenue en ajoutant la valeur de `déplacement` à une valeur de base qui peut être la position courante, la fin ou le début du fichier. C'est l'argument `origine` qui indique de quelle base il s'agit ; la valeur de cet argument doit être une des constantes (définies dans `<stdio.h>`) :

`SEEK_SET` : base = le début du fichier
`SEEK_CUR` : base = la position courante
`SEEK_END` : base = la fin du fichier

La fonction renvoie zéro en cas de succès, une valeur non nulle en cas d'erreur. Sur un fichier de texte, il vaut mieux utiliser `getpos` et `setpos`.

Un appel de cette fonction sur un fichier en écriture produit la vidange du tampon (`fseek` implique `fflush`). Lire ci-après les explications sur l'accès relatif aux fichiers.

[59] Lorsqu'un fichier est lu par des appels de la fonction `fread`, comparer avec zéro le nombre d'objets effectivement lus est la manière la plus pratique de détecter la fin du fichier.

`void rewind(FILE *flot)`

Positionnement au début du fichier du pointeur de fichier associé au flot indiqué. Un appel de cette fonction équivaut à `fseek(flot, 0L, SEEK_SET)` suivi de la mise à zéro de tous les indicateurs d'erreur.

`void fgetpos(FILE *flot, fpos_t *ptr)`

Place dans `ptr` la position courante dans le `flot` indiqué en vue de son utilisation par `setpos`. Le type `fpos_t` (défini dans `<stdio.h>`) a une taille suffisante pour contenir une telle information.

`void fsetpos(FILE *flot, const fpos_t *ptr)`

La valeur de `ptr` provenant d'un appel de `getpos`, cette fonction positionne le flot indiqué à l'emplacement correspondant.

L'ACCÈS RELATIF AUX ÉLÉMENTS DES FICHIERS. La principale application des fonctions de positionnement est la réalisation de ce qu'on appelle l'*accès relatif*[60] ou parfois *accès direct* aux composantes d'un fichier : l'accès à une composante (on dit un *enregistrement*) que ce soit pour le lire ou pour l'écrire, à partir de la donnée de son rang dans le fichier sans être obligé d'accéder au préalable aux enregistrements qui le précèdent.

Pour donner une signification utilisable à la notion de « n^{eme} article », il faut que le fichier puisse être vu comme une suite d'articles de même taille (mais la bibliothèque C ignorera ce fait ; le fichier restera une suite d'octets). Cela s'obtient généralement en déclarant une structure qui représente un article, selon le modèle :

```
typedef struct {
    ...
    champs dont se composent tous les articles
    ...
} ARTICLE;...
```

Un fichier destiné à être utilisé en accès relatif sera déclaré normalement, et ouvert de façon à autoriser les lectures et les écritures :

```
FILE *fichier;
...
fichier = fopen(nom , "rb+");
```

Si les articles sont numérotés à partir de zéro, l'opération « lecture de l'enregistrement de rang n » se programme alors selon le schéma suivant :

```
ARTICLE article;
...
fseek(fichier, n * sizeof(ARTICLE), SEEK_SET);
fread( & article, sizeof(ARTICLE), 1, fichier);
...
exploitation des valeurs des champs de article
...
```

L'opération « écriture de la composante de rang n » obéit au schéma

```
ARTICLE article;
...
affectation des champs de article
...
fseek(fichier, n * sizeof(ARTICLE), SEEK_SET);
fwrite( & article, sizeof(ARTICLE), 1, fichier);
...
```

ATTENTION. La possibilité d'effectuer des lectures et des écritures sur le même fichier pose des problèmes dans la gestion des tampons associés au fichier *que les fonctions de la bibliothèque standard ne résolvent pas.* Ainsi chaque fois qu'une série de lectures sur un fichier va être suivie d'une série d'écritures sur le même fichier, ou réciproquement, il appartient au programmeur de prévoir l'écriture effective du tampon, soit par un appel explicite de la fonction `fflush`, soit par un appel d'une fonction de positionnement comme `fseek`. Notez que cette contrainte est satisfaite si on pratique des accès relatifs selon les schémas indiqués ci-dessus, puisqu'il y a toujours un appel de `fseek` entre deux accès au fichier, quel que soit leur mode.

[60] L'accès relatif s'oppose à l'*accès séquentiel*, dans lequel une composante ne peut être lue ou écrite autrement qu'à la suite de la lecture ou de l'écriture de la composante qui la précède.

7.4 Exemples

Les programmes suivants sont assez naïfs ; ils se proposent d'illustrer les fonctions de traitement de fichiers, aussi bien pour les fichiers de texte que pour les fichiers binaires.

7.4.1 Fichiers "en vrac"

Le programme suivant fait une copie identique d'un fichier (ce programme n'est pas bien utile, généralement une fonction du système d'exploitation fait ce travail) :

```
#include <stdio.h>

#define PAS_D_ERREUR      0   /* codes conventionnels */
#define ERREUR_OUVERTURE  1   /* à l'usage du système */
#define ERREUR_CREATION   2   /* d'exploitation */

FILE *srce, *dest;

main() {
    char tampon[512];
    int nombre;

    printf("source : ");
    gets(tampon);
    srce = fopen(tampon, "rb");
    if (srce == NULL)
        return ERREUR_OUVERTURE;

    printf("destination : ");
    gets(tampon);
    dest = fopen(tampon, "wb");
    if (dest == NULL)
        return ERREUR_CREATION;

    while ((nombre = fread(tampon, 1, 512, srce)) > 0)
        fwrite(tampon, 1, nombre, dest);

    fclose(dest);
    fclose(srce);
    return PAS_D_ERREUR;
}
```

L'essentiel de ce programme est sa boucle `while`. Elle se compose d'un appel de `fread` demandant la lecture de `512` octets, suivi d'un appel de `fwrite` pour écrire les `nombre` octets effectivement lus. Supposons que le fichier à copier comporte `N` octets ; ces deux opérations seront exécutées $\frac{N}{512}$ fois avec `nombre` égal à `512` puis (sauf si `N` est multiple de `512`) une dernière fois avec `nombre` égal au reste de la division de `N` par `512`.

7.4.2 Fichiers binaires et fichiers de texte

L'objet du programme suivant est la constitution d'un fichier d'articles à partir d'un fichier de texte qui se présente comme une répétition du groupe suivant :

- un nom et un prénom sur la même ligne ;
- une adresse sur la ligne suivante ;
- un entier seul sur une ligne, exprimant un certain « nombre de passages » (à un péage autoroutier).

Exemple

```
TOMBAL Pierre
Rue de la Gaiete de Vivre
  10
```



```
MENSOIF Gerard
Allee 'Les verts'
  20
    etc.
```

Le programme est indépendant de la nature du fichier de texte, qui peut être aussi bien le clavier d'un terminal qu'un fichier existant par ailleurs.

```
/* constitution d'un fichier d'articles */
/* à partir d'un fichier de texte */

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

#define PAS_D_ERREUR      0
#define ERREUR_OUVERTURE  1
#define ERREUR_CREATION   2
#define ERREUR_ECRITURE   3
#define ERREUR_FERMETURE  4

struct                              /* un article du fichier */
    {
    char nom[32], prenom[32];
    char adresse[80];
    int nombre_de_passages;
    } art;

FILE *srce;                             /* fichier de texte */
FILE *dest;                             /* fichier binaire */

main() {
    char nom[256];

    printf("source : ");
    gets(nom);
    if ((srce = fopen(nom, "r")) == NULL)
        exit(ERREUR_OUVERTURE);
    printf("destination : ");
    gets(nom);
    if ((dest = fopen(nom, "wb")) == NULL)
        exit(ERREUR_CREATION);

    for (;;) {
        if (fscanf(srce, " %s %s ", art.nom, art.prenom) != 2)
            break;
        fgets(art.adresse, 80, srce);
                                        /* enlevons le '\n' : */
        art.adresse[strlen(art.adresse) - 1] = '\0';
        fscanf(srce, " %d ", &art.nombre_de_passages);

        if (fwrite(&art, sizeof art, 1, dest) != 1)
            exit(ERREUR_ECRITURE);
    }
    if (fclose(dest) != 0)
        exit(ERREUR_ECRITURE);
    exit(PAS_D_ERREUR);
}
```

Notez les diverses fonctions qui ont été utilisées pour lire du texte. Le nombre de passages est lu par `scanf`, car cette fonction est la seule qui convertit les nombres. Le nom et le prénom sont lus par `scanf` aussi, car cela nous arrange bien que les blancs soient pris pour des séparateurs et enlevés. Mais l'adresse est lue par `gets`, car elle peut contenir des blancs qui ne doivent pas être supprimés.

7.4.3 Fichiers en accès relatif

Dans le programme précédent nous avons vu un fichier binaire organisé en articles et *traité séquentiellement* (chaque article étant écrit à la suite du précédent). Voyons un exemple où ce même fichier est traité en accès relatif.

Imaginons que le fichier créé à l'exemple précédent soit le fichier des abonnés à un certain péage, et que nous disposions par ailleurs d'un fichier binaire contenant les numéros des abonnés qui ont emprunté ce péage durant une période donnée. On nous demande d'écrire un programme pour incrémenter la valeur du champ `nbr_passages` de chaque abonné concerné.

```
#include <stdio.h>

#define PAS_D_ERREUR              0
#define ERREUR_OUVERTURE_ABONNES  1
#define ERREUR_OUVERTURE_PASSAGES 2

struct
    {
    char nom[32], prenom[32];
    char adresse[80];
    int nombre_de_passages;
    } art;

FILE *passages, *abonnes;

main() {
    char nom[256];
    long n;

    printf("fichier des passages : ");
    if ((passages = fopen(gets(nom), "rb")) == NULL)
        exit(ERREUR_OUVERTURE_PASSAGES);
    printf("fichier des abonnes : ");
    if ((abonnes = fopen(gets(nom), "rb+")) == NULL)
        exit(ERREUR_OUVERTURE_ABONNES);

    for (;;) {
        if (fread(&n, sizeof n, 1, passages) != 1)
            break;
        fseek(abonnes, n * sizeof art, SEEK_SET);
        fread(&art, sizeof art, 1, abonnes);
        art.nombre_de_passages++;
        fseek(abonnes, n * sizeof art, SEEK_SET);
        fwrite(&art, sizeof art, 1, abonnes);
        }

    fclose(abonnes);
    exit(PAS_D_ERREUR);
}
```

7.5 Les fichiers de bas niveau d'UNIX

Les *fichiers de bas niveau* du système UNIX ne sont pas fondamentalement distincts des flots que l'on vient de décrire. En fait, les flots et les fichiers de bas niveau sont deux manières de voir depuis un programme les mêmes entités de base (les fichiers). Les fichiers de bas niveau sont la manière habituelle de réaliser les fichiers binaires lorsqu'on ne possède pas un C ANSI.

Dans le système UNIX chaque processus dispose, pour gérer ses fichiers, d'une *table de descripteurs* de fichiers. Lorsque les fichiers sont gérés au plus bas niveau, ils sont représentés tout simplement par un indice dans cette table. Par exemple, les trois unités standard `stdin`, `stdout` et `stderr` sont respectivement représentées en tant que fichiers par les descripteurs d'indices `0`, `1` et `2`.

Les fonctions disponibles sont :

`int open(char *nom, int mode, int permissions)`

Ouvre un fichier existant ayant le nom externe indiqué. L'argument `mode` doit avoir pour valeur une des constantes symboliques (définies dans `<ioctl.h>` ou dans `<sys/file.h>`) :

`O_RDONLY` : ouverture en lecture seulement
`O_WRONLY` : ouverture en écriture seulement
`O_RDWR` : ouverture en lecture / écriture

Cette fonction renvoie le numéro du descripteur ouvert pour le fichier, ou un nombre négatif en cas d'erreur. Pour l'argument `permissions` voir ci-dessous.

`int creat(char *nom, int permissions)`

Crée un fichier nouveau, ayant le `nom` indiqué, et l'ouvre en écriture. Comme pour la fonction `open`, l'argument `permissions` indique que le droit de lecture, le droit d'écriture ou le droit d'exécution est accordé ou non au propriétaire du fichier, aux membres de son groupe de travail ou à tous les utilisateurs du système. Cela fait trois groupes de trois bits, soit un nombre qui s'écrit particulièrement bien en octal. Par exemple :

```
fic = creat("tmp/monfic", 0751);
```

crée un fichier nouveau et lui accorde les permissions :

	lecture	écriture	exécution	en octal
propriétaire	1	1	1	7
groupe	1	0	1	5
autres	0	0	1	1

`int read(int fichier, char *adresse, int nombre)`

Lit nombre octets depuis le `fichier` indiqué et les range à partir de l'`adresse` indiquée. Renvoie le nombre d'octets effectivement lus, qui peut être inférieur à `nombre` (fin de fichier, erreur de lecture, etc.).

`int write(int fichier, void *adresse, int nombre)`

Ecrit les `nombre` octets qui se trouvent à l'`adresse` indiquée dans le `fichier` indiqué. Rend le nombre d'octets effectivement écrits, qui peut être inférieur à `nombre` (erreur d'écriture, etc.).

`int close(int descfic)`

Ferme le fichier indiqué.

`long lseek(int fichier, long rang, int origine)`

Positionne le fichier indiqué sur l'octet ayant le rang indiqué. Ce rang exprime une position...
...relative au début du fichier (si `origine` = 0)
...relative à la position courante (si `origine` = 1)
...relative à la fin du fichier (si `origine` = 2)

EXEMPLE. Voici la version UNIX (et, de plus, non ANSI) du programme, donné plus haut, qui effectue une copie identique d'un fichier quelconque. Au sujet de l'utilisation qui est faite des arguments de `main` voir la section 8.3.1.

```
main(int argc, char *argv[]) {
    int srce, dest, n;
    char tampon[512];
    if (argc < 3
            || (srce = open(argv[1], 0, 0777)) < 0
            || (dest = creat(argv[2], 0777)) < 0)
        return 1;
    while ((n = read(srce, tampon, 512)) > 0)
        write(dest, tampon, n);
    close(dest);
    close(srce);
    return 0;
}
```


8 Autres éléments du langage C

Cette section regroupe un ensemble de notions ayant en commun que leur intérêt apparaît surtout à l'occasion de l'écriture de gros programmes. Ce sont des éléments importants du langage mais, sauf quelques exceptions, on peut s'en passer aussi longtemps qu'on limite la pratique de C à des exercices de petite taille.

8.1 Le préprocesseur

Le préprocesseur transforme le texte source d'un programme avant que la compilation ne commence. Généralement associées au processus de la compilation, ses actions semblent en faire partie, mais il faut savoir que le préprocesseur est un programme séparé, largement indépendant du compilateur. En particulier le préprocesseur ne connaît pas la syntaxe du langage C. Les transformations qu'il effectue sur un programme sont simplement des ajouts, des suppressions et des remplacements de morceaux de texte nullement astreints à correspondre aux entités syntaxiques du langage.

Les directives destinées au préprocesseur comportent, dans l'ordre :
- un signe `#` (qui, en syntaxe originale, doit occuper la première position de la ligne) ;
- un mot-clé parmi `include`, `define`, `if`, `ifdef`, `ifndef`, `else` et `endif` qui identifie la directive. Nous passerons sous silence quelques autres directives mineures comme `pragma`, `line` ou `error` ;
- le corps de la directive, dont la structure dépend de la nature de celle-ci.

8.1.1 Inclusion de fichiers

La possibilité d'inclure des fichiers sources dans d'autres fichiers sources s'avère très utile, par exemple pour insérer dans chacun des fichiers séparés qui constituent un programme l'ensemble de leurs déclarations communes (types, structures, etc.). Ainsi ces définitions sont écrites dans un seul fichier, ce qui en facilite la maintenance.

Un cas particulier très important de ce partage des déclarations est celui des bibliothèques livrées avec le compilateur. Elles sont essentiellement composées de fonctions utilitaires déjà compilées dont seul le fichier objet est fourni. Pour que les appels de ces fonctions puissent être correctement compilés[61] dans des programmes qui les utilisent, un certain nombre de *fichiers en-tête* sont livrés avec les bibliothèques, comprenant
- principalement, les prototypes des fonctions qui forment la bibliothèque ou qui incarnent une fonctionnalité particulière (les entrées - sorties, la gestion des chaînes de caractères, la bibliothèque mathématique, etc.) ;
- souvent, un ensemble de directives `#define` et de déclarations `struct`, union et `typedef` qui définissent les constantes et les types nécessaires pour utiliser la bibliothèque en question ;
- parfois, quelques directives `#include` concernant d'autres fichiers en-tête lorsque les éléments définis dans le fichier en-tête en question ne peuvent eux-mêmes être compris sans ces autres fichiers ;
- plus rarement, quelques déclarations de variables externes.

Cette directive possède deux formes :

`#include "`*nom-de-fichier*`"`

C'est la forme générale. Le fichier spécifié est inséré à l'endroit où la directive figure avant que la compilation ne commence ; ce doit être un fichier source écrit en C. La chaîne de caractères `"`*nom-de-fichier*`"` doit être un nom complet et correct pour le système d'exploitation utilisé.

La deuxième forme est :

`#include <`*nom-de-fichier*`>`

Ici, le nom est incomplet ; il ne mentionne pas le chemin d'accès au fichier, car il est convenu qu'il s'agit d'un fichier appartenant au système, faisant partie de la bibliothèque standard : le préprocesseur « sait » dans quels répertoires ces fichiers sont rangés. Ce sont des fichiers dont le nom se termine par `.h` (pour *header*, en-tête) et qui contiennent les déclarations nécessaires pour qu'un programme puisse utiliser correctement les fonctions de la bibliothèque standard.

Exemples (avec des noms de fichier en syntaxe UNIX) :

```
#include <stdio.h>
#include "/users/henri/projet_grandiose/gadgets.h"
```

[61] Notez bien qu'il ne s'agit pas de fournir les fonctions de la bibliothèque (cela est fait par l'apport du fichier objet au moment de l'édition de liens) mais uniquement de donner l'information requise pour que les appels de ces fonctions puissent être correctement traduits par le compilateur. Pour cette raison on ne trouve jamais dans les fichiers en-tête ni des variables non externes, ni le corps des fonctions.

ATTENTION. Il ne faut pas confondre *inclusion de fichiers* et *compilation séparée*. Les fichiers que l'on inclut au moyen de la directive `#include` contiennent du texte source C qui sera compilé chaque fois que l'inclusion sera faite (c'est pourquoi ces fichiers ne contiennent jamais de fonctions). Le service rendu par la directive `#include` est de permettre d'avoir un texte en un seul exemplaire, non de permettre de le compiler une seule fois.

8.1.2 Définition et appel des "macros"

Les macros sans argument[62] se définissent par la directive :

`#define` *nom corps*

Le *nom* de la macro est un identificateur. Le corps de la macro s'étend jusqu'à la fin de la ligne[63]. Partout dans le texte où le nom de la macro apparaît en qualité d'identificateur (ce qui exclut les commentaires, les chaînes de caractères et les occurrences où le nom de la macro est collé à un autre identificateur), il sera remplacé par le *corps* de la macro, lequel est un texte quelconque n'ayant à obéir à aucune syntaxe. Par exemple, entre l'endroit où figurent les deux directives suivantes :

```
#define NBLIGNES 15
#define NBCOLONNES (2 * NBLIGNES)
```

et la fin du fichier où elles apparaissent, chaque occurrence de l'identificateur `NBLIGNES` sera remplacée par le texte `15` et chaque occurrence de l'identificateur `NBCOLONNES` par le texte `(2 * 15)`. Au moins d'un point de vue logique, ces remplacements se feront *avant* que la compilation ne commence. Ainsi, à la place de

```
double matrice[NBLIGNES][NBCOLONNES];
```

le compilateur trouvera

```
double matrice[15][(2 * 15)];
```

(il remplacera immédiatement `2 * 15` par `30`).

ATTENTION. Voici une erreur que l'on peut faire :

```
#define NBLIGNES 15;
#define NBCOLONNES (2 * NBLIGNES);
```

C'est une erreur difficile à déceler, car le compilateur ne signalera ici rien de particulier. Mais plus loin il donnera pour erronée une déclaration comme :

```
double matrice[NBLIGNES][NBCOLONNES];
```

En effet, cette expression qui paraît sans défaut aura été transformée par le préprocesseur en[64]

```
double matrice[15;][(2 * 15;);];
```

MACROS AVEC ARGUMENTS. Elles se définissent par des expressions de la forme :

`#define` *nom*`(`*ident*$_1$`,` ... *ident*$_k$`)` *corps*

Il ne doit pas y avoir de blanc entre le *nom* de la macro et la parenthèse ouvrante. Comme précédemment, le corps de la macro s'étend jusqu'à la fin de la ligne. Les identificateurs entre parenthèses sont les arguments de la macro ; ils apparaissent aussi dans le *corps* de celle-ci. Les utilisations (on dit les *appels*) de la macro se font sous la forme

nom`(`*texte*$_1$`,` ... *texte*$_k$`)`

Cette expression sera remplacée par le corps de la macro, dans lequel *ident*$_1$ aura été remplacé par *texte*$_1$, *ident*$_2$ par *texte*$_2$, etc. Exemple[65] :

```
#define permuter(a, b, type) { type w_; w_ = a; a = b; b = w_; }
```

Exemple d'appel :

```
permuter(t[i], t[j], short *)
```

Résultat de la substitution (on dit aussi *développement*) de la macro :

```
{ short * w_; w_ = t[i]; t[i] = t[j]; t[j] = w_; }
```

[62] Le mot *macro* est une abréviation de l'expression « macro-instruction » désignant un mécanisme qui existe depuis longtemps dans beaucoup d'assembleurs. Les macros avec arguments sont appelées aussi *pseudo-fonctions*.

[63] Mais il découle de la section 1.2.1 que le corps d'une macro peut occuper en fait plusieurs lignes : il suffit que chacune d'elles, sauf la dernière, se termine par le caractère \.

[64] Pour permettre de trouver ce genre d'erreurs, certains compilateurs C possèdent une option (sous UNIX l'option `-E`) qui produit l'affichage du programme source uniquement transformé par le préprocesseur.

[65] Dans cet exemple, les accolades { et } donnent au corps de la macro une structure de bloc qui sera comprise et exploitée par le compilateur, mais cette structure est parfaitement indifférente au préprocesseur.

Les appels de macros ressemblent beaucoup aux appels de fonctions. La même facilité, ne pas récrire un certain code source, peut être obtenue avec une macro et avec une fonction. Mais il faut comprendre que ce n'est pas du tout la même chose. Alors que le corps d'une fonction figure en un seul exemplaire dans le programme exécutable dont elle fait partie, le corps d'une macro est *recopié* puis compilé à chaque endroit où figure l'appel de la macro.

Il faut donc que le corps d'une macro soit réduit, sans quoi son utilisation peut finir par être très onéreuse en termes d'espace occupé. Un autre inconvénient des macros est que leur définition « hors syntaxe » les rend très difficiles à maîtriser au-delà d'un certain degré de complexité (pas de variables locales, etc.).

En faveur des macros : leur efficacité. Le texte de la macro étant recopié à la place de l'appel, on gagne à chaque fois le coût d'un appel de fonction. Il eût été bien peu efficace d'appeler une fonction pour ne faire que la permutation de deux variables ! Autre intérêt des macros, elles sont indépendantes des types de leurs arguments ou même, comme ci-dessus, elles peuvent avoir un type pour argument.

En définitive, la principale utilité des macros est d'améliorer l'expressivité et la portabilité des programmes en centralisant la définition d'opérations compliquées, proches du matériel ou sujettes à modification. Voici un exemple élémentaire : en C original (dans lequel on ne dispose pas du type `void *`) la fonction `malloc` doit être généralement utilisée en association avec un changement de type :

```
p = (machin *) malloc(sizeof(machin));
```

Des telles expressions peuvent figurer à de nombreux endroits d'un programme. Si nous définissons la macro

```
#define NOUVEAU(type) ((type *) malloc(sizeof(type)))
```

alors les allocations d'espace mémoire s'écriront de manière bien plus simple et expressive

```
p = NOUVEAU(machin);
```

De plus, un éventuel remplacement ultérieur de `malloc` par un autre procédé d'obtention de mémoire sera facile et fiable, puisqu'il n'y aura qu'un point du programme à modifier.

Attention. 1. Il est quasiment obligatoire d'écrire les arguments et le corps des macros entre parenthèses, pour éviter des problèmes liés aux priorités des opérateurs. Par exemple, la macro

```
#define TVA(p) 0.196 * p
```

est très imprudente, car une expression comme `(int) TVA(x)` sera développée en `(int) 0.196 * x`, ce qui vaut *toujours* `0`. Première correction :

```
#define TVA(p) (0.186 * p)
```

Le problème précédent est corrigé, mais il y en a un autre : l'expression `TVA(p1 + p2)` est développée en `(0.186 * p1 + p2)`, ce qui est probablement erroné. D'où la forme habituelle :

```
#define TVA(p) (0.186 * (p))
```

2. *Il ne faut jamais appeler une macro inconnue avec des arguments qui ont un effet de bord*, car ceux-ci peuvent être évalués plus d'une fois. Exemple classique : on dispose d'une macro nommée `toupper` qui transforme toute lettre minuscule en la majuscule correspondante, et ne fait rien aux autres caractères. Elle pourrait être ainsi définie (mais on n'est pas censé le savoir) :

```
#define toupper(c) ('a' <= (c) && (c) <= 'z' ? (c) + ('A' - 'a') : (c))
```

On voit que l'évaluation de `toupper(c)` comporte au moins deux évaluations de `c`. Voici un très mauvais appel de cette macro :

```
calu = toupper(getchar())
```

En effet, cette expression se développe en

```
calu = ('a' <= (getchar()) && (getchar()) <= 'z'
                                          ? (getchar()) + ('A' - 'a') : (getchar()));
```

A chaque appel, au moins deux caractères sont lus, ce qui n'est sûrement pas l'effet recherché. On notera que si `toupper` avait été une fonction, l'expression `toupper(getchar())` aurait été tout à fait correcte. Mais puisque ce n'en est pas une, il faut l'appeler avec un argument sans effet de bord. Par exemple :

```
{ calu = getchar(); calu = toupper(calu); }
```

8.1.3 Compilation conditionnelle

Les directives de conditionnelle (compilation) se présentent sous les formes suivantes :


```
{ #if expression          }
{ #ifdef identificateur   }
{ #ifndef identificateur  }
    ...
    texte compilé si la condition est vraie, ignoré si elle est fausse
    ...
#else
    ...
    texte compilé si la condition est fausse, ignoré si elle est vraie
    ...
#endif
```

La partie else est facultative ; on a donc droit aussi aux formes :

```
{ #if expression          }
{ #ifdef identificateur   }
{ #ifndef identificateur  }
    ...
    texte compilé si la condition est vraie, ignoré si elle est fausse
    ...
#endif
```

Lorsque le premier élément de cette construction est `#if` *expression*, *expression* doit pouvoir être évaluée par le préprocesseur. Ce doit donc être une expression constante ne contenant pas l'opérateur de changement de type ni les opérateurs `sizeof` ou `&`. Elle sera évaluée et

- si elle est vraie, c'est-à-dire non nulle, alors le texte qui se trouve entre `#if` et `#else` sera traité par le compilateur, tandis que celui qui figure entre `#else` et `#endif` sera purement et simplement tenu pour inexistant ;
- si elle est fausse, c'est-à-dire nulle, c'est le premier texte qui sera ignoré tandis que le second sera lu par le compilateur. Dans la forme sans `#else`, si la condition est fausse aucun texte ne sera compilé.

Les directives

`#ifdef` *identificateur*
`#ifndef` *identificateur*

équivalent respectivement à

`#if` « *identificateur* est le nom d'une macro actuellement définie »
`#if` « *identificateur* n'est pas le nom d'une macro actuellement définie »

La compilation conditionnelle se révèle être un outil précieux pour contrôler les parties des programmes qui dépendent de la machine ou du système sous-jacent. On obtient de la sorte des textes sources qui restent portables malgré le fait qu'ils contiennent des éléments non portables. Voici un exemple : les fonctions de lecture-écriture sur des fichiers binaires ne sont pas les mêmes dans la bibliothèque ANSI et dans la bibliothèque UNIX classique. Une manière d'écrire un programme indépendant de ce fait consiste à enfermer les appels de ces fonctions et les déclarations qui s'y rapportent dans des séquences conditionnelles. Déclaration :

```
...
#ifdef BINAIRES_ANSI
    FILE *fic;
#else
    int fic;
#endif
...
```

Ouverture :

```
...
#ifdef BINAIRES_ANSI
    fic = fopen(nom, "r");
    ok = (fic != NULL);
#else
    fic = open(nom, 0);
    ok = (fic >= 0);
#endif
...
```


Lecture :

```
...
#ifndef BINAIRES_ANSI
    ok = (fread(&art, sizeof(art), 1, fic) == 1);
#else
    ok = (read(fic, &art, sizeof(art)) == sizeof(art));
#endif
...
```

L'emploi de la compilation conditionnelle associée à la définition de noms de macros est rendu encore plus commode par le fait qu'on peut utiliser

- des noms de macros normalement définis (`#define...`) ;
- des noms de macros définis dans la commande qui lance la compilation, ce qui permet de changer le texte qui sera compilé sans toucher au fichier source. De plus, puisque ces définitions se font au niveau de la commande de compilation, elles pourront être faites par des procédures de commandes (scripts) donc automatisées ;
- un ensemble de noms de macros prédéfinis dans chaque système, pour le caractériser. Par exemple dans le compilateur des systèmes UNIX la macro `UNIX` est définie et vaut `1`.

REMARQUE. Le préprocesseur du C ANSI offre également l'opérateur `defined` qui permet d'écrire les directives `#ifdef` *ident* (resp. `#ifndef` *ident*) sous la forme `#if defined` *ident* (resp. `#if !defined` *ident*). D'autre part, la construction

```
#if expression1
...
#else
#if expression2
...
#endif
#endif
```

peut en C ANSI s'abréger en

```
#if expression1
...
#elif expression2
...
#endif
```

8.2 La modularité de C

A ce point de notre exposé nous pouvons nous poser une question importante : le langage C est-il modulaire ? La modularité est une qualité que les bonnes méthodes de conception doivent posséder. Elle n'est pas facile à définir ; on peut toutefois la cerner à travers quelques critères[66]. Une méthode de conception doit

- aider à diviser chaque nouveau problème en sous-problèmes qu'on peut résoudre séparément (critère de *décomposabilité*) ;
- favoriser la production de composants logiciels qui peuvent se combiner librement pour produire de nouveaux systèmes (critère de *composabilité*) ;
- permettre au concepteur d'écrire des modules dont chacun peut être compris isolément par un lecteur humain (critère de *compréhensibilité*) ;
- permettre qu'une petite modification des spécifications du problème entraîne uniquement la modification d'un petit nombre des modules de la solution (critère de *continuité*) ;
- assurer que l'effet d'une condition anormale se produisant dans un module restera localisé à ce module ou, au pire, n'atteindra qu'un petit nombre de modules « voisins » (critère de *protection*).

Théoriquement, tout langage de programmation peut servir à mettre en œuvre une méthode de conception modulaire, mais dans certains cas il faut une telle dose de ruse et d'autodiscipline, pour un résultat si peu fiable, que le jeu n'en vaut pas la chandelle. En pratique un langage est dit modulaire lorsqu'on peut, sans douleur et sans artifice, en faire l'outil d'une méthode de conception modulaire. L'expérience montre que les critères précédents induisent sur le langage en question quelques contraintes assez précises, comme celles-ci :

- les modules doivent correspondre à des entités syntaxiques du langage ;

[66] Suivant l'analyse de Bertrand Meyer (Conception et programmation par objets, InterEditions, 1990) à laquelle il semble difficile d'ajouter ou de retrancher quelque chose.

– chaque module doit partager des informations avec aussi peu d'autres modules que possible, et quand un tel partage existe il doit concerner aussi peu d'éléments que possible ;
– quand deux modules partagent des informations, cela doit être clairement indiqué dans leurs deux textes.

A la lumière de ces principes la modularité de C apparaît fort rudimentaire, pour ne pas dire inexistante. Si l'on prend pour modules les fichiers sources qui composent un programme, on constate qu'aucune structure syntaxique ne signale les modules ni n'en délimite la portée, d'autant plus que le caractère non syntaxique de la directive `#include` brouille l'organisation du programme en fichiers distincts. Aucune déclaration particulière n'est requise pour indiquer que des objets sont partagés entre plusieurs modules. Chaque module communique avec *tous* les autres et, sauf spécification contraire, *tous* les objets de chaque module sont partagés.

Bien sûr, la compilation séparée est une des idées-clés du langage, et il est possible de rendre inaccessibles les noms qui peuvent être privés. Mais le langage offre peu d'outils pour rendre fiable le partage des noms qui doivent être publics, et ces outils restent d'un emploi facultatif, subordonné à l'autodiscipline du programmeur. Par exemple, si dans un module B on doit référencer une variable ou une fonction définie dans un module A, il suffit d'écrire *dans* B une déclaration comme `extern int x ;`. Cet énoncé postule l'existence d'un objet nommé `x`, ce qui sera contrôlé par l'éditeur de liens. Mais il donne à `x` des attributs (la classe variable, le type `int`) que l'objet désigné par `x` ne possède pas forcément ; aucune vérification cependant ne sera faite. Ainsi la compilation de B se fait dans la plus totale insécurité.

8.2.1 Fichiers en-tête

Le seul moyen dont dispose l'auteur d'un module A pour s'assurer que les autres modules qui forment un programme utilisent correctement les variables et fonctions qu'il rend publiques consiste à écrire un fichier en-tête (fichier `A.h`) contenant toutes les déclarations publiques. Ce fichier doit être inclus par la directive `#include` dans le module qui implante les objets publics (fichier `A.c`) et dans chacun des modules qui les utilisent. De cette manière tous ces fichiers « voient » les mêmes définitions de types, les mêmes déclarations de variables et les mêmes prototypes de fonctions ; ces déclarations sont écrites en un seul endroit, et toute modification de l'une d'entre elles se répercute sur tous les fichiers qui en dépendent.

La nécessité de ces fichiers en-tête apparaît encore plus grande quand on considère le cas des bibliothèques, c'est-à-dire des modules que leurs fonctionnalités placent en position de prestataires de services vis-à-vis des autres modules qui composent un programme ; on parle alors de module serveur et de modules clients. En fait, on peut presque toujours voir la modularité en termes de serveurs et clients, car il y a toujours une hiérarchie parmi les modules. Le propre des bibliothèques est d'être conçues de manière indépendante des clients, afin de pouvoir être utilisées dans un programme présent et un nombre quelconque de programmes futurs. L'intérêt de leur associer le meilleur dispositif pour minimiser le risque de mauvaise utilisation est évident.

Typiquement, un fichier en-tête comportera les éléments suivants :
– Des directives `#include` concernant les autres fichiers en-tête nécessaires pour la compréhension (par le compilateur) des éléments qui apparaissent dans le fichier en-tête en question.
– Des définitions de constantes, soit sous forme de directives `#define` soit sous forme de type énuméré, qui sont des informations symboliques échangées entre le serveur et ses clients. Exemple : dans une bibliothèque graphique, les noms conventionnels des couleurs.
– Des définitions de structures (`struct`, `union`) et de types (`typedef`) qui définissent la nature des objets manipulés par la bibliothèque. Typiquement, ces types permettent aux clients de déclarer les objets qui sont les arguments et les résultats des fonctions de la bibliothèque. Exemple : dans une bibliothèque graphique, la définition de types *point*, *segment*, etc.
– Les déclarations extern des variables publiques du serveur. Les définitions correspondantes (sans le qualifieur `extern`) figureront dans le module serveur.
REMARQUE. L'emploi de variables publiques est déconseillé ; un module ne devrait offrir que des fonctions[67].
– Les déclarations des fonctions publiques du serveur. Les définitions correspondantes figureront dans le module serveur. En syntaxe originale seules les déclarations des fonctions qui ne rendent pas un entier sont nécessaires, mais même dans ce cas c'est une bonne habitude que d'y mettre les déclarations de toutes les fonctions, cela constitue un germe de documentation.

Bien entendu, *tous les noms de variables et fonctions du module serveur qui ne figurent pas dans le fichier en-tête doivent être rendus privés* (en les qualifiant `static`).

[67] S'il est utile qu'un client puisse consulter ou modifier une variable du serveur, écrivez une fonction qui ne fait que cela (mais en contrôlant la validité de la consultation ou de la modification). C'est bien plus sûr que de donner libre accès à la variable.

8.2.2 Exemple : `stdio.h`

A titre d'exemple visitons rapidement le plus utilisé des fichiers en-tête de la bibliothèque standard : `stdio.h`. Notre but n'est pas de prolonger ici l'étude des entrées-sorties, mais d'illustrer les indications du paragraphe précédent sur l'écriture des fichiers en-tête. Le texte ci-après est fait de morceaux de la version *MPW* (*Macintosh Programmer Workshop*) du fichier `stdio.h` auquel nous avons enlevé ce qui ne sert pas notre propos :

```
/*  stdIO.h -- Standard C I/O Package
 *  Modified for use with Macintosh C
 *  Apple Computer, Inc.  1985-1988
 *
 *  Copyright American Telephone & Telegraph
 *  Used with permission, Apple Computer Inc. (1985)
 *  All rights reserved.
 */

#ifndef __STDIO__                                    α
#define __STDIO__

#include <stddef.h>
#include <stdarg.h>

/*
 *  Miscellaneous constants
 */
#define EOF     (-1)
#define BUFSIZ   1024
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2

/*
 *  The basic data structure for a stream is the FILE.
 */
typedef struct {                                     β
    int             _cnt;
    unsigned char   *_ptr;
    unsigned char   *_base;
    unsigned char   *_end;
    unsigned short  _size;
    unsigned short  _flag;
    unsigned short  _file;
} FILE;

/*
 *  Things used internally:
 */
extern FILE _iob[];                                  γ
int _filbuf (FILE *);
int _flsbuf (int, FILE *);

/*
 *  The standard predefined streams
 */
#define stdin       (&_iob[0])
#define stdout      (&_iob[1])
#define stderr      (&_iob[2])

/*
 *  File access functions
 */
int fclose (FILE *);                                 δ
int fflush (FILE *);
FILE *fopen (const char *, const char *);
int setvbuf (FILE *, char *, int, size_t);
```

```
/*
 *  Formatted input/output functions
 */
int printf (const char *, ...);                    ε
int fprintf (FILE *, const char *, ...);
int sprintf (char *, const char *, ...);
int scanf (const char *, ...);
int fscanf (FILE *, const char *, ...);
int sscanf (const char *, const char *, ...);

/*
 *  Character input/output functions and macros
 */
int fgetc (FILE *);
int ungetc (int, FILE *);
int fputc (int, FILE *);
char *fgets (char *, int, FILE *);
char *gets (char *);
int puts (const char *);
int fputs (const char *, FILE *);
#define getc(p) (--(p)->_cnt >= 0 \
        ? (int) *(p)->_ptr++ : _filbuf(p))
#define getchar()   getc(stdin)
#define putc(x, p) (--(p)->_cnt >= 0 \
        ? ((int) (*(p)->_ptr++ = (unsigned char) (x))) \
        : _flsbuf((unsigned char) (x), (p)))
#define putchar(x)  putc((x), stdout)

/*
 *  Direct input/output functions
 */
size_t fread (void *, size_t, size_t, FILE *);
size_t fwrite (const void *, size_t, size_t, FILE *);
#endif __STDIO__
```

RENVOIS :

α Lorsque les fichiers en-tête sont d'un intérêt général ils finissent par être inclus dans de nombreux autres fichiers, eux-mêmes inclus les uns dans les autres. Le compilateur risque alors de lire plusieurs fois un même fichier, ce qui entraîne un travail inutile et parfois des erreurs liées à des redéfinitions. Par exemple, imaginons que `MonBazar.h` soit un fichier comportant la directive `#include <stdio.h>`. Dans la compilation d'un fichier commençant par les deux directives

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

le compilateur risquerait de compiler deux fois le fichier `stdio.h`. Les deux premières directives de ce fichier résolvent ce problème : lorsque ce fichier est pris en compte une première fois, le nom `__STDIO__` (nom improbable arbitrairement associé à `stdio.h`) devient défini. Durant la présente compilation, la directive `#ifndef __STDIO__` fera que l'intérieur du fichier ne sera plus jamais lu par le compilateur. Tous les fichiers en-tête peuvent être protégés de cette manière contre les inclusions multiples.

β On peut noter ici que la vue du type `FILE` qu'ont les modules clients est très différente de celle qu'ils auraient en utilisant un langage très modulaire comme Modula II ou Ada. Dans ces langages les noms et les types des champs d'une structure comme `FILE` resteraient privés, ainsi que la structure elle-même ; seul le type *adresse d'une telle structure* serait rendu public. Cela s'appelle un *type opaque* et augmente la fiabilité des programmes.

En C nous n'avons pas d'outils pour procéder de la sorte : ou bien un type est privé, connu uniquement du module serveur, ou bien il faut « tout dire » à son sujet. L'utilisation d'un pointeur générique, dans le style de

```
typedef void *FILE_ADDRESS;
```

dissimulerait bien l'information sur la structure `FILE`, mais aurait la conséquence de rendre impossibles à déceler les mauvaises utilisations du type en question par le client. Notez d'autre part que la connaissance

dans les modules clients des noms des champs de la structure `FILE` est indispensable pour l'utilisation des macros comme `getc` ou `putc`.

γ Tous les noms qui apparaissent dans le fichier en-tête deviennent de ce fait publics, même ceux qu'on aurait aimé garder privés alors qu'on ne le peut pas, par exemple parce qu'ils apparaissent dans le corps d'une macro, comme ici le nom de la table des descripteurs de fichiers `_iob`. Le langage C n'ayant rien prévu à ce effet, la « privacité » n'est assurée que par une convention entre le système et les utilisateurs : tout nom commençant par un blanc souligné « _ » appartient au système et l'utilisateur doit feindre d'en ignorer l'existence.

δ Ceci n'est pas la meilleure manière de donner les prototypes des fonctions dans un fichier en-tête. Dans une déclaration qui n'est pas une définition, la syntaxe n'exige pas les noms des arguments mais uniquement leurs types. Cependant, s'ils sont bien choisis, ces noms apportent une information supplémentaire qui augmente la sécurité d'utilisation de la bibliothèque. Par exemple, les prototypes

```
FILE *fopen(const char *, const char *);
size_t fread(void *, size_t, size_t, FILE *);
```

ne seraient d'aucune utilité à un programmeur qui hésiterait à propos du rôle ou de l'ordre des arguments de ces fonctions, contrairement à leurs versions « équivalentes » :

```
FILE *fopen(const char *filename, const char *mode);
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

ε Les appels de fonctions avec des arguments variables ne bénéficient pas des vérifications syntaxiques que le C ANSI effectue lorsque la fonction a fait l'objet d'une définition de prototype. Sauf pour les arguments nommés (le premier ou les deux premiers), les arguments que l'on passe à l'une de ces six fonctions échappent donc à tout contrôle du compilateur. L'information concernant la nature de ces arguments est portée par le format ; elle ne pourra être exploitée qu'à l'exécution.

Autres remarques. Le premier client de ce dispositif doit être le fournisseur lui-même. Pour que la protection contre l'erreur recherchée soit effective, il faut que chacun des fichiers qui réalisent l'implantation des variables et fonctions « promises » dans `stdio.h` comporte la directive

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

De cette manière on garantit que l'auteur et l'utilisateur de chaque variable ou fonction sont bien d'accord sur la définition de l'entité en question.

Une autre règle à respecter par l'auteur du ou des modules serveurs : qualifier `static` tous les noms qui ne sont pas déclarés dans le fichier en-tête, pour éviter les collisions de noms. Le fichier en-tête joue ainsi, pour ce qui concerne les variables et les fonctions, le rôle de liste officielle des seuls noms publics.

8.3 Deux ou trois choses bien pratiques...

8.3.1 Les arguments du programme principal

Cette section ne concerne que les environnements, comme UNIX ou MS-DOS, dans lesquels les programmes sont activés en composant une commande de la forme *nom-du-programme argument*$_1$ ... *argument*$_k$.

L'exécution d'un programme commence par la fonction `main`. Tout se passe comme si le système d'exploitation avait appelé cette fonction comme une fonction ordinaire. Il faut savoir que lors de cet appel, des arguments sont fournis au programme. Voici l'en-tête complet de `main`, en syntaxe ANSI :

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

avec :

`argc` : nombre d'arguments du programme

`argv` : tableau de chaînes de caractères, qui sont les arguments du programme. Par convention, le premier argument est le nom du programme lui-même.

Imaginons avoir écrit un programme qui, une fois compilé, se nomme echo ; supposons que ce programme soit lancé par la commande :

```
echo Pierre Paul
```

alors, `main` reçoit les arguments que montre la figure 18.

Supposons que tout ce que l'on demande à `echo` soit de recopier la liste de ses arguments. Voici comment on pourrait écrire ce programme :

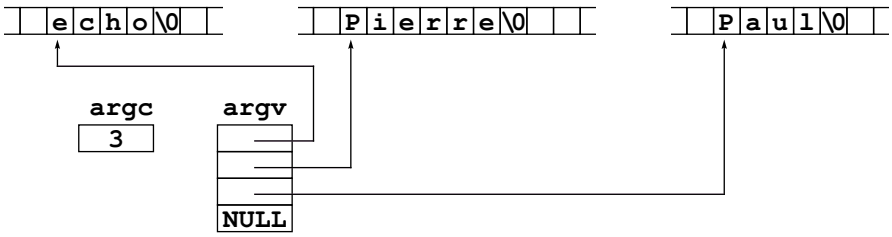

FIG. 18 – Arguments de `main`

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

    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

La frappe de la commande « `echo Pierre Paul` » produit l'affichage de

```
echo
Pierre
Paul
```

La principale application de ce mécanisme est la fourniture à un programme des paramètres dont il peut dépendre, comme des noms de fichiers, des tailles de tableaux, etc. Par exemple, voici une nouvelle version du programme de copie de fichiers donné à la section 7.4.1, qui prend les noms des fichiers source et destination comme arguments :

```
#include <stdio.h>

#define PAS_D_ERREUR            0
#define ERREUR_OUVERTURE        1
#define ERREUR_CREATION         2
#define PAS_ASSEZ_D_ARGUMENTS   3

FILE *srce, *dest;

main(int argc, char *argv[]) {
    char tampon[512];
    int nombre;
    if (argc < 3)
        return PAS_ASSEZ_D_ARGUMENTS;
    if ((srce = fopen(argv[1], "rb")) == NULL)
        return ERREUR_OUVERTURE;
    if ((dest = fopen(argv[2], "wb")) == NULL)
        return ERREUR_CREATION;

    while ((nombre = fread(tampon, 1, 512, srce)) > 0)
        fwrite(tampon, 1, nombre, dest);

    fclose(dest);
    return PAS_D_ERREUR;
}
```

Si nous appelons `copier` le fichier exécutable produit en compilant le texte ci-dessus, alors nous pourrons l'exécuter en composant la commande

`copier` *fichier-source fichier-destination*

8.3.2 Branchements hors fonction : `setjmp.h`

Le mécanisme des longs branchements permet d'obtenir la terminaison immédiate de toutes les fonctions qui ont été appelées (et ne sont pas encore terminées) depuis que le contrôle est passé par un certain point du programme, quel que soit le nombre de ces fonctions. Il est réalisé à l'aide des deux fonctions :

```
int setjmp(jmp_buf contexte); void
longjmp(jmp_buf contexte, int code);
```

Cela fonctionne de la manière suivante : tout d'abord il faut déclarer une variable, généralement globale, de type `jmp_buf` (type défini dans le fichier en-tête `setjmp.h`) :

```
#include <setjmp.h>
...
jmp_buf contexte;
```

L'appel

```
setjmp(contexte);
```

enregistre dans `contexte` certaines informations traduisant l'état du système, puis renvoie zéro. Ensuite, l'appel

```
longjmp(contexte, valeur);
```

remet le système dans l'état qui a été enregistré dans la variable `contexte`. Plus précisément, le système se trouve comme si l'appel de `setjmp(contexte)` venait tout juste de se terminer, en rendant cette fois non pas zéro mais la `valeur` indiquée dans l'appel de `longjmp`.

Le principal service rendu par ce dispositif est de permettre de programmer simplement, et en maîtrisant le « point de chute », l'abandon d'une famille de fonctions qui se sont mutuellement appelées. Examinons un exemple classique : supposons qu'une certaine fonction `expression` soit un analyseur syntaxique présentant deux caractéristiques fréquentes :

- la fonction `expression` est à l'origine d'une imbrication dynamique fort complexe (`expression` appelle une fonction `terme`, qui appelle une fonction `facteur` qui à son tour appelle une fonction `primaire` laquelle rappelle `expression`, etc.) ;
- chacune des fonctions `expression`, `terme`, `facteur`, etc., peut à tout moment rencontrer une erreur dans ses données, qui rend la poursuite du traitement inutile ou impossible.

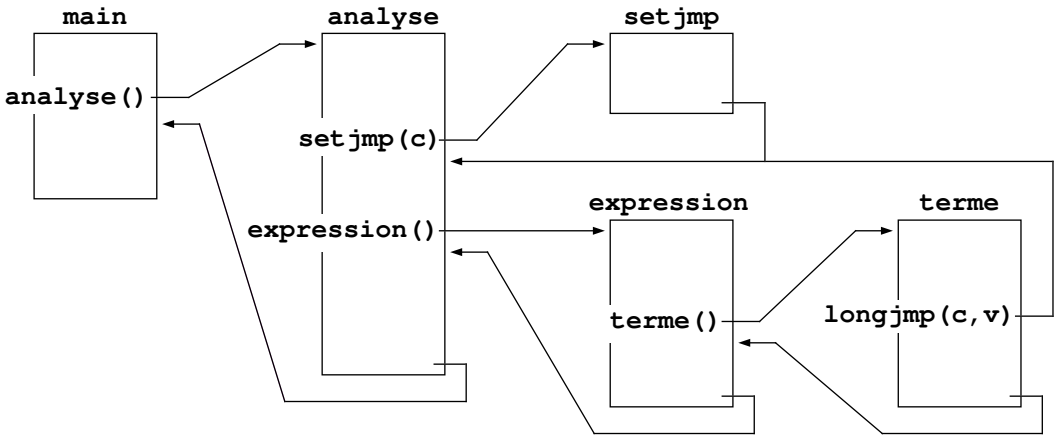

FIG. 19 – Branchement hors fonction

Voici comment l'appel d'expression pourrait être emballé dans une fonction « enveloppe » nommée `analyse` (voir la figure 19) :


```
#include <setjmp.h>
...
jmp_buf contexte;
...
int analyse(void) {
    if (setjmp(contexte) == 0) {
        expression();
        return SUCCES;
    }
    else
        return ERREUR;
}
```

Fonctionnement : lorsque la fonction `analyse` est activée, `setjmp` est appelée et rend `0` ; elle est donc immédiatement suivie par l'appel de `expression`. Si le travail de `expression` se termine normalement, alors analyse rendra `SUCCES` et elle aura été transparente. D'autre part, dans la fonction `expression` et toutes celles appelées « au-dessus » d'elle (`terme`, `facteur`, etc.), il est possible d'effectuer l'appel

```
longjmp(contexte, 1);
```

qui ramène le contrôle dans la fonction `analyse`, exactement sur la partie condition de l'instruction

```
if (setjmp(contexte) == 0)
```

mais cette fois la valeur rendue par `setjmp` sera `1` et `analyse` rendra donc la valeur `ERREUR`.

Remarques. 1. L'appel de `longjmp` se présente comme une remise du système dans l'un des états par lesquels il est passé. Cela est vrai, sauf pour ce qui concerne la valeur des variables, notamment les variables globales : elles ne reprennent pas la valeur qu'elles avaient lorsque `longjmp` a été appelée.

2. On ne peut effectuer un appel de `longjmp` que pour ramener le système dans l'une des fonctions appelées et *non encore terminées*, pour laquelle l'espace local est encore alloué dans la pile. Appeler `longjmp` avec un contexte qui a été mémorisé dans une fonction dont l'activation est terminée est une erreur aux conséquences indéfinies, même si cette fonction a été rappelée depuis. Par exemple, un appel `longjmp(c,v)` qui serait placé dans la fonction main après l'appel `analyse()` (voir la figure 19) serait erroné, car il utiliserait le contexte d'une activation de fonction (la fonction `analyse`) qui n'existe plus.

8.3.3 Interruptions : `signal.h`

L'objet de la fonction `signal` est la détection d'événements asynchrones qui peuvent se produire pendant l'exécution d'un programme. « Asynchrones » signifie qu'on ne peut pas prévoir le moment où ils se produiront, car ils ne résultent pas d'instructions normalement insérées dans la séquence qui forme le programme.

Un certain ensemble de types d'événements, dépendant de chaque système, est récupérable à travers le mécanisme décrit ici. En général on traite de la même manière le vrai et le faux asynchronisme. Une coupure de courant ou une interruption provoquée par l'utilisateur sont des événements vraiment imprévisibles. Une référence à travers un pointeur invalide ou une division par zéro ne sont pas réellement asynchrones (si on avait connu parfaitement le programme et les données on aurait pu prédire très exactement de tels événements), mais il est commode de les considérer comme tels et de les récupérer de la même manière.

La fonction `signal` est déclarée dans le fichier `signal.h` de la façon suivante :

```
void (*signal(int numero, void (*manip)(int)))(int);
```

Ce prototype n'est pas très facile à lire. Définissons le type `PROCEDURE` comme celui d'une fonction sans résultat défini ayant un argument de type int :

```
typedef void PROCEDURE(int);
```

Avec cela, la déclaration précédente se récrit plus simplement :

```
PROCEDURE *signal(int numero, PROCEDURE *manip);
```

et elle nous apprend que la fonction `signal` prend deux arguments, à savoir un `int` et l'adresse d'une `PROCEDURE`, et renvoie l'adresse d'une `PROCEDURE`.

Lors d'un appel de signal, l'argument `numero` doit désigner un des événements qu'il est possible de récupérer. L'effet de `signal` est d'enregistrer la `PROCEDURE` donnée en argument, de telle manière que si l'événement en question se produit, alors elle sera automatiquement appelée par le système avec pour argument le numéro de l'événement. La fonction `signal` rend l'adresse de la `PROCEDURE` qui était jusqu'alors enregistrée pour ce même événement.

Six événements sont prévus par le standard ANSI (ils ne sont pas tous implantés dans tous les systèmes ; en outre, chaque système peut ajouter ses propres particularités) :

`SIGABRT` : fin anormale du programme (appel de la fonction abort)
`SIGINT` : interruption provoquée par l'utilisateur (touche Ctrl-C, etc.)
`SIGTERM` : demande d'arrêt de l'utilisateur (interruption " forte ")
`SIGFPE` : erreur arithmétique (division par zéro, etc.)
`SIGSEGV` : accès mémoire illégal (souvent : mauvais pointeur dans l'accès à une donnée, débordement de tableau)
`SIGILL` : instruction illégale (souvent : mauvais pointeur dans l'appel d'une fonction)

La bibliothèque fournit en outre deux mécanismes de récupération d'une interruption prédéfinis :

`SIG_DFL` : le mécanisme par défaut utilisé par le système pour l'événement en question lorsque la fonction signal n'a pas été appelée
`SIG_IGN` : le mécanisme trivial qui consiste à ignorer l'événement

Exemple. Supposons qu'une section d'un certain programme effectue des calculs très complexes dont la durée risque d'inquiéter l'utilisateur final. On souhaite donc offrir à ce dernier la possibilité d'interrompre le programme lorsqu'il estime que celui-ci devient excessivement long, en appuyant sur une touche particulière[68]. L'utilisateur apprendra alors l'état d'avancement de son calcul (traduit par la valeur de la variable `iteration_numero`) et aura à choisir entre la continuation et la terminaison du programme :

```
#include <signal.h>

int iteration_numero;
typedef void PROCEDURE(int);

void voirSkissPass(int numero) {
    int c;
    printf("J'en suis a l'iteration: %d\nOn arrete? ", iteration_numero);
    do
        c = getchar();
    while ((c = toupper(c)) != 'O' && c != 'N');
    if (c == 'O')
        exit(1);        /* abandonner le programme */
    else
        return;         /* reprendre le travail interrompu */
}

main() {
    PROCEDURE *maniprec;
    ...
    autres opérations
    ...
/* entrée dans la section onéreuse */
    maniprec = signal(SIGINT, voirSkissPass);
    ...
    calculs terriblement complexes
    ...
/* sortie de la section onéreuse */
    signal(SIGINT, maniprec);
    ...
    autres opérations
    ...
}
```

8.4 La bibliothèque standard

La bibliothèque standard ANSI se compose de deux sortes d'éléments :
– un ensemble de *fichiers objets*[69] contenant le code compilé des fonctions de la bibliothèque et participant,

[68] Sur plusieurs systèmes, comme UNIX et VMS, il s'agit de la combinaison des deux touches « Ctrl » et « C ».
[69] Il s'agit en réalité de fichiers « bibliothèques » (extensions `.a`, `.lib`, `.dll`, etc.), mais de tels fichiers ne sont que des fichiers objets un peu arrangés pour en faciliter l'emploi par les éditeurs de liens.

pour la plupart sans qu'il y soit nécessaire de l'indiquer explicitement,[70] à l'édition de liens de votre programme,
– un ensemble de *fichiers en-tête* contenant les déclarations nécessaires pour que les appels de ces fonctions puissent être correctement compilés.

Ces fichiers en-tête sont organisés par thèmes ; nous reprenons cette organisation pour expliquer les principaux éléments de la bibliothèque.

Des parties importantes de la bibliothèque standard ont déjà été expliquées ; notamment :
– la bibliothèque des entrées - sorties (associée au fichier `stdio.h`), à la section 7 ;
– les listes variables d'arguments (introduites dans le fichier `stdarg.h`) à la section 4.3.4 ;
– les branchements hors-fonction (déclarés dans le fichier `setjmp.h`) à la section 8.3.2 ;
– la récupération des interruptions (avec le fichier `signal.h`) à la section 8.3.3.

Nous passerons sous silence certains modules mineurs (comme la gestion de la date et l'heure). Au besoin, reportez-vous à la documentation de votre système.

8.4.1 Aide à la mise au point : `assert.h`

Cette « bibliothèque » ne contient aucune fonction. Elle se compose d'une macro unique :

`void assert(int` *expression*`)`

qui fonctionne de la manière suivante : si l'*expression* indiquée est vraie (c'est-à-dire non nulle) au moment où la macro est évaluée, il ne se passe rien. Si l'*expression* est fausse (c.-à-d. si elle vaut zéro), un message est imprimé sur `stderr`, de la forme

`Assertion failed:` *expression*`, file` *fichier*`, line` *numero*

ensuite l'exécution du programme est avortée. Exemple naïf :

```
#include <assert.h>
...
#define TAILLE 100
int table[TAILLE];
...
for (i = 0; j < TAILLE; i++) {
    ...
    assert(0 <= i && i < TAILLE);
    table[i] = 0;
    ...
}
```

l'exécution de ce programme donnera (puisqu'une faute de frappe a rendu infinie la boucle *for*) :

```
Assertion failed: 0 <= i && i < TAILLE, file ex.c, line 241
```

La macro `assert` est un outil précieux pour la mise au point des programmes. Sans elle, il arrive seuvent qu'une situation anormale produite en un point d'un programme ne se manifeste qu'en un autre point sans rapport avec avec le premier ; il est alors très difficile de remonter depuis la manifestation du défaut jusqu'à sa cause. Beaucoup de temps peut être gagné en postant de telles assertions aux endroits « délicats », où les variables du programme doivent satisfaire des contraintes cruciales.

Il est clair, cependant, que cet outil s'adresse au programmeur et n'est utile que pendant le développement du programme. Les appels de `assert` ne doivent pas figurer dans l'exécutable livré à l'utilisateur final, pour deux raisons : d'une part, ce dernier n'a pas à connaître des détails du programme source (en principe il ne connaît même pas le langage de programmation employé), d'autre part parce que les appels de `assert` ralentissent les programmes.

Lorsque la mise au point d'un programme est terminée, on doit donc neutraliser tous les appels de `assert` qui y figurent. Cela peut s'obtenir en une seule opération, simplement en définissant en tête du programme[71] l'identificateur `NDEBUG` :

`#define NDEBUG` *peu importe la valeur*

[70] Notez que, sous UNIX, lorsque des fichiers de la bibliothèque standard ne sont pas implicites, il y a un moyen simple de les indiquer explicitement. Par exemple, l'option `-lm` dans la commande `gcc` spécifie l'apport de la bibliothèque `/lib/libm.a` (ce fichier contient le code des fonctions mathématiques).

[71] Sous UNIX il n'est même pas nécessaire d'ajouter une ligne au programme : il suffit de composer la commande `gcc` en spécifiant une option « `-D` », comme ceci : `gcc -DNEBUG -o` *monprog monprog.c etc.*

juste avant d'effectuer la compilation finale du programme. Celle-ci se fera alors comme si tous les appels de `assert` avaient été gommés.

N.B. Le fait qu'il soit possible et utile de faire disparaître les appels de `assert` du programme exécutable final montre bien que `assert` *n'est pas* la bonne manière de détecter des erreurs que le programmeur ne peut pas éviter, comme les erreurs dans les données saisies. Ainsi – sauf pour un programme éphémère à usage strictement personnel – le code suivant n'est pas acceptable :

```
...
scanf("%d", &x);
assert(0 <= x && x <= N);           /* NON ! */
...
```

La vérification que `0 <= x && x <= N` porte sur une valeur lue à l'exécution et garde tout son intérêt dans le programme final, lorsque les `assert` sont désarmés. C'est pourquoi elle doit plutôt être programmée avec du « vrai » code :

```
...
scanf("%d", &x);
if( ! (0 <= x && x <= N)) {
    fprintf(stderr, "Erreur. La valeur de x doit être comprise entre 0 et %d\n", N);
    exit(-1);
}
...
```

8.4.2 Fonctions utilitaires : `stdlib.h`

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

Ces fonctions calculent et rendent l'`int` (resp. le `long`, le `double`) dont la chaîne `s` est l'expression écrite. Exemple

```
int i;
char *s;
...
affectation de s
...
i = atoi(s);
maintenant i a la valeur numérique dont la chaîne s est l'expression textuelle
```

NOTE. Pour faire le travail inverse de `atoi` ou une autre des fonctions précédentes on peut employer `sprintf` selon le schéma suivant :

```
int i;
char s[80];
...
affectation de i
...
sprintf(s, "%d", i);
maintenant la chaîne s est l'expression textuelle de la valeur de i
```

`int rand(void)`

Le i^{eme} appel de cette fonction rend le i^{eme} terme d'une suite d'entiers pseudo-aléatoires compris entre `0` et la valeur de la constante `RAND_MAX`, qui vaut au moins 32767. Cette suite ne dépend que de la valeur de la `semence` donnée lors de l'appel de `srand`, voir ci-dessous. Si `srand` n'a pas été appelée, la suite obtenue est celle qui correspond à `srand(1)`.

Voyez `srand` ci après.

`void srand(unsigned int semence)`

Initialise une nouvelle suite pseudo-aléatoire, voir `rand` ci-dessus. Deux valeurs différentes de la `semence` – éloignées ou proches, peu importe – donnent lieu à des suites tout à fait différentes, du moins à partir du deuxième terme.

APPLICATION. Pour que la suite aléatoire fournie par les appels de `rand` que fait un programme soit différente à chaque exécution de ce dernier il fait donc commencer par faire *un* appel de `srand` en veillant

à lui passer un argument différent chaque fois. Une manière d'obtenir cela consiste à utiliser l'heure courante (elle change tout le temps !), par exemple à travers la fonction `time`[72].

Par exemple, le programme suivant initialise et affiche une suite pseudo-aléatoire de dix nombres flottants x_i vérifiant $0 \leq x_i \leq 1$:

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

main() {
    int i;
    float x;

    srand(time(NULL));              /* version C de randomize */
    rand();     /* le premier tirage est très lié à la semence */

    for(i = 0; i < 10; i++) {
        x = rand() / (float) RAND_MAX;
        printf("%f\n", x);
    }
}
```

`void *malloc(size_t taille)`

Alloue un espace pouvant mémoriser un objet ayant la taille indiquée, ou `NULL` en cas d'échec. Cet espace n'est pas initialisé.

`void *calloc(size_t nombre, size_t taille)`

Alloue un espace suffisant pour loger un tableau de `nombre` objets, chacun ayant la `taille` indiquée, ou `NULL` en cas d'échec. L'espace alloué est initialisé par des zéros.

`void free(void *adresse)`

Indique au système que l'espace mémoire ayant l'adresse indiquée n'est plus utile au programme. Cette adresse doit nécessairement provenir d'un appel de `malloc`, `calloc` ou `realloc`.

`void *realloc(void *adr, size_t taille)`

Ré-allocation d'espace. La valeur de `adr` doit provenir d'un appel de `malloc`, `calloc` ou `realloc`. Cette fonction essaie d'agrandir ou de rétrécir (usage rare) l'espace pointé par `adr` afin qu'il ait la la `taille` demandée. Si cela peut se faire sur place alors la fonction renvoie la même adresse `adr`. Sinon, elle obtient un nouvel espace ayant la taille voulue, y copie le contenu de l'espace pointé par `adr`, libère cet espace, enfin renvoie l'adresse de l'espace nouvellement alloué.

`void exit(int code)`

Produit l'arrêt normal (fermeture des fichiers ouverts, etc.) du programme. La valeur du `code` indiqué est transmise au système d'exploitation, cette valeur est utilisée lorsque le programme a été appelé dans le cadre d'un procédure de commandes (ou *script*). La signification de ce code dépend du système ; on peut utiliser les constantes :

- `EXIT_SUCCESS` : la valeur conventionnelle qui indique, pour le système sous-jacent, que le programme a réussi à accomplir sa mission ;
- `EXIT_FAILURE` : le programme n'a pas réussi.

`void abort(void)`

Provoque un arrêt anormal du programme (c'est un événement détecté par la fonction `signal`).

`int system(const char *commande)`

Suspend momentanément l'exécution du programme en cours, demande à l'interprète des commandes du système d'exploitation d'exécuter la `commande` indiquée et rend le code donné par cette commande (cette

[72] La fonction `time` renvoie le nombre de secondes écoulées depuis le premier janvier 1970. Elle renvoie donc un nombre entier qui change toutes les secondes.

Si on doit écrire un programme devant initialiser plusieurs suites aléatoires distinctes en moins d'une seconde – ce qui est quand même rarissime – on aura un problème, qu'on peut régler en faisant intervenir d'autres fonctions de mesure du temps. Par exemple `clock` compte le temps (mais c'est un temps relatif au lancement du programme) en fractions de seconde, par exemple des millisecondes.

valeur dépend du système). La chaîne `commande` doit être l'expression complète, avec options et arguments, d'une commande légale pour le système utilisé.

Toutes les versions de C n'offrent pas ce service, mais on peut savoir ce qu'il en est : l'appel `system(NULL)` rend une valeur non nulle si l'interprète de commandes peut effectivement être appelé depuis un programme, zéro sinon.

Exemple (bien connu des utilisateurs de *Dev-C++* ; `pause` est une commande MS-DOS/Windows) :

```
system("pause");
```

`char *getenv(char *nom)`

Rend la chaîne qui est la valeur de la *variable d'environnement* ayant le `nom` indiqué, ou `NULL` si une telle variable n'est pas définie. La notion de variable d'environnement est définie au niveau du système sous-jacent.

`void qsort(void *tab, size_t nbr, size_t taille, int (*comp)(const void *, const void *))`

Quick sort, ou tri rapide. Cette fonction trie sur place (c'est-à-dire sans utiliser un tableaux auxiliaire), par ordre croissant, un tableau dont `tab` donne l'adresse de base, formé de `nbr` objets chacun ayant la `taille` indiquée.

Pour comparer les éléments du tableau, `qsort` utilise la fonction qui est la valeur de l'argument `comp`. Cette fonction doit prendre les adresses de deux objets comme ceux dont le tableau est fait, et rendre une valeur négative, nulle ou positive selon que le premier objet est respectivement inférieur, égal ou supérieur au second.

Voir à la section 6.3.2, exemple 2, un modèle d'utilisation de cette fonction.

```
int bsearch(const void *ptr, const void *tab, size_t nbr, size_t taille,
    int (*comp)(const void *, const void *))
```

Binary search, recherche dichotomique. Cette fonction recherche dans un tableau *qui doit être trié*, ayant l'adresse de base donnée par `tab`, formé de `nbr` objets chacun ayant la `taille` indiquée, un objet égal à celui ayant `ptr` pour adresse. Pour cela, elle utilise la fonction de comparaison donnée par `comp`, qui est définie comme pour `qsort`, voir ci-dessus.

`int abs(int x), long labs(long x)`

Valeur absolue, avec un argument `int` (resp. `long`)[73].

8.4.3 Traitement de chaînes : `string.h`

`char *strcpy(char *destin, const char *source)`

Copie la chaîne `source` à l'adresse `destin`. Rend `destin`.

`char *strcat(char *destin, const char *source)`

Copie la chaîne `source` à la suite de la chaîne `destin`. Rend `destin`.

`int strcmp(const char *a, const char *b)`

Compare les chaînes `a` et `b` pour l'ordre lexicographique (c'est-à-dire l'ordre par lequel on range les mots dans un dictionnaire) et rend une valeur négative, nulle ou positive selon que `a` est, respectivement, inférieur, égal ou supérieur à `b`.

`size_t strlen(const char *s)`

Rend le nombre de caractères de la chaîne `s`. Il s'agit du nombre de caractères *utiles* : le caractère `'\0'` qui se trouve à la fin de toutes les chaînes n'est pas compté. Ainsi, `strlen("ABC")` vaut 3.

`void *memcpy(char *destin, const char *source, size_t nombre)`

Copie la zone mémoire d'adresse `source` de de taille `nombre` dans la zone de même taille et d'adresse `destin` . Ces deux zones *ne doivent pas se rencontrer*.

`void *memmove(char *destin, const char *source, size_t nombre)`

Copie la zone mémoire d'adresse `source` de de taille `nombre` dans la zone de même taille et d'adresse `destin` . Fonctionne correctement même si ces deux zones se rencontrent ou se chevauchent.

[73] La fonction correspondante pour les réels s'appelle `fabs` (cf. section 8.4.5).

8.4.4 Classification des caractères : `ctype.h`

Les éléments de cette bibliothèque peuvent être implantés soit par des fonctions, soit par des macros. Les prédicats rendent, lorsqu'ils sont vrais, une valeur non nulle qui n'est pas forcément égale à 1 :

`int islower(int c)` ⇔ `c` est une lettre minuscule.

`int isupper(int c)` ⇔ `c` est une lettre majuscule.

`int isalpha(int c)` ⇔ `c` est une lettre.

`int isdigit(int c)` ⇔ `c` est un chiffre décimal.

`int isalnum(int c)` ⇔ `c` est une lettre ou un chiffre.

`int isspace(int c)` ⇔ `c` est un caractère d'espacement : `' '`, `'\t'`, `'\n'`, `'\r'` ou `'\f'`.

`int iscntrl(int c)` ⇔ `c` est un caractère de contrôle (c'est-à-dire un caractère dont le code ASCII est compris entre 0 et 31).

`int isprint(int c)` ⇔ `c` est un caractère imprimable, c'est-à-dire qu'il n'est pas un caractère de contrôle.

`int isgraph(int c)` ⇔ `c` est un caractère imprimable autre qu'un caractère d'espacement.

`int ispunct(int c)` ⇔ `c` est une ponctuation, c'est-à-dire un caractère imprimable qui n'est ni un caractère d'espacement, ni une lettre ni un chiffre.

`int tolower(int c)`, `int toupper(int c)` si c est une lettre majuscule (resp. minuscule) rend la lettre minuscule (resp. majuscule) correspondante, sinon rend le caractère c lui-même.

8.4.5 Fonctions mathématiques : `math.h`

`double sin(double x)` rend la valeur de $\sin x$.

`double cos(double x)` rend la valeur de $\cos x$.

`double tan(double x)` rend la valeur de $\tan x$.

`double asin(double x)` rend la valeur de $\arcsin x$, dans $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. On doit avoir $x \in [-1, 1]$.

`double acos(double x)` rend la valeur de $\arccos x$, dans $[0, \pi]$. On doit avoir $x \in [-1, 1]$.

`double atan(double x)` rend la valeur de $\arctan x$, dans $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

`double atan2(double y, double x)` rend la limite de la valeur de $arctan\frac{v}{u}$ dans $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ lorsque $u \to x^+$ et $v \to y$. Pour $x \neq 0$ c'est la même chose que $\arctan \frac{y}{x}$

`double sinh(double x)` rend la valeur du sinus hyperbolique de x, soit $\text{sh}\, x = \frac{e^x - e^{-x}}{2}$

`double cosh(double x)` rend la valeur du cosinus hyperbolique de x, $\text{ch}\, x = \frac{e^x + e^{-x}}{2}$

`double tanh(double x)` rend la valeur de la tangente hyperbolique de x, $\text{th}\, x = \frac{\text{ch}x}{\text{sh}x}$

`double exp(double x)` rend la valeur de l'exponentielle des, e^x.

`double log(double x)` rend la valeur du logarithme népérien de x, $\log x$. On doit avoir $x > 0$.

`double log10(double x)` rend la valeur du logarithme décimal de x, $\log_{10} x$. On doit avoir $x > 0$.

`double pow(double x, double y)` rend la valeur de x^y. Il se produit une erreur si $x = 0$ et $y = 0$ ou si $x < 0$ et y n'est pas entier.

`double sqrt(double x)` rend la valeur de $\sqrt{x}$. On doit avoir $x \geq 0$.

`double ceil(double x)` rend la valeur du plus petit entier supérieur ou égal à x, transformé en `double`.

`double floor(double x)` rend la valeur du plus grand entier inférieur ou égal à x, transformé en double.

`double fabs(double x)` la valeur absolue de x.

8.4.6 Limites propres à l'implémentation : `limits.h`, `float.h`

Ces fichiers en-tête définissent les tailles et les domaines de variation pour les types numériques. On y trouve :

`CHAR_BIT` nombre de bits par caractère.

`CHAR_MAX`, `CHAR_MIN` valeurs extrêmes d'un `char`.

`SCHAR_MAX`, `SCHAR_MIN` valeurs extrêmes d'un `signed char`.

`UCHAR_MAX` valeur maximum d'un `unsigned char`.

`SHRT_MAX, SHRT_MIN` valeurs extrêmes d'un `short`.

`USHRT_MAX` valeur maximum d'un `unsigned short`.

`INT_MAX, INT_MIN` valeurs extrêmes d'un `int`.

`UINT_MAX` valeur maximum d'un `unsigned int`.

`LONG_MAX, LONG_MIN` valeurs extrêmes d'un `long`.

`ULONG_MAX` valeur maximum d'un unsigned long.

Le fichier `float.h` définit :

`FLT_DIG` précision (nombre de chiffres décimaux de la mantisse).

`FLT_EPSILON` plus petit nombre e tel que $1.0 + e \neq 1.0$.

`FLT_MANT_DIG` nombre de chiffres de la mantisse.

`FLT_MAX` plus grand nombre représentable.

`FLT_MAX_10_EXP` plus grand n tel que 10^n soit représentable.

`FLT_MAX_EXP` plus grand exposant n tel que $\texttt{FLT\_RADIX}^n - 1$ soit représentable.

`FLT_MIN` plus petit nombre positif représentable (sous forme normalisée).

`FLT_MIN_10_EXP` plus petit exposant n tel que 10^n soit représentable (sous forme normalisée).

`FLT_MIN_EXP` plus petit exposant n tel que $\texttt{FLT\_RADIX}^n$ soit représentable (sous forme normalisée).

`FLT_RADIX` base de la représentation exponentielle.

`FLT_ROUNDS` type de l'arrondi pour l'addition.

Ces fichiers définissent également les constantes analogues pour les `double` (noms en « `DBL_...` » à la place de « `FLT_...` ») et les `long double` (noms en « `LDBL_...` »).

Index

() *(opérateur)*, 19
* *(opérateur)*, 23
*= *(opérateur)*, 32
++ *(opérateur)*, 22
+= *(opérateur)*, 32
, *(opérateur)*, 33
- *(opérateur)*, 23
-= *(opérateur)*, 32
->*(opérateur)*, 21
-- *(opérateur)*, 22
. *(opérateur)*, 21
/= *(opérateur)*, 32
= *(opérateur)*, 31
== *(opérateur)*, 28
? : *(opérateur)*, 31
[ ] *(opérateur)*, 20
%= *(opérateur)*, 32
& *(opérateur)*, 24, 29
&= *(opérateur)*, 32
&& *(opérateur)*, 30
^*(opérateur)*, 29
^= *(opérateur)*, 32
~*(opérateur)*, 22
|*(opérateur)*, 29
|= *(opérateur)*, 32
||*(opérateur)*, 30
>*(opérateur)*, 28
>= *(opérateur)*, 28
>>*(opérateur)*, 28, 32
<*(opérateur)*, 28
<= *(opérateur)*, 28
<<*(opérateur)*, 28
<<= *(opérateur)*, 32

abort, 117
abs, 118
accès à un champ (opérateur), 21
accès relatif, 97
acos, 119
adresse (opérateur), 24
adresse d'un objet, 24, 64
adresse d'une fonction, 77
affectation (opérateur), 31
appel de fonction, 19, 45, 47
argc, 110
argument formel, 12
arguments des fonctions (passage), 48
arguments en nombre variable, 50
arguments par adresse, 49
argv, 110
arithmétique des adresses, 66
arithmétiques (opérateurs), 27
asin, 119
assert, 115
assert.h, 115
atan, 119
atan2, 119
atof, 116
atoi, 116
atol, 116
automatique (variable), 13

bits (champs de bits), 56
bloc (instruction), 36, 37
break, 36, 42
bsearch, 118

calloc, 117
caractère, 7
case, 36, 41
cast (opérateur), 25, 62
ceil, 119
chaîne de caractères, 7, 53
chaîne de caractères (initialisation), 54
champs de bits, 56
char, 10
CHAR_BIT, 119
CHAR_MAX, 119
CHAR_MIN, 119
close, 101
commentaire, 6
comparaison (opérateur), 28
complément à 1 (opérateur), 22
conditionnelle (compilation), 104
conjonction (opérateur), 30
conjonction bit-à-bit (opérateur), 29
connecteurs logiques (opérateurs), 30
const, 15, 60
continue, 36, 42
conversion de type (opérateur), 25, 62
conversions arithmétiques usuelles, 34
cos, 119
cosh, 119
creat, 101
ctype.h, 119

décalage de bits (opérateur), 28
déclarateur complexe, 58
default, 36, 41
define, 103
disjonction (opérateur), 30
disjonction bit-à-bit (opérateur), 29
do, 36, 39
durée de vie (d'une variable), 13

effet de bord, 18
else, 38, 105
en-tête (fichier), 107
endif, 105
enumération, 57
etiquette, 38
exit, 117
EXIT_FAILURE, 117
EXIT_SUCCESS, 117
exp, 119
expression conditionnelle (opérateur), 31
extern, 16

externe (identificateur), 16

F, f (suffixe d'une constante), 7
`fabs`, 119
`fclose`, 86
`feof`, 86
`ferror`, 86
`fflush`, 85
`fgetc`, 87
`fgetpos`, 97
`fgets`, 88
fichier binaire, 84
fichier de texte, 84
`FILE`, 85
`float.h`, 119, 120
`floor`, 119
flots, 84
flottante (constante), 7
`FLT_DIG`, 120
`FLT_EPSILON`, 120
`FLT_MANT_DIG`, 120
`FLT_MAX`, 120
`FLT_MAX_10_EXP`, 120
`FLT_MAX_EXP`, 120
`FLT_MIN`, 120
`FLT_MIN_10_EXP`, 120
`FLT_MIN_EXP`, 120
`FLT_RADIX`, 120
`FLT_ROUNDS`, 120
fonction formelle, 78
`fopen`, 85
`for`, 36, 40
`fprintf`, 95
`fputc`, 89
`fputs`, 89
`fread`, 96
`free`, 117
`fscanf`, 96
`fseek`, 96
`fsetpos`, 97
`fwrite`, 96

`getc`, 87
`getchar`, 88
`getenv`, 118
`gets`, 88
globale (variable), 12
`goto`, 36, 38

hexadécimale (écriture d'un nombre), 7

identificateur, 6
`if`, 36, 38, 105
`ifdef`, 105
`ifndef`, 105
`include`, 102
indexation (opérateur), 20, 66
indirection (opérateur), 23
initialisation d'un tableau, 52
initialisation d'une chaîne de caractères, 54
initialisation d'une structure, 55
initialisation d'une variable, 13

`INT_MAX`, 120
`INT_MIN`, 120
`isalnum`, 119
`isalpha`, 119
`iscntrl`, 119
`isdigit`, 119
`isgraph`, 119
`islower`, 119
`isprint`, 119
`ispunct`, 119
`isspace`, 119
`isupper`, 119

`jmp_buf`, 112

L, l (suffixe d'une constante), 7
`labs`, 118
`limits.h`, 119
locale (variable), 12
`log`, 119
`log10`, 119
logiques (opérateurs), 30
`LONG_MAX`, 120
`LONG_MIN`, 120
`longjmp`, 112
`lseek`, 101
lvalue, 18

macros, 103
`main`, 110
`malloc`, 117
`math.h`, 119
`memcpy`, 118
`memmove`, 118
moins unaire (opérateur), 23
mot-clé, 6

négation (opérateur), 22
`NDEBUG`, 115
nombre entier, 7, 9
nombre flottant, 7, 11
`NULL`, 66

`O_RDONLY`, 101
`O_RDWR`, 101
`O_WRONLY`, 101
octale (écriture d'un nombre), 7
opérateur, 6
`open`, 101
ordre d'évaluation des expressions, 34
ou exclusif bit-à-bit (opérateur), 29

passage des arguments des fonctions, 48
pointeur, 64
post-decrémentation (opérateur), 22
post-incrémentation (opérateur), 22
`pow`, 119
pré-decrémentation (opérateur), 22
pré-incrémentation (opérateur), 22
préprocesseur, 102
`printf`, 89, 95
priorité des opérateurs, 19

privé (dentificateur), 15
prototype d'une fonction, 44
public (dentificateur), 15
`putc`, 89
`putchar`, 89
`puts`, 89

`qsort`, 78, 118

`rand`, 116
`read`, 101
`realloc`, 117
`register`, 14
`return`, 36, 43
`rewind`, 97
rvalue, 18

séquence d'échappement, 7
`scanf`, 92, 96
`SCHAR_MAX`, 119
`SCHAR_MIN`, 119
`SEEK_CUR`, 96
`SEEK_END`, 96
`SEEK_SET`, 96
`setjmp`, 112
`setjmp.h`, 112
`setvbuf`, 86
`SHRT_MAX`, 120
`SHRT_MIN`, 120
`signal`, 113
`signal.h`, 113
`sin`, 119
`sinh`, 119
`sizeof`, 24
`sprintf`, 95
`sqrt`, 119
`srand`, 116
`sscanf`, 96
`static`, 14
statique (variable), 14
`stderr`, 87
`stdin`, 87
`stdio.h`, 85
`stdlib.h`, 116
`stdout`, 87
`strcat`, 118
`strcmp`, 118
`strcpy`, 118
`string.h`, 118
`strlen`, 118
`struct`, 54
structure (déclaration), 54
structure (initialisation), 55
structure (signification d'une variable structure), 55
structures récursives, 81, 82
`switch`, 36, 41
`system`, 117

tableau (déclaration), 52
tableau (initialisation), 52
tableau (signification d'une variable tableau), 52
tableau [dynamique] multidimensionnel, 69
tableau de chaînes de caractères, 72
tableau de fonctions, 79
tableau dynamique, 68
taille d'un objet (opérateur), 24
tampon (d'entrée ou de sortie), 84
`tan`, 119
`tanh`, 119
`tmpfile`, 86
`tolower`, 119
typé énuméré, 57
type énuméré, 11
`typedef`, 61
types, 9
types désincarnés, 62

U, u (suffixe d'une constante), 7
`UCHAR_MAX`, 119
`UINT_MAX`, 120
`ULONG_MAX`, 120
`ungetc`, 89
`union`, 56
`USHRT_MAX`, 120

variable, 11
virgule (opérateur), 33
`void`, 45
`void *`, 65
`volatile`, 15, 60

`while`, 36, 39
`write`, 101