

Université d'ORLÉANS

Le langage C

Une rapide introduction

AbdelAli ED-DBALI

`AbdelAli.Ed-Dbali@lifo.univ-orleans.fr`

Table des matières

1	Introduction	2
2	Présentation générale	2
2.1	Forme générale d'un programme \mathcal{C}	2
3	Éléments de base du langage	4
3.1	Identificateurs	4
3.2	Constantes	4
4	Types standards	4
4.1	Les caractères	5
4.2	Les entiers	5
4.3	Les réels	5
5	Opérateurs	6
5.1	Opérateurs arithmétiques	6
5.2	Opérateurs booléens	6
5.3	Opérateurs sur les bits	6
5.4	Conversions de types	7
5.5	Evaluation d'une expression	7
5.6	Opérations d'affectation	7
6	Structures de contrôle	8
6.1	Conditionnelle	8
6.2	Boucles	9
7	Introduction aux tableaux à une dimension	10
8	Fonctions	11
8.1	Forme générale	11
8.2	Passage des paramètres	12
8.3	Exemple	13
8.4	Variables externes, variables locales	14
8.5	Variables statiques	14
8.6	Registres	15
8.7	Réursivité	15
8.8	Initialisation	16
9	Pointeurs	16
10	Rapports entre tableaux et pointeurs	17
10.1	Tableaux à une dimension	17
10.2	Tableau à deux dimensions	18

11 Structures	18
11.1 Structures imbriquées	19
11.2 Structures de bits	20
11.3 Enumérations	20
11.4 Unions	20
12 Complément sur les pointeurs	22
12.1 Les listes	22
12.2 Tableau de chaînes de caractères	23
12.3 Tableau de fonctions	23
13 Entrées/Sorties	24
14 Compilation séparée et création de bibliothèques	25
15 Fonctions à nombre variable de paramètres	27
15.1 Dans l'ancienne norme	28
15.2 Dans la nouvelle norme	28

1 Introduction

Le langage *C* doit son apparition au système UNIX. En effet, c'est pour écrire une version *portable* de ce système que ses auteurs, (D. Ritchie, K. Thompson et B. Kernighan)¹, ont été amenés à définir un langage d'assemblage de haut niveau. Le succès du système UNIX a incontestablement contribué à celui du langage *C* qui n'a pas vu le jour dans le but de concurrencer les langages existants de l'époque (FORTRAN et PASCAL par exemple).

Après s'être imposé comme le langage de programmation sous UNIX, *C* s'est forgé une réputation, au fil du temps, de langage de programmation tout court. En voici quelques caractéristiques (bonnes ou mauvaises) :

- Il est très laxiste quant au problème du contrôle des types.
- Contient une grande quantité d'opérateurs (on peut descendre au niveau du bit).
- Il ne gère ni les entrées-sorties (pas de fichiers, pas d'instruction d'entrées-sorties) ni l'allocation de la mémoire dynamique. Mais tout ceci existe dans les bibliothèques, standardisées, de l'environnement de programmation (d'où la nécessité d'édition des liens).
- Ses instructions sont en nombre moins élevé que dans un langage classique, aussi le compilateur tourne plus vite qu'un compilateur PASCAL par exemple.
- Il est réputé assurer une bonne portabilité².
- Il offre de grandes possibilités de manipulation des pointeurs.
- La compilation séparée des modules de programmes se fait d'une manière naturelle.

2 Présentation générale

2.1 Forme générale d'un programme *C*

A priori, le programme peut être composé de plusieurs segments.

<i>Fichier 1</i>	<i>Fichier 2</i>
Déclarations	Déclarations
fonction ₁	fonction ₃
fonction ₂	fonction ₄
main	

Voici notre premier programme (comme d'habitude) :

En PASCAL

En *C*

```
program MonPremier;                               main() {
```

1. Le langage *C* a été écrit en 1972 par une seule personne : D. Ritchie des Laboratoires Bell de AT&T. K. Thompson avait auparavant développé UNIX sur PDP-11 en langage d'assemblage et en B (un langage non typé, ancêtre de *C*).

2. Ceci dépend, bien évidemment, de la manière avec laquelle on programme.

```

begin
    writeln('Bonjour !')
end.
    printf("Bonjour !\n");
}

```

main est une fonction, sans arguments pour le moment, qui joue le rôle de programme principal en PASCAL. Les accolades { et } jouent le rôle de **begin** et **end** de PASCAL. Pour matérialiser le retour à la ligne (comme le fait le `writeln`) on inclut, dans la chaîne à afficher, le caractère spécial retour à la ligne `'\n'`.

Exercice 1 : *Ecrire en PASCAL puis en C un programme qui lit un caractère et affiche son code.*

En PASCAL	En C
<code>program Code;</code>	<code>#include <stdio.h></code>
<code>var x : char;</code>	<code>main() {</code>
<code>begin</code>	<code>char x;</code>
<code> readln(x);</code>	<code>x = getchar() ;</code>
<code> writeln('Caractere :',</code>	<code>printf("Caractere : %c Code : %d\n", x, x);</code>
<code> x, 'Code :', ord(x));</code>	
<code>end.</code>	<code>}</code>

Remarque: Un commentaire en C s'écrit entre les 2 couples de caractères: `/* et */`.

L'appel de la fonction `getchar()` renvoie le code décimal du caractère lu à partir de l'entrée standard (c'est-à-dire le clavier). Le premier argument de `printf` contient le format d'affichage des arguments suivants: `%c` pour caractère et `%d` pour décimal. On peut déclarer la variable `x` avec `'int x'` pour éviter le problème des codes négatifs (car le type **char** n'est autre que le sous ensemble $0..2^8 - 1$ du type **int**).

La directive `#include <stdio.h>`

demande à inclure avant compilation le fichier **stdio.h** qui se trouve à l'emplacement "standard" des fichiers de la bibliothèque C (en UNIX cette bibliothèque se trouve dans le répertoire: `/usr/include`). Ce fichier comprend des définitions indispensables pour la plupart des opérations d'entrée/sortie.

Exercice 2 : *Lire deux entiers et écrire leur somme.*

```

#include <stdio.h>
main( ) {
    int x,y;
    scanf("%d %d", &x, &y);
    printf("somme de deux entiers : %d + %d = %d\n", x, y, x+y);
}

```

Dans l'appel de la fonction `scanf` les arguments `x` et `y` sont précédés du symbole `'&'`. `&x` désigne l'adresse en mémoire de la variable `x`. Le pourquoi de cette gymnastique est expliqué dans la section 8.2 (**Passage des paramètres**).

Pour la somme de deux réels, nous aurons le même programme avec `float` au lieu de `int` et `%f` au lieu de `%d`.

Note: Pour imprimer le caractère %, il faut le doubler: `printf ("%d %% %d\n", i , i)` affiche `2 % 2` si `i` vaut 2. De même pour avoir dans une chaîne le caractère %.

3 Eléments de base du langage

3.1 Identificateurs

Un identificateur est une suite de lettres, de chiffres et du caractère '_' (blanc souligné ou underscore) qui commence par une lettre ou '_' (C fait la différence entre majuscules et minuscules: `Pi` et `pi` sont deux identificateurs différents). Seuls les 31 premiers caractères sont significatifs. Par tradition, les minuscules désignent les variables, les majuscules désignent les constantes.

3.2 Constantes

L'ensemble des constantes regroupe:

1. les entiers: suite de chiffres;
2. les réels: `123.5e3` (e pour l'exposant 10);
3. les entiers représentés en octal et hexadécimal: le premier chiffre est un 0 (zéro).

`015` octal = 13 en décimal ($1 \star 8 + 5$);

`0x15` hexadécimal = 21 en décimal ($1 \star 16 + 5$).

4. les caractères: `'a'`, `'\'`, `'#'`, `''''`

Voici d'autres caractères spéciaux:

`'\a'` caractère d'alerte (sonnerie, *bell*)

`'\b'` retour arrière (*backspace*)

`'\f'` saut de page (*from feed*)

`'\n'` fin de ligne (*line feed*)

`'\r'` retour chariot (*carriage return*)

`'\t'` tabulation horizontale

`'\v'` tabulation verticale

`'\\'` \

`'\015'` caractère dont le code est 15 en octal

`'\x15'` caractère dont le code est 15 en hexadécimal

5. les chaînes de caractères: `"abcd"`, `"a\015bcd\n"`

La chaîne `"abcd"` est représentée en mémoire par un tableau, de 5 cases, terminé par `'\0'` (caractère nul ASCII):

<code>'a'</code>	<code>'b'</code>	<code>'c'</code>	<code>'d'</code>	<code>'\0'</code>
------------------	------------------	------------------	------------------	-------------------

4 Types standards

Nous énumérons les types de la nouvelle norme dont certains ne figurent pas dans l'ancienne. Ces derniers sont signalés par une \star .

4.1 Les caractères

`char` caractères (1 octet) selon les machines : -128..127 ou 0..255 ;
`signed char*` caractères signés (1 octet) : -128..127 ;
`unsigned char*` caractères non signés (1 octet) : 0..255.

Ces types basés sur `char` sont des sous-ensembles du type des entiers : `int`.

4.2 Les entiers

`int` entiers (2 ou 4 octets) selon les machines :
 $-2^{15}..2^{15} - 1$ (-32 768..32 767) ou
 $-2^{31}..2^{31} - 1$ (-2 147 483 648..2 147 483 647) ;
`short int` ou `short` entiers courts (2 octets) ;
`long int` ou `long` entiers longs (4 octets) ;
`signed int*` entiers signés (2 ou 4 octets) ;
`signed short int*` ou
`signed short*` entiers courts signés (2 octets) ;
`signed long int*` ou
`signed long*` entiers longs signés (4 octets) ;
`unsigned int` entiers non signés (2 ou 4 octets) selon les machines :
 $0..2^{16} - 1$ ou $0..2^{32} - 1$;
`unsigned short int` ou
`unsigned short` entiers courts non signés (2 octets) ;
`unsigned long int` ou
`unsigned long` entiers longs non signés (4 octets).

4.3 Les réels

`float` nombres flottants en simple précision (4 octets) ;
`double` ou `long float` nombres flottants en double précision (8 octets) ;
`long double*` nombres flottants en précision étendue (12 octets).

Exercice 3 : *Calculer la taille des types de données.*

```
#include <stdio.h >
main() {
    printf ("caractere %d\n", sizeof(char));
    printf ("entier %d\n", sizeof(int));
    printf ("float %d\n", sizeof(float));
    printf ("short %d\n", sizeof(short));
    printf ("long %d\n", sizeof(long));
    printf ("double %d\n", sizeof(double));
}
```

Résultat de l'exécution :

```
caractere  1
entier     2
float      4
short      2
long       4
double     8
```

5 Opérateurs

5.1 Opérateurs arithmétiques

- Opérations sur les entiers: + - * /(division euclidienne) % (modulo).
Comparateurs: < <= > >= == (== égal) != (! = non égal)
- Opérations sur les réels: + - * / et comparateurs.
- Opérations sur les caractères: Les mêmes que pour les entiers. Les caractères peuvent être considérés comme des entiers ($\text{char} \subset \text{int}$):
Si on suppose que le code ASCII du caractère 'a' est 97, l'expression booléenne ('a' == 97) donnera le résultat vrai. En plus, l'évaluation de ('a' + 1) donne ((code du caractère 'a') + 1) = 'b'.

5.2 Opérateurs booléens

Il n'y a pas de type booléen à proprement parlé. Les valeurs logiques *vrai* et *faux* sont représentées par des entiers: 0 pour *faux* et n'importe quel autre nombre ($\neq 0$) pour *vrai*.

Les connecteurs logiques *et*, *ou* et *non* sont notés respectivement &&, || et !.

5.3 Opérateurs sur les bits

```
&      et
|      ou
~      négation
^      ou exclusif
<<    décalage à gauche (compléter par des zéros)
      (Concretement,  $x \ll n = x * 2^n$ )
>>    décalage à droite (compléter par le signe si quantité signée, par des zéros sinon)
      (Concretement,  $x \gg n = x / 2^n$ )
```

▷ Exemples :

- `x & (1 « i)` donne la valeur du ième bit de x (réalise un masque);
- « -- forcer à zéro un bit: faire un et avec un 0 en ième position et des 1 ailleurs
`x = x & ~ (1 « i);`

```
« -- mettre un 1 en ième position : x = x | (1 « i)
```

5.4 Conversions de types

Soient les déclarations suivantes : `int i` et `float f`.

L'affectation `i = f` tronque la partie décimale et garde la partie entière. C'est donc équivalent à l'affectation à `i` de la partie entière de `f`. L'affectation `f = i` ne fait que transformer la valeur entière de `i` sous format réel.

5.5 Evaluation d'une expression

Pour évaluer une expression, on commence par convertir les caractères en entiers et les réels simple précision *float* en réels double précision *double*. Puis s'il y a un mélange de types sur un opérateur binaire, le type faible est converti dans le type fort.

▷ Exemple :

```
'a'   +   2.5
  ↓       ↓
 int     double
  ↓
double + double → double
```

Hiérarchie des types :

```
double → unsigned long → long → unsigned int → int → char
  ↑                                     ↑
 plus fort                             plus faible
```

On peut aussi forcer les conversions : (*type*) *valeur*

▷ Exemple : `(int) 'a'` (coder le caractère 'a' dans un entier).

▷ Autre exemple : `sqrt` (racine carrée) attend un argument de type `double`. Si on veut calculer la racine carrée d'un entier `i` (déclaré par `int i`;) il faut le convertir avant de le passer en argument à `sqrt` : `j = sqrt((double)i)`;

5.6 Opérations d'affectation

`x++` incrémente `x` de 1. Cette *expression* d'affectation donne comme valeur, celle de `x` avant l'incrément ;

`++x` augmente d'abord la valeur de `x` avant de l'utiliser ;

`x--` et `--x` ont les mêmes significations qu'avec `++` mais avec décrément.

L'affectation `x = x + 2` peut être abrégée en `x += 2`. De même l'affectation `x = x + 1` est équivalente à `x += 1` elle même équivalente à `++x`.

L'affectation est une expression qui a une valeur. Cette valeur est celle de la variable affectée : `x = y = 0` ; est équivalent à `x = (y = 0)` ;

Les affectations abrégées supportent tous les opérateurs (arithmétiques, booléens,

opérations sur les bits). De ce fait on peut réécrire, pour toute opération Op , l'affectation " $x = x Op (Exp)$ " en " $x Op= Exp$ ".

▷ Exemple : d'opérations sur les bits :

Forcer le bit i à 0 dans x peut s'écrire : $x \&= \sim (1 \ll i)$;

Forcer le bit i à 1 dans x peut s'écrire : $x |= 1 \ll i$;

6 Structures de contrôle

6.1 Conditionnelle

Conditionnelle if :

```
if (< condition >)
  instruction1;
[else
  instruction2;
```

La partie **else** est optionnelle. Dans les deux parties (if et else) il n'y a qu'une instruction. Chacune d'elles est terminée par un point virgule, y compris celle avant le else (erreur classique en PASCAL).

Exercice 4 : Lire un caractère, si c'est une lettre minuscule la convertir en majuscule.

Expression conditionnelle :

```
(condition) ? valeur1 : valeur2;
```

Cette expression a pour valeur $valeur_1$ si $condition$ est vraie et $valeur_2$ sinon.

▷ Exemple : Maximum de 2 nombres.

```
max = (x < y) ? y : x;
```

équivalent à :

```
if (x < y)
  max = y;
else
  max = x;
```

Les branchements sélectifs (Aiguillage) :

C'est l'équivalent du '**case ... of**' en PASCAL (avec quelques différences bien entendu).

```
switch (expression) {
  case valeur1 : ....; break;
  case valeur2 : ....; break;
  :
  [default : ....;]
}
```

▷ Exemple :

```
switch(c) {
  case 'a' : x = 0; break;
  case 'b' : case 'l' : case 'p' : x = 5; break;
  default  : x = 8;
}
```

L'instruction `break` provoque l'abandon du block où elle se trouve (ici l'instruction `switch`). `default` ainsi que `break` ne sont pas obligatoires. `break` est conseillé sinon l'instruction `switch` teste les autres cas qui suivent.

Exercice 5 : Lire deux nombres et un signe opératoire (+, -, *, / ou %) puis effectuer l'opération correspondante.

6.2 Boucles

Première variante : `while (<condition>) instruction`

▷ Exemple : Affichage des entiers inférieurs ou égaux à n .

```
while (n > 0) printf ("%d", n--);
```

Deuxième variante : `for (exp1; exp2; exp3) instruction`

Avec: exp_1 une expression d'initialisation;
 exp_2 la condition de poursuite; exp_3 l'incrément, faite en bas de boucle.

L'équivalent de cette instruction est :

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

▷ Exemples :

```
for (i = 1 ; i <= 10 ; i++) ...
```

```
for (i = 1 ; j > 15 ; k += 2) ...
```

```
for ( ; ; ) { /* boucle infinie */
  ...
  if (x == 0) break; /* on peut en sortir par break */
}
```

Attention: L'instruction `if (x = Exp) instruction1; else instruction2;` ne fait pas ce qu'on peut s'attendre d'elle, c'est-à-dire: si x est différent de Exp alors exécuter `instruction1` sinon exécuter `instruction2`. En effet, la condition `(x = Exp)` ne teste pas l'égalité entre la valeur de x et Exp mais réalise l'affectation. La valeur

de cette condition est donc celle de `Exp`. Autrement dit, notre conditionnelle sera équivalente à :

```
x = Exp;
if (x != 0) instruction1; else instruction2;
```

Troisième variante : `do instruction while (condition)`

L'instruction est effectuée au moins une fois. C'est l'équivalent de la boucle

```
repeat ... until de PASCAL.
```

Exercice 6 : Lire une suite de caractères et la convertir en un entier.

Exercice 7 : Dans un texte, remplacer les tabulations par des espaces. Il ne s'agit pas de remplacer systématiquement une tabulation par huit espaces.

Si on numérote les colonnes à partir de 0 :

```
«-- si le caractère tabulation est en colonne 2 il faut le remplacer
    par 6 espaces pour se retrouver en colonne 8 ;
«-- si le caractère tabulation est en colonne 0 il faut le remplacer
    par 8 espaces pour se retrouver en colonne 8 ;
«-- si le caractère tabulation est en colonne 14 il faut le remplacer
    par 2 espaces pour se retrouver en colonne 16 ;
    etc.
```

D'où l'algorithme :

- Avancer de $8 - i \% 8$
- Aller à $i + 8 - i \% 8$

7 Introduction aux tableaux à une dimension

Déclaration : `type identificateur[taille]`;

Le tableau de nom `identificateur` a `taille` éléments de type `type`. Le tableau est indicé de 0 à `taille - 1`.

▷ Exemple : `int t [100]` ; déclare un tableau `t` de 100 entiers indicés de 0 à 99.

On ne peut pas affecter globalement un tableau.

`int t1[100], t2[100]; t1 = t2;` est interdit.

Boucle classique pour parcourir un tableau :

On utilise la directive de définition de macros (`#define`) pour définir la longueur du tableau comme constante (ou par `const int` dans la nouvelle norme).

```
#define L 100
int t[L] ;
for (i = 0 ; i < L ; i++) ... boucle de parcours du tableau
```

Exercice 8 : Saisir puis trier un tableau d'entiers (tri par sélection).

Exercice 9 : Lire une série de nombres et à chaque lecture insérer le nouveau nombre lu au bon endroit (tri par insertion).

Définitions par l'ordre **define**.

▷ Exemples :

```
#define AND &&
#define OR ||
#define begin {
#define end }
#define True 1
#define False 0
#define carre(x) (x) * (x)
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Les parenthèses dans `(x) * (x)` sont utiles car sinon `carre (a + b)` donnerait `a + b * a + b` qui s'évalue par `a + (b * a) + b!`

8 Fonctions

8.1 Forme générale

Ancienne norme :

```
[type] nom([par1 ... parn])
      [type1 par1 ... typen parn] {
      [Déclarations locales]
      Instructions
      return [Résultat]
      }
```

Si le type de la fonction est absent c'est `int` qui est pris par défaut.

Pour dire qu'une fonction est sans type (l'équivalent d'une procédure en PASCAL), il faut précéder son nom du mot **void**.

Nouvelle norme :

Elle ne diffère de l'ancienne norme qu'au niveau de l'en-tête où sont intégrés les types des paramètres de la fonction:

```
[type] nom([type1 par1 ... typen parn]) {
      [Déclarations locales]
      Instructions
      return [Résultat]
      }
```

Dans cette norme, **void** est un type générique.

▷ Exemple : Calcule de a^b (a et b des entiers positifs)

```
puissance (a,b) /* cette fonction renvoie un resultat de type int */
int a, b; {
```

```

    int p, i;      /* p et i sont des variables locales */
    p = 1;
    for (i = 1; i <= b; i++) p *= a;
    return p;     /* le resultat est la valeur de p */
}

```

```

main () {
    int i, j;
    scanf ("%d %d", &i, &j);
    printf ("%d\n", puissance(i, j)) ;
}

```

On peut faire : $x = \text{puissance}(i-j, 3) / 5$
a reçoit la valeur de $i-j$
b reçoit la valeur 3

```

    Autre méthode pour calculer la puissance
    puissance (a,b)
    int a, b; {
    int p = 1;
    while (b-- > 0) p *= a;
    return p;
}

```

8.2 Passage des paramètres

En C, les paramètres effectifs ne sont transmis que par valeur. Autrement dit, un paramètre effectif ne peut pas être modifié par une fonction (c'est uniquement sa copie qui lui est transmise). Pour simuler une transmission par variable (comme en PASCAL), on doit transmettre comme valeur l'adresse du paramètre. Il va sans dire que le texte de la fonction doit tenir compte du fait que c'est un pointeur qui est transmis et que la donnée qui l'intéresse est l'objet pointé.

Opérateurs d'adressage : Ils sont en nombre de 2 et jouent des rôles symétriques :

$\&x$: adresse de x .

$*p$: objet adressé par p .

Ceci veut dire que $*\&x = x$ (l'objet pointé par l'adresse de x est x lui même).

Exercice 10 : Ecrire une fonction $\text{echange}(x,y)$ qui échange les valeurs des 2 variables x et y .

On ne peut pas définir une fonction à l'intérieur d'une autre. **main**³ est une fonction qui peut-être définie avant les fonctions qu'elle

3. L'équivalent du programme principal en PASCAL

utilise. Dans la nouvelle norme, si une fonction f utilise une fonction g non encore définie, il faut précéder la définition de f par le prototype de g . Un prototype a la forme suivante :

```
[type] fonction([type1,type2,...,typen]);
```

8.3 Exemple

Problème : Trier un tableau par échanges.

```
#include stdio.h>
#define L 10
int t[L] ;
main () {
    LireTab(t) ;
    TriTab(t) ;
    EcrireTab(t) ;
}

LireTab(u)
int u[] { /* Inutile de donner la longueur, le compilateur doit */
int i; /* seulement savoir que c'est un tableau d'entiers. */
for (i = 0 ; i < L ; i++)
    scanf("%d", &u[i]) ;
}

EcrireTab(u)
int u [] {
int i,
for (i = 0 ; i < L ; i++) printf("%d ", u[i]) ;
putchar('\n') ;
}

echange(x,y) (cf. plus haut)

int RechMin (u, d, f) /* Cherche l'indice du minimum : */
int u[],d, f { /* tableau, indice debut, indice fin */
int Imin, j ;
Imin = d ;
for (j = d+1 ; j <= f ; j++)
    if (u[j] < u[Imin]) Imin = j ;
return Imin ;
}

TriTab (u) int u[] { /* trie le tableau u de longueur L */
int i ;
for (i=0 ; i < L-1 ; i++)
```

```

    echange(&u[RechMin(u, i, L-1)], &u[i]) ;
}

```

8.4 Variables externes, variables locales

«-- Soit le schéma de programme suivant :

```

int x
f(...) ...{
int y ;
...
}

```

La variable x est dite *globale externe*. Globale : car elle peut servir dans la fonction f , comme elle peut servir dans d'autres fonctions. Externe : car elle peut dépasser le cadre du fichier où elle a été déclarée (voir ci-dessous).

La variable y est dite *locale dynamique*. Locale : car elle n'est visible qu'au sein de la fonction f . Dynamique : car elle est créée au moment de l'appel de la fonction f et disparaît à la fin de son exécution.

«-- Dans le schéma du programme suivant :

```

fich 1          fich 2
int x;          extern int x;
f(...) ...     h(...) ...
g(...) ...     k(...) ...
main ()

```

la variable x de fich 2 est commune à toutes les fonctions de fich 2 et c'est le même x que celui défini dans fich 1.

Remarque : Les fonctions a priori sont externes.

8.5 Variables statiques

On peut distinguer deux catégories :

globales : Soit la situation suivante : (programme en deux segments (ou fichiers) F_1 et F_2)

F_1 contient la ligne : `static int x` et F_2 la ligne : `extern int x` `static` cache x des autres segments du programme. Autrement dit, ce x n'est plus utilisable par les autres segments. La tentative `extern int x` ne permet pas d'utiliser cette variable précise.

locales : Etudions le cas de la fonction f suivante :

```

f(...) ... {
static int x ;
...
}

```

La variable `x` est invisible à l'extérieur de la fonction mais elle persiste d'un appel à l'autre. On mémorise quelque chose d'une fois à l'autre, mais on ne peut pas l'utiliser dans une autre fonction. La place de `x` est réservée dès la compilation et l'initialisation a lieu la première fois uniquement. Ce mécanisme permet par exemple de compter le nombre de fois qu'on a utilisé une fonction.

▷ Exemple :

```
f() {
    static int nb = 0 ; /* initialisation operée seulement la première fois */
    ...
    return ++nb ;      /* renvoie le nombre de fois qu'on a appelé f */
}
```

8.6 Registres

Après la déclaration `register int x;` `x` est placé dans un registre⁴ si c'est possible, utilisable uniquement pour des variables locales ou en paramètres.

```
f() {
    register int x;
    ...
}
ou f(x) register int x; {
    ...
}
```

8.7 Récursivité

▷ Exemples :

1. Calcul de la factorielle d'un entier positif :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n - 1)! & \text{si } n > 0 \end{cases}$$

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

ou bien : `{return (n <= 0)? 1: n * factorielle(n-1)}`

ou bien : `{return (n > 0)? n * factorielle(n-1): 1}`

2. **Exercice 11** : *Décoder un nombre en une suite d'entiers de 0 à 9.*

```
decoder(n) int n; {
    int q;
    q = n / 10;
    if (q > 0) decoder(q);
}
```

4. Un registre est une zone mémoire rapide.

```
    putchar(n % 10 + '0');
}
```

8.8 Initialisation

On peut initialiser une variable à la déclaration (exemple : `int i = 0;`). L'initialisation est toujours possible avec une variable simple, qu'elle soit globale, locale dynamique ou locale statique. Avec une variable structurée (structure (voir plus loin) ou tableau) elle n'est possible que si elle est globale ou statique.

▷ Exemples :

```
«-- int t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
«-- int t[2][3] = {{1, 2, 3}, {4, 5, 6}};
    ou plus lisiblement:   int t[2][3] = {{1,2,3},
                                {4,5,6}}
«-- int t[ ] = {1, 2, 3, 4}: Si la longueur du tableau n'est pas spécifiée,
    elle est déterminée par l'initialisation (ici la longueur est
    égale à 4).
«-- int t[10] = {1, 2, 3, 4}: La longueur du tableau est 10 mais seulement
    les 4 premières cases sont initialisées.
«-- char S[10] = "beau": tableau de caractères représenté en mémoire
    comme suit:


|     |     |     |     |      |   |   |   |   |   |
|-----|-----|-----|-----|------|---|---|---|---|---|
| 0   | 1   | 2   | 3   | 4    | 5 | 6 | 7 | 8 | 9 |
| 'b' | 'e' | 'a' | 'u' | '\0' |   |   |   |   |   |


«-- char S[ ] = "beau": définit un tableau de longueur 5 (les 4 lettres
    du mot plus le caractère '\0').
«-- char S[1+1] = "t" ou encore char S[3 + sizeof(int)] = "c" (3 +
    sizeof(int) est une expression constante).
    Attention: L'affectation S = "beau" est interdite parce que S
    est un tableau; de même l'instruction t = {0, 1, 2, 3} est interdite
    (en dehors de l'initialisation).
```

9 Pointeurs

Déclaration : `type * pointeur;`

pointeur est une variable qui pointerà vers une zone mémoire de type *type*. Cette déclaration ne réserve pas de place pour l'objet pointé.

▷ Exemple : `int * p;` déclare que *p* est un pointeur vers un entier mais ne réserve pas de place pour cet entier.

Illustration de la manipulation des pointeurs : `int x;`

`int * p;`

...

`p = &x;` &x désigne l'adresse de *x*. Après cette instruction *p* pointe

sur l'entier x.

...

```
int * q;
```

...

q = p; les 2 pointeurs pointent vers le même entier x.

p++; incrémente le pointeur de la taille de l'objet pointé.

*q = *p; \Leftrightarrow (l'objet pointé par q) = (l'objet pointé par p);

C'est-à-dire que si q pointe vers un entier y et p pointe vers x alors

*q = *p; \Leftrightarrow y = x;

Gestion de la mémoire: (allocation/libération)

Pour gérer la mémoire par le biais des pointeurs il faut inclure le fichier en-tête 'malloc.h' (#include <malloc.h> en début de programme). Le fichier malloc.h contient, entre autres, la fonction dont le prototype est: char * malloc(unsigned int);

Utilisation: Soit p un pointeur vers un entier. L'instruction malloc(sizeof(int)); alloue une zone de la taille d'un entier. Le résultat est un pointeur vers un caractère 'char *'. Pour que p pointe sur cette zone, il faut donner l'instruction suivante:

```
p = (int *) malloc(sizeof(int)); (c'est l'équivalent de l'instruction new(p) de PASCAL).
```

L'instruction free(p); libère la zone adressée par p (l'équivalent du dispose(p) de PASCAL).

10 Rapports entre tableaux et pointeurs

10.1 Tableaux à une dimension

Un tableau n'est, en réalité, qu'un pointeur *constant* qui pointe vers le premier élément. La déclaration int a[10]; alloue une zone de taille égale à dix fois la taille d'un entier et la fait pointer à son début par a.

Soit la déclaration suivante: int * p;.

L'instruction p = a; fait pointer p vers la première case du tableau (qui donne l'adresse du début du tableau).

```
p = a;  $\Leftrightarrow$  p = &a[0];
```

p++; fait avancer p de la taille d'un entier. C'est-à-dire que p pointe maintenant sur la seconde case du tableau.

Plus généralement, p + i pointe sur la ième case du tableau:

```
p + i  $\Leftrightarrow$  &a[i] = a + i et
```

```
*(a + i)  $\Leftrightarrow$  a[i].
```

En fait, la notation `a[i]` n'est qu'une abréviation de `*(a + i)`. Sachant que l'addition est commutative (`*(a + i) = *(i + a)`), on peut accéder à la *i*ème case du tableau `a` avec `i[a]` (abréviation de `*(i + a)`)⁵.

Attention: L'instruction `a = p;` est interdite car `a` est un pointeur constant donc ne peut être modifié. Ceci veut dire aussi que si nous avons la déclaration `int a[10], b[10];` on ne peut pas écrire `a = b;` Mais on peut écrire: `*a = *b` qui équivaut à `a[0] = b[0]`.

Exercice 12 : Copier un tableau dans un autre: `copier(t1, t2, n)` avec *t₁* le tableau source, *t₂* le tableau destination et *n* le nombre d'éléments à copier.

A l'appel de cette fonction, on peut vouloir copier les *N* premiers éléments (de `a` vers `b`): `copier(a, b, N)`; ou tous les éléments à partir du *N*ème: `copier(a + N, b, L - N)`; (si *L* est la taille du tableau `a`).

10.2 Tableau à deux dimensions

Soit `t` un tableau à 2 dimensions décalé comme suit: `int t[5][10];` L'élément d'indices *i* et *j* se note `t[i][j]`. L'affectation `p = t` (avec `int **p`: pointeur sur un pointeur d'entier) fait pointer `p` sur le premier élément de la première ligne de `t`. Après l'instruction `p++;` `p` pointe sur le premier élément de la deuxième ligne.

Dans le cas d'un paramètre de fonction, il faut donner la longueur de chaque ligne:

```
... f(...int t[ ][10]...);
```

Exercice 13 : Copier un tableau à 2 dimensions dans un autre.

Exercice 14 : Copier une chaîne dans une autre.

Exercice 15 : Concaténer deux chaînes *S₁* et *S₂* dans une troisième chaîne *S₃* en rendant la longueur de *S₃*.

Exercice 16 : Concaténer *S₁* et *S₂* dans *S₃* sans déclarer *S₃* à l'avance mais en rendant son adresse.

11 Structures

L'équivalent du **record** en PASCAL.

▷ Exemple : Définition de la structure `date` :

```
struct date {
```

⁵. Avouez que ça donne le vertige!

```

    int j;
    char * m;
    int a;
};

```

Déclaration: struct date d; définit une variable d de type date.
ou bien

```

    struct date {
        int j;
        char * m;
        int a;
    } d;

```

Exemples d'utilisation: d.j = 25;, d.m = "septembre"; d.a = 1994;

La seule opération qu'on peut effectuer sur des structures est l'affectation globale.

Pointeur vers une structure:

Soit p un pointeur vers struct date, (struct date * p;)

L'accès au jour et à l'année de la structure pointée par p se fait ainsi:

(*p).j et (*p).a

L'écriture abrégée correspondant à ceci est la suivante:

p ->j ⇔ (*p).j et p ->a ⇔ (*p).a

▷ Exemple :

```

f(p) struct date * p; {
    ...
    printf("jour = %d\n", p -> j) ;
    ...
}

```

11.1 Structures imbriquées

▷ Exemple :

```

struct individu {
    char nom [50];
    char prenom [50];
    struct date dnaiss;
};

```

Donc i.dnaiss.a est l'année de naissance de l'individu i.

Si nous avons la déclaration suivante: struct individu * p;
p ->dnaiss.a donne l'année de naissance de l'individu pointé par p.

Exercice 17 : Saisir une liste d'individus puis les trier par ordre alphabétique en échangeant des pointeurs plutôt que des individus.

11.2 Structures de bits

Permettent de découper une donnée en zones de bits.

▷ Exemple :

```
struct deux_octets {
    unsigned x : 8; /* zone de 8 bits */
    unsigned y : 4; /* zone de 4 bits */
    unsigned z : 1;
    unsigned n : 3;
}
```

on peut aussi découper un octet en bits :

```
struct un_octet {unsigned b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1,
b8:1;} x;
```

On peut utiliser alors le bit 4 en écrivant `x.b4`.

11.3 Enumérations

Permettent surtout d'écrire des programmes lisibles.

▷ Exemple : `enum mois {janvier, fevrier, mars, avril, ..., novembre, decembre};`⁶

La déclaration se fait comme suit : `enum mois m;`

Sauf indication contraire : `janvier = 0`, `fevrier = 1`, `mars = 2`, etc.

C'est-à-dire que les affectations `m = fevrier`; et `m = 1`; sont équivalentes.

On peut modifier la numérotation automatique. Par exemple :

```
enum semaine {lundi = 1, mardi, mercredi, jeudi = 0, vendredi, samedi,
dimanche};
```

Dans ce cas, `lundi = vendredi = 1`.

On ne peut pas écrire directement une variable de type énumération.

Dans le cas du type `enum semaine` par exemple, il faut passer par l'intermédiaire d'un tableau de chaînes de caractères : `char * jour[7] = {"lundi", "mardi", ..., "dimanche"};` Pour afficher le jour `j` (déclaré par `enum semaine j;`), il faut écrire : `printf("%s", jour[j]);`

11.4 Unions

Permettent de faire un choix entre plusieurs champs d'une structure.

6. Attention : Ne pas confondre entre l'identificateur `janvier` et la chaîne `"janvier"`.

▷ Exemple :

```
union donnee {
    int i; /* c'est soit un entier, soit */
    int x[L]; /* un tableau de L entiers, soit */
    char c; /* un caractere selon le cas */
};
```

```
union donnee d;
```

On peut utiliser d.i, d.x[j] ou d.c mais pas les trois en même temps.

L'utilisation des *unions* est souvent couplée, dans une structure, avec un champ de choix.

▷ Exemple :

```
struct donnee {
    char indicateur; /* Indique le type de l'information */
    union {
        float f;
        char c;
    } information ;
} x;
```

Selon la valeur de x.indicateur on accède à x.information.f ou à x.information.c.

▷ Autre exemple: Définir un complexe représenté dans les deux systèmes polaire et cartésien:

```
typedef enum { cartesein, polaire } SYSTEME;
typedef struct complexe {
    SYSTEME systeme;
    union {
        struct { float re, im; } cart;
        struct { float rho, theta; } pol;
    } cart_pol;
} COMPLEXE;
```

La partie réelle d'une variable Z de type COMPLEXE s'écrit: Z.cart_pol.cart.re
Ecriture trop lourde pour être insérée dans un programme. Pour remédier à cette lourdeur, nous utilisons la directive #define.

```
#define RE    cart_pol.cart.re
#define IM    cart_pol.cart.im
#define RHO   cart_pol.pol.rho
#define THETA cart_pol.pol.theta
```

La partie réelle de Z s'écrit donc Z.RE, la partie imaginaire Z.IM, etc.

12 Complément sur les pointeurs

12.1 Les listes

Il est parfois plus intéressant d'utiliser une liste chaînée, pour représenter une structure de données, que d'utiliser un tableau.

▷ Exemple : Etant donnée une suite d'entiers (leur nombre n'est pas connu à l'avance), créer une liste qui contient les éléments de cette suite.

Définition du type liste :

```
typedef struct element {
    int Entier;
    struct element * EntierSuivant;
} ELEMENT, *LISTE;
```

Ceci veut dire que ELEMENT est le synonyme du type struct element et que LISTE est le synonyme du type struct element *.

Cette définition est une abréviation de :

```
struct element {
    int Entier;
    struct element * EntierSuivant;
};
```

```
typedef struct element ELEMENT;
typedef struct element *LISTE;
```

Construction de la liste : chaque nouvel élément est placé en tête de la liste.

```
#include<stdio.h>
#include <malloc.h>
... /* Texte de la definition des type ELEMENT et LISTE */

LISTE construire () {
    int x;
    LISTE L = NULL, /* La tete de la liste en construction. */
    p; /* Pointeur qui recoit le nouvel element. */
    while (scanf("%d", &x) != EOF) { /* Tant qu'il y a un entier a lire.
        Fin de saisie par ^D (fin de fichier d'entree standard) */
        p = (LISTE) malloc(sizeof(ELEMENT)) ; /* (LISTE)malloc : conversion de type necessaire
        car malloc est du type (char *) */
        p -> Entier = x; /* Le nouvel entier mis a sa place. */
        p -> EntierSuivant = L; /* Le nouvel element raccorde a la liste existante. */
        L = p; /* Mise-a-jour de la tete de liste. */
    }
    return L;
}
```

```

void afficher(L) LISTE L; {
    while (L != NULL) {
        printf ("%d ", L -> Entier);
        L = L -> EntierSuivant;
    }
}

main () {
    afficher(construire());
}

```

Exercice 18 : Trier, par ordre croissant, une suite d'entiers (utilisation de l'arbre binaire de recherche).

L'idée est de mettre cette suite dans un arbre binaire telque pour chaque nœud d'étiquette l'entier N on ait les éléments inférieurs ou égaux à N dans le sous arbre gauche et les éléments supérieurs à N dans le sous arbre droit.

Exercice 19 : Ecrire une fonction `evaluer(Exp)` qui évalue une expression. `Exp` est une expression représentée par un arbre binaire dont les nœuds sont des opérateurs binaires et les feuilles des nombres. Exemple : L'expression $3 * (4 + 5) / 7 - 9$ sera représentée par l'arbre dont l'écriture termale est : $-(/ (* (3, +(4, 5)), 7), 9)$.

12.2 Tableau de chaînes de caractères

`char * t[5]`; définit un tableau de 5 pointeurs vers des caractères (autrement dit, un tableau de 5 chaînes de caractères).
On peut avoir comme initialisation: `char * t[7] = {"lundi", "mardi", ..., "dimanche"};`

Dans ce cas, `**t` et `*t[0]` ont la même valeur 'l'.

La fonction principale `main` a potentiellement deux principaux paramètres : `argc` et `argv`.

«-- `argv` est un tableau de pointeurs vers les paramètres.

«-- `argc` est la taille du tableau `argv`.

12.3 Tableau de fonctions

Reprenons l'exercice (lire deux nombres et un signe (+, -, *, / ou %)
puis effectuer l'opération correspondante) et définissons les fonctions correspondant à ces opérations :

```

int plus    (x, y) int x, y; { return x + y; }
int moins  (x, y) int x, y; { return x - y; }

```

```
int mult    (x, y) int x, y; { return x * y; }
int quotient(x, y) int x, y; { return x / y; }
int reste   (x, y) int x, y; { return x % y; }
```

Ces fonctions peuvent être mises dans un tableau et appelées par leurs indices :

```
définition et initialisation: int (*fonc[])() = {plus, moins, mult,
quotient, reste};
```

Le programme est alors le suivant :

```
main() {
    int x, y, Op;
    scanf("%d %d %d", &x, &Op, &y); /* maintenant Op est un indice de 0 a 4 */
    printf("%d", (*fonc[Op])(x, y));
}
```

13 Entrées/Sorties

Un fichier est un pointeur vers un type FILE défini dans la bibliothèque standard `stdio.h` : `FILE * f;`

Fichiers standards :

```
«-- stdin: fichier standard d'entrée (standard input file)
«-- stdout: fichier standard de sortie (standard output file)
«-- stderr: fichier standard de sortie d'erreurs (standard error
file)
```

Principales opérations :

```
«-- fopen: ouverture d'un fichier. Cette fonction a pour en-tête:
FILE * fopen(nom_fichier, mode) char * nom_fich, * mode;
fopen ouvre le fichier, de nom externe nom_fichier, dans le
mode indiqué en second paramètre (mode = "r" pour une ouverture
en lecture (read), "w" pour une ouverture en écriture (write)
et "a" pour une ouverture en extension par la fin (append))7.
Le résultat de cette fonction est un FILE *. Si l'ouverture
s'est mal passée, fopen renvoie la constante NULL.
Voici une bonne utilisation de fopen:
FILE * f;
...
if ((f = fopen(nom_fichier, mode)) == NULL) {
    fprintf(stderr, "%s : ouverture mal operée\n", nom_fichier);
    exit(un code de retour);
}
«-- fclose: fermeture d'un fichier: fclose(variable_fichier);
fclose renvoie 0 si la fermeture s'est bien passée, la constante
EOF sinon.
```

7. En C, il n'y a pas que ces 3 modes

```

«-- fread: lecture d'un ou plusieurs éléments dans un fichier.
Elle a pour en-tête:
int fread(ptr, taille, nombre, fichier)
char * ptr; int taille; int nombre; FILE * fichier;
fread lit dans la donnée pointée par ptr, nombre élément(s)
de taille taille depuis le fichier fichier. Elle renvoie le
nombre d'éléments lus.

«-- fwrite: écriture d'un ou plusieurs éléments dans un fichier.
Elle a pour en-tête:
int fwrite(ptr, taille, nombre, fichier)
char * ptr; int taille; int nombre; FILE * fichier;
fwrite écrit, dans le fichier fichier, nombre fois la donnée
de taille taille pointée par ptr. Elle renvoie comme résultat
le nombre d'éléments écrits.

«-- fscanf: opère comme la fonction scanf sauf qu'ici, le fichier
de lecture est spécifié (scanf(...) ⇔ fscanf(stdin, ...)).

«-- même chose pour les fonctions: fprintf, putc, getc.
printf(...) ⇔ fprintf(stdout, ...), putchar(c) ⇔ putc(c,
stdout) et getchar() ⇔ getc(stdin) (getc rend EOF quand la
fin du fichier est atteinte).

```

Exercice 20 : *Ecrire un programme qui lit les noms de deux fichiers d'entiers puis réalise la copie de l'un vers l'autre.*

Exercice 21 : *Réécrire en C la commande UNIX cp qui réalise la copie d'un ou plusieurs fichiers dans un autre. Le dernier fichier passé en paramètre est le fichier récepteur. (passage de paramètres à un programme).*

Exercice 22 : *Ecrire un programme qui crée un fichier d'individus (comme défini plus haut).*

14 Compilation séparée et création de bibliothèques

L'objet de section est de montrer comment peut-on constituer une ou plusieurs bibliothèques de fonctions et en faire appel dans un ou plusieurs programmes. Les propos de cette section dépendent fortement du compilateur et de la plateforme utilisée. Les exemples donnés ici font référence au compilateur C sous Unix.

Cette méthode dite de compilation séparée n'a aucun rapport avec le fait de scinder un programme en plusieurs fichiers qu'on rassemble par des # include pour former le programme principal. Cette façon de faire force, à chaque fois, la compilation de tous les bouts de programmes qui ont été écrits, testés et donc 'sans erreurs'. C'est donc une méthode couteuse et dépassée.

Il s'agit donc de simuler le fonctionnement de ce qu'on utilise depuis toujours quand on inclu des bibliothèque de fonctions d'entrées/sorties ou de calcul mathématique.

Nous allons illustrer ceci par un exemple. Nous voulons développer une bibliothèque de fonctions de traitement des nombres complexes. Pour ne pas charger le discours, nous nous contenterons des deux fonctions : addition de deux complexes ($\text{add}(z_1, z_2)$) et norme d'un complexe ($\text{norme}(z)$). Pour cela, nous devons créer un fichier qui regroupera la définition du type complexe et les en-têtes des deux fonctions `add` et `norme`. Soit `complexe.h` ce fichier. Le fichier `complexe.h` sera inclu dans tout programme utilisant les complexes par le biais de la directive `#include`.

Voici le contenu du fichier `complexe.h` :

```
typedef struct {float re, im} complexe;
```

```
complexe add(complexe z1, complexe z2)
float norme(complexe z)
```

Puis il nous faut définir effectivement les 2 fonctions dans un fichier nommé, par exemple, `complexe.c` :

```
#include <math.h>      /* a cause de la fonction 'sqrt' */
#include "complexe.h"
```

```
complexe add(complexe z1, complexe z2) {
    complexe z;
    z.re = z1.re + z2.re;
    z.im = z1.im + z2.im;
    return z;
}
```

```
float norme(complexe z) {
    return sqrt(z.re * z.re + z.im * z.im);
}
```

La compilation de `complexe.c` (sans édition de liens : `cc -c complexe.c`) fournit un fichier *objet* nommé `complexe.o` qui contient la traduction en langage machine des 2 fonctions qui y sont définies. Ce fichier n'est pas exécutable comme il est.

L'utilisation dans des programmes des fonctions définies plus haut devient simple. Voici un exemple : Le programme suivant consiste à faire la somme de 2 complexes et puis fournir la norme de cette somme.

```
/* Programme exemple.c */
#include <stdio.h>
#include "complexe.h"
```

```
main () {
    complexe x = {2.1, 3.5};
```

```

    complexe y = {1.0, 4.5};
    complexe z;
    z = add(x, y);
    printf("somme = (%5.2f , %5.2f). norme = %5.2f\n", z.re, z.im, norme(z));
}

```

Pour produire le fichier objet de ce programme, il suffit de compiler avec : `cc -c exemple.c`

L'exécutable peut être produit en combinant les 2 fichiers objets :
`cc -o exemple exemple.o complexe.o`

Une autre méthode plus intéressante consiste à regrouper plusieurs fichiers objets dans un même module appelé *bibliothèque* (ou librairie). C'est le cas par exemple d'une librairie de calcul numérique où on regroupe plusieurs modules ou fichiers objets chacun portant sur un sujet. On peut ainsi y regrouper un module sur le calcul matriciel, un autre sur le calcul polynomial, etc.

Un exemple de librairie standard est celui de la librairie mathématique qui regroupe une panoplie de fonctions telles que les fonctions trigonométriques. Cette librairie a pour nom `libm.a` et se situe dans l'emplacement standard des bibliothèques : `/usr/lib`. L'inclusion d'une telle librairie pour produire un exécutable se fait de la manière suivante :

```
cc -o exemple exemple.o -lm
```

Remarquons que le préfixe 'lib' et le suffixe '.a' ont disparu de la ligne de commande.

Pour créer sa propre librairie de nom `libCalcul.a`, à partir d'un certain nombre de fichiers objets, il suffit de lancer la commande :

```
ar vrc libCalcul.a complexe.o matrice.o polynome.o ...
```

où `complexe.o` est le module défini plus haut, `matrice.o` est le module du calcul matriciel et `polynome.o` un module de calcul polynomial. On peut archiver dans une librairie autant de modules qu'on souhaite⁸. L'utilisation d'une telle librairie se fait comme suit :

```
cc -o programme programme.o -lCalcul
```

Remarque : Une librairie `x` (utilisée à l'aide de la directive de compilation `-l`) doit toujours être créée dans le fichier `libx.a`.

15 Fonctions à nombre variable de paramètres

Remarquons que certaines fonctions de la bibliothèque peuvent être appelées avec un nombre variable de paramètres suivant l'utilisation.

⁸. Se référer au manuel en ligne de la commande `ar` pour plus d'informations

C'est le cas par exemple des deux fameuses fonctions d'entrée/sortie: `printf` et `scanf`.

Ce privilège n'est pas réservé aux seules fonction prédéfinies. Il nous est possible de définir des fonctions sans avoir, au préalable, connaissance du nombre d'arguments. Mais ces fonctions doivent être définies telles qu'à l'appel le nombre des paramètres effectifs ainsi que le type de chacun d'eux soient connus. Pour la fonction `scanf`, par exemple, la chaîne (premier argument) donne le nombre de paramètres par les formats `%x` ainsi que les types (`%d` pour `int`, `%f` pour `float`, etc).

A l'appel, les paramètres effectifs sont mis dans une pile que l'on peut manipuler grâce à un certain nombre d'opérations. Dans la suite, nous donnerons les deux façons de faire (dans l'ancienne et dans la nouvelle norme).

15.1 Dans l'ancienne norme

Il faut d'abord inclure, par `#include`, le fichier de la bibliothèque standard `varargs.h`.

En-tête :

```
type nom fonction(va_alist) va_dcl {  
    ...  
}
```

Avec *type* le type du résultat de la fonction, *va_alist* (*variable argument list*) le mot clé qui remplace les noms des paramètres formels et *va_dcl* (*variable declaration*) le mot clé qui remplace les déclarations des paramètres.

Corps de la fonction :

La pile où sont stockés les paramètres est pointée par une variable de type `va_list`. Exemple de déclaration: `va_list vp`;

Pour naviguer dans cette pile, il faut d'abord faire appel à la fonction `va_start(vp)` qui initialise `vp` au début de la pile. En suite, la fonction `va_arg(vp, type)` renvoie le prochain élément de la pile dont le type est supposé être *type*. L'appel `va_end(vp)` termine la visite de la pile.

15.2 Dans la nouvelle norme

Il faut d'abord inclure, par `#include`, le fichier de la bibliothèque standard `stdarg.h`. A part l'en-tête de la fonction, le reste est identique:

En-tête :

```
type nom fonction(type1 arg1, ..., typen argn, ...) {
```

```
...  
}
```

Les paramètres arg_1, \dots, arg_n sont dit *partie fixe* de la fonction. La fin de l'en-tête '...' est dite *partie variable* ou *ellipse*.

Exemple: printf a comme prototype: `int printf(char * Format, ...)`.

▷ Exemple : la fonction `execv` permet de faire appel à un programme exécutable (une commande par exemple) depuis un programme C. Nous proposons d'en faire appel avec, à priori, un nombre inconnu d'arguments⁹.

```
#include <stdarg.h>  
#define MAXARGS 20  
...  
mon_execl(char * Prog, ...) {  
    va_list ArgPointeur;  
    char *Args[MAXARGS];  
    int ArgNbr = 0;  
    va_start (ArgPointeur);  
    Prog = va_arg(ArgPointeur, char *);  
    while ((Args[ArgNbr++] = va_arg(ArgPointeur, char *)) != NULL);  
    va_end (ArgPointeur);  
    return execv(Prog, Args);  
}
```

9. Ceci est déjà fourni par la fonction `execl`.

Références

- « -- B.W. KERNIGHAN et D.M. RITCHIE, ‘‘Le langage C : ANSI’’, Collection MIM, Ed. MASSON. (traduction française)
- « -- C.L. TONDO et S.E. GIMPEL, ‘‘Le langage C : Solutions aux exercices de l’ouvrage de B.W. Kernighan et D.M. Ritchie’’, Collection MIM, Ed. MASSON. (traduction française)
- « -- A.R. FEUER, ‘‘Langage C : Problèmes et exercices’’, Collection MIM, Ed. MASSON. (traduction française)
- « -- PH. DRIX, ‘‘Langage C : norme ANSI, vers une approche orientée objet’’, Collection MIM, Ed. MASSON.
- « -- J.M. RIFFLET, ‘‘la programmation sous UNIX’’, 3ème édition, Ed. Ediscience.
- « -- C. DELANNOY, ‘‘Programmer en Langage C’’, Ed. Eyrolles.
- « -- C. DELANNOY, ‘‘Exercices en langage C’’, Ed. Eyrolles.
- « -- ‘‘Le langage C’’, Ed. PC POCHE.