I. LES BASES DE LA PROGRAMMATION EN C.

I.1. Généralité.

a) Historique.

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language*. Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

b) La compilation.

Le C est un langage *compilé* (par opposition aux langages interprétés). Cela signifie qu'un programme C est décrit par un fichier texte, appelé *fichier source*. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé *compilateur*. La compilation se décompose en fait en 4 phases successives :

1. **Le traitement par le préprocesseur :** le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source…).
2. **La compilation :** la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.
3. **L'assemblage :** cette opération transforme le code assembleur en un fichier binaire, c'est-`a-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé *fichier objet*.
4. **L'édition de liens :** un programme est souvent subdivisé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit *exécutable*.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par ***.c***, les fichiers prétraités par le préprocesseur par ***.i***, les fichiers assembleur par ***.s***, et les fichiers objet par ***.o***. Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe ***.a***.

c) Les composants élémentaires du C.

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

1. les identificateurs,

2. les mots-clefs,
3. les constantes,
4. les chaînes de caractères,
5. les opérateurs,
6. les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

c.1) Les identificateurs.
Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner

- ✓ un nom de variable ou de fonction,
- ✓ un type défini par typedef, struct, union ou enum,
- ✓ une étiquette.

Un identificateur est une suite de caractères parmi :

- ✓ les lettres (minuscules ou majuscules, mais non accentuées),
- ✓ les chiffres,
- ✓ le "blanc souligné" (tiret de 8).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Il est cependant déconseillé d'utiliser comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C. Les majuscules et minuscules sont différenciées.

c.2) Les mots-clefs
Un certain nombre de mots, appelés *mots-clefs*, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 33 mots clefs qui sont:

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

On peut les ranger en catégories.

- ✓ **Les spécificateurs de stockage :**
  auto ; register ; static ; extern.
- ✓ **Les spécificateurs de type :**
  bit ; char ; double ; float ; int ; long ; short ; signed ; unsigned ; void ;
- ✓ **Les constructeurs.**
  enum; union; struct; typedef.
- ✓ **Les qualificateurs de type :**
  const ; volatile.
- ✓ **Les instructions de boucle.**
  do; for; while.
- ✓ **Les instructions de selection.**
  case; default; else; if; switch.
- ✓ **Les instructions de rupture de séquence.**
  break; continue; goto; return.
- ✓ **Diver:**

sizeof.

c.3) Les commentaires.

Il existe deux types de commentaires :

- ✓ *Les commentaires uni-ligne*. Ils sont indiqués par une double barre.
  Exemple :
  ```
  // Voici un commentaire sur une seul ligne.
  ```
- ✓ *Les commentaires multi-lignes.* Un commentaire multi-ligne débute par /* et se termine par */.
  Exemple.
  ```
  /* Ceci est un commentaire
   ainsi que celui-ci celui-ci
   celui-ci celui-ci celui-ci celui-ci  */
  ```

d) Structure d'un programme C.

Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple x = 0

ou bien

(i >= 0) && (i < 10) && (p[i] != 0)

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression". Plusieurs instructions peuvent être rassemblées par des accolades *{* et *}* pour former une *instruction composée* ou *bloc* qui est syntaxiquement équivalent à une instruction. Par exemple,

```
if (x != 0)
{
    z = y / x;
    t = y % x;
}
```

Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés

par une virgule est une *déclaration*. Par exemple,

```
int a;
int b, c;
double x = 2.38e4;
char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Un programme C se présente ainsi de la façon suivante :

Directives au préprocesseur

Déclarations de variables externes

Fonctions secondaires

```
main()          //Fonction principale
{
        Déclarations de variables internes
        Instructions
}
```

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale.

e) Les conventions d'écriture d'un programme C.

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes difficilement lisibles. Aussi existe-t-il un certain nombre de conventions.

- ✓ On n'écrit qu'une seule instruction par ligne : le point-virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.
- ✓ Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.
- ✓ On laisse un blanc
  - o entre les mots-clefs if, while, do, switch et la parenthèse ouvrante qui suit,
  - o après une virgule,
  - o de part et d'autre d'un opérateur binaire.
- ✓ On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
- ✓ Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées.

I.2. Les types prédéfinis.

Le C est un langage *typé*. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire. La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son *adresse*, qui est un entier. Deux octets contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

La taille mémoire correspondant aux différents types dépend des compilateurs ; toutefois, la norme ANSI spécifie un certain nombre de contraintes. Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants : Char ; int ; float ; double ; short ; long ; unsigned.

I.5.1 Le type caractère.

Le mot-clef ***char*** désigne un objet de type caractère. Un char peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. La plupart du temps, un objet de type char est codé sur un octet ; c'est l'objet le plus élémentaire en C. Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits). La plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits), dont les 128 premiers caractères correspondent aux caractères ASCII. Les 128 derniers caractères (codés sur 8 bits) sont utilisés pour les caractères propres aux différentes langues.

Une des particularités du type ***char*** en C est qu'il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier.

Par exemple, si c est de type char, l'expression c + 1 est valide. Elle désigne le caractère suivant dans le code ASCII.

Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
        char c = 'A';
        printf("%c", c + 1);
}
```

Suivant les implémentations, le type char est signé ou non. En cas de doute, il vaut mieux préciser ***unsigned char*** ou ***signed char***. Notons que tous les caractères imprimables sont positifs.

I.5.2 Les types entiers.

Le mot-clef désignant le type entier est ***int***. Un objet de type int est représenté par un mot 32 bits pour un DEC alpha ou un PC Intel. Le type int peut être précédé d'un attribut de précision (***short*** ou ***long***) et/ou d'un attribut de représentation (***unsigned***).

Un objet de type ***short int*** a au moins la taille d'un char et au plus la taille d'un int. En général, un short int est codé sur 16 bits.

Un objet de type ***long int*** a au moins la taille d'un int (64 bits sur un DEC alpha, 32 bits sur un PC Intel) (tableau ci-dessous).

	DEC Alpha	PC Intel (Linux)	
`char`	8 bits	8 bits	caractère
`short`	16 bits	16 bits	entier court
`int`	32 bits	32 bits	entier
`long`	64 bits	32 bits	entier long
`long long`	n.i.	64 bits	entier long (non ANSI)

`signed char`	$[-2^7; 2^7[$
`unsigned char`	$[0; 2^8[$
`short int`	$[-2^{15}; 2^{15}[$
`unsigned short int`	$[0; 2^{16}[$
`int`	$[-2^{31}; 2^{31}[$
`unsigned int`	$[0; 2^{32}[$
`long int` (DEC alpha)	$[-2^{63}; 2^{63}[$
`unsigned long int` (DEC alpha)	$[0; 2^{64}[$

I.5.3 Les types flottants.

Les types ***float***, ***double*** et ***long double*** servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

	DEC Alpha	PC Intel	
`float`	32 bits	32 bits	flottant
`double`	64 bits	64 bits	flottant double précision
`long double`	64 bits	128 bits	flottant quadruple précision

I.5.4. Les constantes.

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération.

Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

a) Les constantes entières.

Une constante entière peut être représentée de 3 manières différentes suivant la base dans laquelle elle est écrite :

- ✓ ***Décimale*** **:** par exemple, 0 et 2437348 sont des constantes entières décimales.
- ✓ ***Binaire :*** la représentation binaire d'un entier correspond à sa décomposition en base 2. Les constantes binaires doivent commencer par un ***0b*** ou ***0B***. Par exemple, les représentations binaires des entiers 0 et 255 sont respectivement ***0b0*** et ***0b11111111***.
- ✓ ***Hexadécimale :*** la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de ***a*** à ***f*** sont utilisées pour représenter les nombres de 10 à 15. Les constantes hexadécimales doivent commencer par ***0x*** ou ***0X***. Par exemple, les représentations hexadécimales de 14 et 255 sont respectivement ***0xe*** et ***0xff***.

b) Les constantes réelles.

Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre e ou E ; il s'agit d'un nombre décimal éventuellement signé.

Exemple : 12.25 ; 24.11e5 ;

c) Les constantes caractères
Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. ’A’ ou ’$’). Les seuls caractères imprimables qu’on ne peut pas représenter de cette façon sont l’antislash et l’apostrophe, qui sont respectivement désignés par \\ et \’. Le point d’interrogation et les guillemets peuvent aussi être désignés par les notations \? et \". Les caractères non imprimables peuvent être désignés par’\x*code-hexa*’ où *code-hexa* est le code en hexadécimal du caractère. Par exemple, ’\33’ et ’\x1b’ désignent le caractère *escape*. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d’une notation plus simple :

\n nouvelle ligne

\r retour chariot

\t tabulation horizontale

\f saut de page

\v tabulation verticale

\a signal d’alerte

\b retour arrière

d) Les constantes chaînes de caractères.
Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple, "*Ceci est une chaîne de caractères*"

Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple,

"ligne 1 \n ligne 2"

A l’intérieur d’une chaîne de caractères, le caractère " doit être désigné par \". Enfin, le caractère \ suivi d’un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple,

"ceci est une longue longue longue longue longue longue longue \

chaîne de caractères"

1.3. Les opérateurs

I.6.1 L’affectation

En C, l’affectation est un opérateur à part entière. Elle est symbolisée par le signe =. Sa syntaxe est la suivante :

variable = expression

Le terme de gauche de l’affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d’évaluer *expression* et d’affecter la valeur obtenue à *variable*. De plus, cette expression possède une valeur, qui est celle *expression*. Ainsi, l’expression i = 5 vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant

```
main()
{
        int i, j = 2;
        float x = 2.5;
        i = j + x;
        x = x + i;
        printf("\n %f \n",x);
}
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction i = j + x; , l'expression j + x a été convertie en entier.

I.6.2 Les opérateurs arithmétiques.

Les opérateurs arithmétiques classiques sont l'opérateur unaire

- ✓ - (changement de signe)

ainsi que les opérateurs binaires :

- ✓ + (addition) ;
- ✓ - (soustraction) ;
- ✓ * (multiplication) ;
- ✓ / (division) ;
- ✓ % (reste de la division « modulo »).

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- ✓ Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple :
  float x;
  x = 3 / 2;
  affecte à x la valeur 1. Par contre :
  x = 3 / 2.0;
  affecte à x la valeur 1.5.
- ✓ L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.
- ✓ Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction pow(x,y) de la librairie math.h pour calculer x^y.

I.6.3 Les opérateurs relationnels

- ✓ > strictement supérieur ;
- ✓ >= supérieur ou égal ;
- ✓ < strictement inférieur ;
- ✓ <= inférieur ou égal ;
- ✓ == égal ;
- ✓ != différent ;

Leur syntaxe est :

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Attention à ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =. Ainsi, le programme :

I.6.4 Les opérateurs logiques booléens

- ✓ && et logique ;
- ✓ || ou logique ;
- ✓ ! négation logique ;

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon. Dans une expression de type :

expression-1 op-1 expression-2 op-2 ...expression-n

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans :

int i;

int p[10];

if ((i >= 0) && (i <= 9) && !(p[i] == 0))...

la dernière clause ne sera pas évaluée si i n'est pas entre 0 et 9.

I.6.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

& et	| ou inclusif
^ ou exclusif	~ complément à 1
<< décalage à gauche	>> décalage à droite.

L'opérateur unaire ~ change la valeur de chaque bit d'un entier. Les décalages à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît). Considérons par exemple les entiers a=77 et b=23 de type unsigned char (*i.e.* 8 bits). En base 2 ils s'écrivent respectivement 01001101 et 00010111.

expression	valeur binaire	valeur décimale	
a	01001101	77	
b	00010111	23	
a & b	00000101	5	
a | b	01011111	95	
a ^ b	01011010	90	
~a	10110010	178	
b << 2	01011100	92	multiplication par 4
b << 5	11100000	112	ce qui dépasse disparaît
b >> 1	00001011	11	division entière par 2

I.6.6 Les opérateurs d'affectation composée.

Les opérateurs d'affectation composée sont :

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

Pour tout opérateur *op*, l'expression

expression-1 op= expression-2

est équivalente à

expression-1 = expression-1 op expression-2

Toutefois, avec l'affection composée, *expression-1* n'est évaluée qu'une seule fois.

I.6.7 Les opérateurs d'incrémentation et de décrémentation.

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en suffixe (i++) qu'en préfixe (++i). Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple :

```
int a = 3, b, c;

b = ++a; /* a et b valent 4 */

c = b++; /* c vaut 4 et b vaut 5 */
```

I.6.8 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

expression-1, expression-2, ... , expression-n

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme

```
main()
```

```
{
        int a, b;
        b = ((a = 3), (a + 2));
        printf("\n b = %d \n",b);
}
```

imprime b = 5.

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie. Par exemple l'instruction composée

```
{
        int a=1;
        printf("\%d \%d",++a,a);
}
```

(compilée avec gcc) produira la sortie 2 1 sur un PC Intel/Linux et la sortie 2 2 sur un DEC Alpha/OSF1.

I.6.9 L'opérateur conditionnel ternaire

L'opérateur conditionnel ? est un opérateur ternaire. Sa syntaxe est la suivante :

condition ? *expression-1* : *expression-2*

Cette expression est égale à *expression-1* si *condition* est satisfaite, et à *expression-2*

sinon. Par exemple, l'expression :

x >= 0 ? x : -x

correspond à la valeur absolue d'un nombre. De même l'instruction

m = ((a > b) ? a : b);

affecte à m le maximum de a et de b.

I.6.10 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet. On écrit

(*type*) *objet*

Par exemple :

```
main()
{
        int i = 3, j = 2;
```

```
    printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

I.6.11 L'opérateur adresse

L'opérateur d'adresse & appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est :

&*objet*

I.6.12. L'opérateur *sizeof*.

L'opérateur *sizeof*, dont l'emploi ressemble à celui d'une fonction, fournit la taille en octets. Par exemple, dans une implémentation où le type *int* est représenté sur 2 octets et le typr *double* sur 8 octets, si l'on suppose que l'on a affaire à ces déclarations :

```
Int n ;

double z ;
```

- ✓ sizeof(n) vaudra 2.
- ✓ sizeof(z) vaudra 8.

Cet opérateur peut également s'appliquer à un type de nom donné. Exemple :

- ✓ sizeof(int) vaut 2 ;
- ✓ sizeof(double) vaut 8.

I.6.13. Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est défini par la flèche de la seconde colonne du tableau. On préfèrera toutefois mettre des parenthèses en cas de doute.

opérateurs	
`()  []  ->  .`	→
`!  ~  ++  --  -(unaire)  (type)  *(indirection)  &(adresse)  sizeof`	←
`*  /  %`	→
`+  -(binaire)`	→
`<<  >>`	→
`<  <=  >  >=`	→
`==  !=`	→
`&(et bit-à-bit)`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`? :`	←
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	←
`,`	→

Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut mettre en parenthèse les expressions. Par exemple, il faut écrire :

if ((x ^ y) != 0)

I.4. Les instructions de contrôle.

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les *instructions de branchement* et les *boucles*.

I.4.1. Les instructions de branchement conditionnel.

Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

a) Branchement conditionnel if---else

La forme la plus générale est celle-ci :

if (*expression-1*)

instruction-1

else

instruction-2.

Le else est facultatif.

La forme la plus simple est :

if (*expression*)

instruction

Chaque *instruction* peut être un bloc d'instructions.

b) Branchement multiple switch.

Sa forme la plus générale est celle-ci :

switch (*expression*)

{

case *constante-1*:

liste d'instructions 1

break;

case *constante-2*:

liste d'instructions 2

break;

...

```
        case constante-n:
                liste d'instructions n
        break;
        default:
                liste d'instructions n+1
        break;
}
```

Si la valeur de *expression* est égale à l'une des *constantes*, la *liste d'instructions* correspondant est exécutée. Sinon la *liste d'instructions n+1* correspondant à default est exécutée. L'instruction default est facultative.

I.4.2. Les instructions de branchement non conditionnel.

a) Branchement non conditionnel break.

On a vu le rôle de l'instruction break; au sein d'une instruction de branchement multiple switch. L'instruction break peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, break fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main()
{
        int i;
        for (i = 0; i < 5; i++)
        {
                printf("i = %d\n",i);
                if (i == 3)
                break;
        }
        printf("valeur de i à la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
```

valeur de i à la sortie de la boucle = 3

b) Branchement non conditionnel *continue*

L'instruction *continue* permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
        int i;
        for (i = 0; i < 5; i++)
        {
                if (i == 3)
                        continue;
                printf("i = %d\n",i);
        }
        printf("valeur de i à la sortie de la boucle = %d\n",i);
}
```

Imprime

i = 0

i = 1

i = 2

i = 4

valeur de i à la sortie de la boucle = 5

c) Branchement non conditionnel goto

L'instruction *goto* permet d'effectuer un saut jusqu'à l'instruction *etiquette* correspondant.

Exemple : Le programme suivant :

```
        main
        {
                int i ;
                for( i = 1 ; i <= 10 ; i++)
                {
                        printf("début tour %d\n", i) ;
                        printf("bonjour\n") ;
                        fi( i == 3 )
```

```
                    goto sortie;
                printf("fin tour %d\n, i);
            }
            sortie : printf("après la boucle") ;
        }
```

Imprime :

début tour 1

bonjour

fin tour 1

début tour 2

bonjour

fin tour 2

début tour 3

bonjour

après la boucle

I.4.3. Les boucles

Les *boucles* permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

a) Boucle while

La syntaxe de while est la suivante :

while (*expression*)

instruction

Tant que *expression* est vérifiée (*i.e.*, non nulle), *instruction* est exécutée. Si *expression* est nulle au départ, *instruction* ne sera jamais exécutée. *instruction* peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
i = 1;
while (i < 10)
{
    printf("\n i = %d",i);
    i++;
}
```

b) Boucle do---while.

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle *do---while*. Sa syntaxe est :

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

Ici, *instruction* sera exécutée tant que *expression* est non nulle. Cela signifie donc que *instruction* est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;
do
{
        printf("\n Entrez un entier entre 1 et 10 : ");
        scanf("%d",&a);
}
while ((a <= 10) || (a > 0));
```

c) Boucle for

La syntaxe de for est :

```
for ( expr 1 ; expr 2 ; expr 3)
        instruction
```

Une version équivalente plus intuitive est :

```
expr 1;
while ( expr 2 )
{
        Instruction
        expr 3;
}
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)
        printf("\n i = %d",i);
```

A la fin de cette boucle, i vaudra 10. Les trois expressions utilisées dans une boucle for peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer le factorielle d'un entier, on peut écrire :

```
int n, i, fact;

for (i = 1, fact = 1; i <= n; i++)

        fact *= i;

        printf("%d ! = %d \n",n,fact);
```

On peut également insérer l'instruction fact *= i; dans la boucle for ce qui donne :

```
int n, i, fact;

for (i = 1, fact = 1; i <= n; fact *= i, i++);

printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

I.5. Les fonctions d'entrées-sorties classiques

Il s'agit des fonctions de la librairie standard stdio.h utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie stdio.h par la directive au préprocesseur *#include <stdio.h>* n'est pas nécessaire pour utiliser printf et scanf.

I.5.1 La fonction d'écriture printf

La fonction *printf* est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est

printf("*chaîne de contrôle* ", *expression-1*, ..., *expression-n*);

La *chaîne de contrôle* contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %, suivi d'un caractère désignant le format d'impression. Les formats d'impression en C sont donnés dans la table ci-dessous. En plus du caractère donnant le type des données, on peut éventuellement préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :

- ✓ largeur minimale du champ d'impression : %10d spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier. Par défaut, la donnée sera cadrée à droite du champ. Le signe - avant le format signifie que la donnée sera cadrée à gauche du champ (%-10d).
- ✓ précision :
    - %.12f signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule.
    - De même %10.2f signifie que l'on réserve 12 caractères (incluant le caractère .) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule.
    - Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule.
    - Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés :%30.4s signifie que l'on réserve un champ de 30 caractères pour

imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

Exemple :

```
#include <stdio.h>

main()

{

        int i = 23674;

        int j = -23674;

        long int k = (1l << 32);

        double x = 1e-8 + 1000;

        char c = ’A’;

        char *chaine = "chaine de caracteres";

        printf("impression de i: \n");

        printf("%d \t %u \t %o \t %x",i,i,i,i);

        printf("\nimpression de j: \n");

        printf("%d \t %u \t %o \t %x",j,j,j,j);

        printf("\nimpression de k: \n");

        printf("%d \t %o \t %x",k,k,k);
```

```
        printf("\n%ld \t %lu \t %lo \t %lx",k,k,k,k);
        printf("\nimpression de x: \n");
        printf("%f \t %e \t %g",x,x,x);
        printf("\n%.2f \t %.2e",x,x);
        printf("\n%.20f \t %.20e",x,x);
        printf("\nimpression de c: \n");
        printf("%c \t %d",c,c);
        printf("\nimpression de chaine: \n");
        printf("%s \t %.10s",chaine,chaine);
        printf("\n");
}
```

Ce programme imprime à l'écran :

impression de i:

23674 23674 56172 5c7a

impression de j:

-23674 4294943622 37777721606 ffffa386

impression de k:

0 0 0

4294967296 4294967296 40000000000 100000000

impression de x:

1000.000000 1.000000e+03 1000

1000.00 1.00e+03

1000.00000001000000000000 1.00000000001000000000e+03

impression de c:

A 65

impression de chaine:

chaine de caracteres chaine de

I.5.2 La fonction de saisie *scanf*

La fonction *scanf* permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonctions.

scanf("*chaîne de contrôle*", *argument-1,..., argument-n*)

La *chaîne de contrôle* indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de \n). Comme pour printf, les conversions de format sont spécifiées par un caractère précédé du signe %. Les formats valides pour la fonction scanf diffèrent légèrement de ceux de la fonction printf. Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple :

- ✓ %3s pour une chaîne de 3 caractères,
- ✓ %10d pour un entier qui s'étend sur 10 chiffres, signe inclus.

Exemple :

```
#include <stdio.h>

main()

{

        int i;

        printf("entrez un entier sous forme hexadecimale i = ");

        scanf("%x",&i);

        printf("i = %d\n",i);

}
```

Si on entre au clavier la valeur 1a, le programme affiche i = 26.

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

I.5.3 Impression et lecture de caractères

Les fonctions *getchar* et *putchar* permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

- ✓ La fonction getchar retourne un int correspondant au caractère lu. Pour mettre le caractère lu dans une variable caractere, on écrit

  caractere = getchar();

  Lorsqu'elle détecte la fin de fichier, elle retourne l'entier EOF (End Of File), valeur définie dans la librairie stdio.h. En général, la constante EOF vaut -1.

- ✓ La fonction putchar écrit *caractere* sur la sortie standard:

  putchar(caractere);

  Elle retourne un int correspondant à l'entier lu ou à la constante EOF en cas d'erreur.

Par exemple, le programme suivant lit un fichier et le recopie caractère par caractère à l'écran.

```
#include <stdio.h>

main()
```

```
{
        char c;
        while ((c = getchar()) != EOF)
                putchar(c);
}
```

Notons que l'expression (c = getchar()) dans le programme précédent a pour valeur la valeur de l'expression getchar() qui est de type int. Le test (c = getchar()) != EOF compare donc bien deux objets de type int (signés). Ce n'est par contre pas le cas dans le programme suivant :

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

Ici, le test c != EOF compare un objet de type char et la constante EOF qui vaut -1. Si le type char est non signé par défaut, cette condition est donc toujours vérifiée. Si le type char est signé, alors le caractère de code 255, ÿ, sera converti en l'entier -1. La rencontre du caractère ÿ sera donc interprétée comme une fin de fichier. Il est donc recommandé de déclarer de type int (et non char) une variable destinée à recevoir un caractère lu par getchar afin de permettre la détection de fin de fichier.

I.6. Les directives au préprocesseur.

Le préprocesseur est un programme exécuté lors de la premiére phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur, introduites par le caract`ere #, ont pour but:

- ✓ l'incorporation de fichiers source (#include),
- ✓ la définition de constantes symboliques et de macros (#define),
- ✓ la compilation conditionnelle (#if, #ifdef,...).

I.6.1. La directive *#include*.

Elle permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (stdio.h, math.h,...) ou n'importe quel autre fichier. La directive #include possède deux syntaxes voisines:

- ✓ ***#include <nom-de-fichier>*** recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, /usr/include/);
- ✓ ***#include "nom-de-fichier"*** recherche le fichier dans le répertoire courant (celui où se trouve le fichier source).

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

I.6.2. La directive *#define.*

La directive #define permet de définir:

- ✓ des constantes symboliques,
- ✓ des macros avec paramètres.

a) Définition de constantes symboliques.

La directive :

#define nom reste-de-la-ligne

demande au préprocesseur de substituer toute occurence de ***nom*** par la chaîne de caractères ***reste-de-la-ligne*** dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée. Par exemple :

```
#define NB_LIGNES 10

#define NB_COLONNES 33

#define TAILLE_MATRICE NB_LIGNES*NB_COLONNES
```

Il n'y a toutefois aucune contrainte sur la chaîne de caractères *reste-de-la-ligne*. On peut écrire :

```
#define BEGIN {

#define END }
```

b) Définition de macros

Une macro avec paramètres se définit de la manière suivante :

#define *nom*(*liste-de-paramètres*) *corps-de-la-macro*

où ***liste-de-paramètres*** est une liste d'identificateurs séparées par des virgules. Par exemple, avec la directive :

```
#define MAX(a,b) (a > b ? a : b)
```

le processeur remplacera dans la suite du code toutes les occurences du type MAX(x,y) où x et y sont des symboles quelconques par (x > y ? x : y).

Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution. La distinction entre une

définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthese ouvrante. Ainsi, si l'on écrit par erreur :

```
#define CARRE (a) a * a
```

la chaîne de caractères CARRE(2) sera remplacée par (a) a * a (2)

Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses:

```
#define CARRE(a) a * a
```

le préprocesseur remplacera CARRE(a + b) par a + b * a + b et non par (a + b) * (a + b) .

De même,

```
!CARRE(x)
```

sera remplacé par ! x * x et non par !(x * x).

Enfin, il faut être attentif aux éventuels effets de bord que peut entraîner l'usage de macros.

Par exemple, CARRE(x++) aura pour expansion (x++) * (x++). L'opérateur d'incrémentation sera donc appliqué deux fois au lieu d'une.

I.6.3 La compilation conditionnelle.

La *compilation conditionnelle* a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera générée par le préprocesseur. Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- ✓ la valeur d'une expression.
- ✓ l'existence ou l'inexistence de symboles.

5.3.1 Condition liée à la valeur d'une expression.

Sa syntaxe la plus générale est :

```
#if condition-1
        partie-du-programme-1
#elif condition-2
        partie-du-programme-2
        ...
        #elif condition-n
                partie-du-programme-n
        #else
                partie-du-programme-n+1
```

```
#endif
```

Le nombre de #elif est quelconque et le #else est facultatif. Chaque *condition-i* doit être une expression constante. Une seule *partie-du-programme* sera compilée : celle qui correspond à la première *condition-i* non nulle, ou bien la *partie-du-programme-n+1* si toutes les conditions sont nulles.

Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
#if PROCESSEUR == ALPHA
    taille_long = 64;
#elif PROCESSEUR == PC
    taille_long = 32;
#endif
```

5.3.2 Condition liée à l'existence d'un symbole

Sa syntaxe est :

```
#ifdef symbole
    partie-du-programme-1
#else condition-2
    partie-du-programme-2
#endif
```

Si *symbole* est défini au moment où l'on rencontre la directive #ifdef, alors *partie-duprogramme-1* sera compilée et *partie-du-programme-2* sera ignorée. Dans le cas contraire, c'est la *partie-du-programme-2* qui sera compilée. La directive #else est évidemment facultative.

De façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef symbole
    partie-du-programme-1
#else condition-2
    partie-du-programme-2
#endif
```

II. LES TYPES COMPOSES.

A partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types composés*, qui permettent de représenter des ensembles de données organisées.

II.1 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contigües. La déclaration d'un tableau à une dimension se fait de la façon suivante:

type nom-du-tableau[*nombre-éléments*];

où *nombre-éléments* est une expression constante entière positive. Par exemple, la déclaration

int liste_Note[10];

indique que liste_Note est un tableau de 10 éléments de type int. Cette déclaration alloue donc en mémoire pour l'objet liste_Note un espace de 10 × 4 octets consécutifs. Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre-éléments* par une directive au préprocesseur, par exemple :

#define *nombre-éléments* 10

On accède à un élément du tableau en lui appliquant l'opérateur []. Les éléments d'un tableau sont toujours numérotés de 0 à *nombre-éléments* -1. Le programme suivant imprime les éléments du tableau liste_Note :

```
#define N 10

main()

{

        int tab[N];

        int i;

        for (i = 0; i < N; i++)

                printf("tab[%d] = %d\n",i,tab[i]);

}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

Par exemple, on ne peut pas écrire "liste_Note1 = liste_Note2; ". Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10

main()

{
```

```
        int tab1[N], tab2[N];
        int i;
        for (i = 0; i < N; i++)
                tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

type nom-du-tableau[N] = *{constante-1, constante-2,..., constante-N}*;

Par exemple, on peut écrire :

```
#define N 4
int tab[N] = {1, 2, 3, 4};
main()
{
        int i;
        for (i = 0; i < N; i++)
        printf("tab[%d] = %d\n",i,tab[i]);
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation static.

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul '\0' . Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

```
#define N 8
char tab[N] = "exemple";
main()
{
        int i;
        for (i = 0; i < N; i++)
        printf("tab[%d] = %c\n",i,tab[i]);
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation. Ainsi le programme suivant imprime le nombre de caractères du tableau liste_Etudiant, ici 8.

```
char liste_Etudiant [] = "Camara";
main()
{
        int i;
        printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char));
}
```

De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

type nom-du-tableau[*nombre-lignes*][*nombre-colonnes*]

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "tableau[i][j] ". Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};
main()
{
        int i, j;
        for (i = 0 ; i < M; i++)
        {
                for (j = 0; j < N; j++)
                printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
        }
}
```

2.2 Les structures.

Une *structure* est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contigües en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est *modele* suit la syntaxe suivante :

```
struct modele
{
        type-1 membre-1;
        type-2 membre-2;
        ...
        type-n membre-n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

```
struct modele objet;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele
{
        type-1 membre-1;
        type-2 membre-2;
        ...
        type-n membre-n;
} objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur *membre de structure*, noté ".". Le i-ème membre de *objet* est désigné par l'expression

```
objet.membre-i
```

On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type *type-i*. Par exemple, le programme suivant définit la structure complexe, composée de deux champs de type double ; il calcule la norme d'un nombre complexe.

```
#include <math.h>
struct complexe
{
        double reelle;
        double imaginaire;
};
```

```
main()
{
        struct complexe z;
        double norme;
        ...
        norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
        printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe z = {2. , 2. };
```

En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
main()
{
        struct complexe z1, z2;
        ...
        z2 = z1;
}
```

2.3 Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (int ou unsigned int). Cela se fait en précisant le nombre de bits du champ avant le point virgule qui suit sa déclaration. Par exemple, la structure suivante :

```
struct registre
{
        unsigned int actif : 1;
        unsigned int valeur : 31;
};
```

possède deux membres, actif qui est codé sur un seul bit, et valeur qui est codé sur 31 bits. Tout objet de type struct registre est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation. Le champ actif de la structure ne peut prendre que les valeurs 0 et 1. Aussi, si r est un objet de type struct registre, l'opération

```
r.actif += 2;
```

ne modifie pas la valeur du champ. La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier. Notons enfin qu'un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur &.

2.4 Les unions

Une *union* désigne un ensemble de variables de types différents susceptibles d'occuper *alternativement* une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand. Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type struct.

Dans l'exemple suivant, la variable hier de type union jour peut être soit un entier, soit un caractère.

```
union jour
{
        char lettre;
        int numero;
};
main()
{
        union jour hier, demain;
        hier.lettre = 'J';
        printf("hier = %c\n",hier.lettre);
        hier.numero = 4;
        demain.numero = (hier.numero + 2) % 7;
        printf("demain = %d\n",demain.numero);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type unsigned int d'une structure en les identifiant à un objet de type unsigned long (en supposant que la taille d'un entier long est deux fois celle d'un int).

```
struct coordonnees
{
        unsigned int x;
        unsigned int y;
```

```
};
union point
{
        struct coordonnees coord;
        unsigned long mot;
};
main()
{
        union point p1, p2, p3;
        p1.coord.x = 0xf;
        p1.coord.y = 0x1;
        p2.coord.x = 0x8;
        p2.coord.y = 0x8;
        p3.mot = p1.mot ^ p2.mot;
        printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

2.5 Les énumérations.

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef enum et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet:

enum *modele {constante-1, constante-2,..., constante-n}*;

En réalité, les objets de type enum sont représentés comme des int. Les valeurs possibles *constante-1*, *constante-2*,..., *constante-n* sont codées par des entiers de 0 à *n-1*. Par exemple, le type enum booleen défini dans le programme suivant associe l'entier 0 à la valeur faux et l'entier 1 à la valeur vraie.

```
main()
{
        enum booleen {faux, vrai};
        enum booleen b;
        b = vrai;
        printf("b = %d\n",b);
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

enum booleen *{*faux = 12, vrai = 23*}*;

2.6 Définition de types composés avec *typedef*

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de typedef :

typedef *type synonyme*;

Par exemple :

```
struct complexe
{
        double reelle;
        double imaginaire;
};
typedef struct complexe complexe;
main()
{
        complexe z;
        ...
}
```

III. LES POINTEURS

Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*. Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

III.1 Adresse et valeur d'un objet

On appelle *Lvalue* (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :

- ✓ son adresse, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- ✓ sa valeur, c'est-à-dire ce qui est stocké à cette adresse.

Dans l'exemple,

```
int i, j;

i = 3;

j = i;
```

Si le compilateur a placé la variable i à l'adresse 4831836000 en mémoire, et la variable j à l'adresse 4831836004, on a :

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

Deux variables différentes ont des adresses différentes. L'affectation i = j; n'opère que sur les valeurs des variables. Les variables i et j étant de type int, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003. L'adresse d'un objet étant un numéro d'octet en mémoire, il s'agit d'un entier quelque soit le type de l'objet considéré. Le format interne de cet entier (16 bits, 32 bits ou 64 bits) dépend des architectures. Sur un DEC alpha, par exemple, une adresse a toujours le format d'un entier long (64 bits). L'opérateur & permet d'accéder à l'adresse d'une variable. Toutefois &i n'est pas une Lvalue mais une constante : on ne peut pas faire figurer &i à gauche d'un opérateur d'affectation. Pour pouvoir manipuler des adresses, on doit donc recourir à un nouveau type d'objets, les pointeurs.

III.2 Notion de pointeur

Un *pointeur* est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

type **nom-du-pointeur*;

où *type* est le type de l'objet pointé. Cette déclaration déclare un identificateur, *nom-du-pointeur*, associé à un objet dont la valeur est l'adresse d'un autre objet de type *type*.

L'identificateur *nom-du-pointeur* est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

Même si la valeur d'un pointeur est toujours un entier (éventuellement un entier long), le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet, pour un pointeur sur un objet de type char, la valeur donne l'adresse de l'octet où cet objet est stocké. Par contre, pour un pointeur sur un objet de type int, la valeur donne l'adresse du premier des 4 octets où l'objet est stocké. Dans l'exemple suivant, on définit un pointeur p qui pointe vers un entier i :

```
int i = 3;
int *p;
p = &i;
```

On se trouve dans la configuration.

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

L'*opérateur unaire d'indirection* * permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, *p désigne la valeur de i. Par exemple, le programme :

```
main()
{
    int i = 3;
    int *p;
    p = &i;
    printf("*p = %d \n",*p);
}
```

imprime *p = 3.

Dans ce programme, les objets i et *p sont identiques : ils ont mêmes adresse et valeur. Nous sommes dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Cela signifie en particulier que toute modification de *p modifie i. Ainsi, si l'on ajoute l'instruction *p = 0; à la fin du programme précédent, la valeur de i devient nulle. On peut donc

dans un programme manipuler à la fois les objets p et *p. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

```
main()
{
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

Et

```
main()
{
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    p1 = p2;
}
```

Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Après l'affectation *p1 = *p2; du premier programme, on a :

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Par contre, l'affectation p1 = p2 du second programme, conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

III.3. Arithmétique des pointeurs.

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- ✓ l'addition d'un entier à un pointeur. Le résultat est un pointeur de mùeme type que le pointeur de départ ;
- ✓ la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- ✓ la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée. Si i est un entier et p est un pointeur sur un objet de type *type*, l'expression p + i désigne un pointeur sur un objet de type *type* dont la valeur est égale à la valeur de p incrémentée de i*sizeof(*type*) . Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation ++ et --. Par exemple, le programme :

```
main()
{
        int i = 3;
        int *p1, *p2;
        p1 = &i;
        p2 = p1 + 1;
        printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche p1 = 4831835984 p2 = 4831835988.

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
main()
{
        double i = 3;
        double *p1, *p2;
```

```
        p1 = &i;
        p2 = p1 + 1;
        printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

affiche p1 = 4831835984 p2 = 4831835992.

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type. L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux. Ainsi, le programme suivant imprime les éléments du tableau *Note* dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int Note[5] = {1, 2, 6, 0, 7};
main()
{
        int *p;
        printf("\n ordre croissant:\n");
        for (p = &Note[0]; p <= & Note [N-1]; p++)
        printf(" %d \n",*p);
        printf("\n ordre decroissant:\n");
        for (p = & Note [N-1]; p >= & Note [0]; p--)
        printf(" %d \n",*p);
}
```

Si p et q sont deux pointeurs sur des objets de type *type*, l'expression p - q désigne un entier dont la valeur est égale à (p - q)/sizeof(*type*).

III.5. Pointeurs et tableaux.

L'usage des pointeurs en C est, en grande partie, orienté vers la manipulation des tableaux.

3.5.1 Pointeurs et tableaux à une dimension.

Tout tableau en C est en fait un pointeur constant. Dans la déclaration :

```
int tNote[10];
```

Note est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, Note a pour valeur & Note[0] . On peut donc utiliser un pointeur initialisé à Note pour parcourir les éléments du tableau.

```
#define N 5
```

```
int Note[5] = {1, 2, 6, 0, 7};
main()
{
        int i;
        int *p;
        p = Note;
        for (i = 0; i < N; i++)
        {
                printf(" %d \n",*p);
                p++;
        }
}
```

On accède à l'élément d'indice i du tableau Note grâce à l'opérateur d'indexation [], par l'expression Note[i]. Cet opérateur d'indexation peut en fait s'appliquer à tout objet p de type pointeur. Il est lié à l'opérateur d'indirection * par la formule :

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

pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire :

```
#define N 5
int Note[5] = {1, 2, 6, 0, 7};
main()
{
        int i;
        int *p;
        p = tab;
        for (i = 0; i < N; i++)
                printf(" %d \n", p[i]);
}
```

Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi :

- ✓ on ne peut pas créer de tableaux dont la taille est une variable du programme,
- ✓ on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à n éléments ou n est une variable du programme, on écrit :

```
#include <stdlib.h>

main()

{

        int n;

        int *tab;

        ...

        tab = (int*)malloc(n * sizeof(int));

        ...

        free(tab);

}
```

Si on veut en plus que tous les éléments du tableau tab soient initialisés à zéro, on remplace l'allocation dynamique avec malloc par :

```
tab = (int*)calloc(n, sizeof(int));
```

Les éléments de tab sont manipulés avec l'opérateur d'indexation [], exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont

- ✓ un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple p = &i ;
- ✓ un tableau n'est pas une Lvalue ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire tab++;).

III.5.2. Pointeurs et tableaux à plusieurs dimensions.

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

tab est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. tab a une valeur constante égale à l'adresse du premier élément du tableau, &tab[0][0].

De même tab[i], pour i entre 0 et M-1, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice i. tab[i] a donc une valeur constante qui est égale à &tab[i][0].

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés. On déclare un pointeur qui pointe sur un objet de type *type* *(deux dimensions) de la même manière qu'un pointeur, c'est-à-dire :

type ***nom-du-pointeur*;

De même un pointeur qui pointe sur un objet de type *type* ** (équivalent à un tableau à 3 dimensions) se déclare par :

type ****nom-du-pointeur*;

Par exemple, pour créer avec un pointeur de pointeur une matrice à k lignes et n colonnes à coefficients entiers, on écrit :

```
main()
{
        int k, n;
        int **tab;
        tab = (int**)malloc(k * sizeof(int*));
        for (i = 0; i < k; i++)
                tab[i] = (int*)malloc(n * sizeof(int));
        ....
        for (i = 0; i < k; i++)
        free(tab[i]);
        free(tab);
}
```

La première allocation dynamique réserve pour l'objet pointé par tab l'espace-mémoire correspondant à k pointeurs sur des entiers. Ces k pointeurs correspondent aux lignes de la matrice.

Les allocations dynamiques suivantes réservent pour chaque pointeur tab[i] l'espace-mémoire nécessaire pour stocker n entiers.

Si on désire en plus que tous les éléments du tableau soient initialisés à zéro, il suffit de remplacer l'allocation dynamique dans la boucle for par :

```
tab[i] = (int*)calloc(n, sizeof(int));
```

Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des lignes tab[i] . Par exemple, si l'on veut que tab[i] contienne exactement i+1 éléments, on écrit :

```
for (i = 0; i < k; i++)
        tab[i] = (int*)malloc((i + 1) * sizeof(int));
```

III.5.3. Pointeurs et chaînes de caractères.

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type char, se terminant par le caractère nul '\0'. On peut donc manipuler toute chaîne de

caractères à l'aide d'un pointeur sur un objet de type char. On peut faire subir à une chaîne définie par :

*char *chaine*;

des affectations comme :

chaine = "ceci est une chaine";

et toute opération valide sur les pointeurs, comme l'instruction chaine++;. Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>
main()
{
        int i;
        char *chaine;
        chaine = "chaine de caracteres";
        for (i = 0; *chaine != '\0'; i++)
                chaine++;
        printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard string.h, procède de manière identique. Il s'agit de la fonction *strlen* dont la syntaxe est :

strlen(*chaine*);

où *chaine* est un pointeur sur un objet de type char. Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère '\0'). L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
        int i;
        char *chaine1, *chaine2, *res, *p;
        chaine1 = "chaine ";
        chaine2 = "de caracteres";
```

```
        res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
        p = res;
        for (i = 0; i < strlen(chaine1); i++)
        *p++ = chaine1[i];
        for (i = 0; i < strlen(chaine2); i++)
                *p++ = chaine2[i];
        printf("%s\n",res);
}
```

On remarquera l'utilisation d'un pointeur intermédiaire p qui est indispensable dès que l'on fait des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de *res*, on aurait évidemment "perdu" la référence sur le premier caractère de la chaîne. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
        int i;
        char *chaine1, *chaine2, *res;
        chaine1 = "chaine ";
        chaine2 = "de caracteres";
        res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
        for (i = 0; i < strlen(chaine1); i++)
                *res++ = chaine1[i];
        for (i = 0; i < strlen(chaine2); i++)
                *res++ = chaine2[i];
        printf("\nnombre de caracteres de res = %d\n",strlen(res));
}
```

imprime la valeur 0, puisque res a été modifié au cours du programme et pointe maintenant sur le caractère nul.

IV. LES FONCTIONS.

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée *main*. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est *récursive*).

IV.1. Définition d'une fonction.

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme :

type nom-fonction (*type-1 arg-1,..., type-n arg-n*)

{

déclarations de variables locales

liste d'instructions

}

La première ligne de cette définition est l'*en-tˆete* de la fonction. Dans cet en-tête, *type* désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot clef *void*. Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramàtres, on remplace la liste de paramètres formels par le mot-clef *void*.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, return, dont la syntaxe est :

return(*expression*);

La valeur de *expression* est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type void), sa définition s'achève par return;

Plusieurs instructions return peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier return rencontré lors de l'exécution. Voici quelques exemples de définitions de fonctions :

```
int produit (int a, int b)
{
        return(a*b);
}
```

```
int puissance (int a, int n)
{
        if (n == 0)
                return(1);
        return(a * puissance(a, n-1));
}
void imprime_tab (int *tab, int nb_elements)
{
        int i;
        for (i = 0; i < nb_elements; i++)
                printf("%d \t",tab[i]);
        printf("\n");
        return;
}
```

IV.2. Appel d'une fonction.

L'appel d'une fonction se fait par l'expression :

nom-fonction(*para-1*, *para-2*,..., *para-n*)

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur *virgule*. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (++ ou --) dans les expressions définissant les paramètres effectifs.

IV.3. Déclaration d'une fonction.

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction main), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype*, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

type nom-fonction(*type-1*,..., *type-n*);

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction main. Par exemple, on écrira :

```
int puissance (int, int );
```

```
int puissance (int a, int n)
{
        if (n == 0)
                return(1);
        return(a * puissance(a, n-1));
}
main()
{
        int a = 2, b = 5;
        printf("%d\n", puissance(a,b));
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction main et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

Ainsi les fichiers d'extension.h de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier math.h le prototype de la fonction pow (élévation à la puissance) :

```
extern double pow(double , double );
```

La directive au préprocesseur :

```
#include <math.h>
```

permet au préprocesseur d'inclure la déclaration de la fonction pow dans le fichier source.

Ainsi, si cette fonction est appelée avec des paramètres de type int, ces paramètres seront convertis en double lors de la compilation.

Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction pow avec des paramètres de type int peut produire un résultat faux !

IV.4. Durée de vie des variables.

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manièê. En particulier, elles n'ont pas toutes la mˆeme *durée de vie*. On distingue deux catégories de variables.

1. **Les variables permanentes (ou statiques).**

Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef *static*.

2. **Les variables temporaires.**

Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire. Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, auto, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire.

On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type register. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt.

Grâce aux performances des optimiseurs de code intégrés au compilateur, il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

La durée de vie des variables est liée à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

IV.4.1. Variables globales.

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, n est une variable globale:

```
int n;

void fonction(void);

void fonction(void)

{

        n++;

        printf("appel numero %d\n",n);

        return;

}
```

```
main()
{
        int i;
        for (i = 0; i < 5; i++)
                fonction();
}
```

La variable n est initialisée à zéro par le compilateur et il s'agit d'une variable permanente.

En effet, le programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

IV.4.2. Variables locales.

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant :

```
int n = 10;
void fonction(void);
void fonction(void)
{
        int n = 0;
        n++;
        printf("appel numero %d\n",n);
        return;
}
main()
{
        int i;
```

```
        for (i = 0; i < 5; i++)
                fonction();
}
```

affiche

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef static :

static *type nom-de-variable*;

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation.

Par exemple, dans le programme suivant, n est une variable locale à la fonction secondaire *fonction*, mais de classe statique.

```
int n = 10;
void fonction();
void fonction()
{
        static int n;
        n++;
        printf("appel numero %d\n",n);
        return;
}
main()
{
        int i;
        for (i = 0; i < 5; i++)
                fonction();
```

```
}
```

Ce programme affiche :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale n est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

IV.5. Transmission des paramètres d'une fonction.

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*.

Par exemple, le programme suivant :

```
void echange (int, int );
void echange (int a, int b)
{
        int t;
        printf("debut fonction :\n a = %d \t b = %d\n",a,b);
        t = a;
        a = b;
        b = t;
        printf("fin fonction :\n a = %d \t b = %d\n",a,b);
        return;
}
main()
{
        int a = 2, b = 5;
```

```
        printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
        echange(a,b);
        printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

imprime :

```
debut programme principal :
a = 2 b = 5
debut fonction :
a = 2 b = 5
fin fonction :
a = 5 b = 2
fin programme principal :
a = 2 b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);
void echange (int *adr_a, int *adr_b)
{
        int t;
        t = *adr_a;
        *adr_a = *adr_b;
        *adr_b = t;
        return;
}
main()
{
        int a = 2, b = 5;
        printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
        echange(&a,&b);
        printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
```

```
}
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont modifiés par la fonction. Par exemple, le programme :

```
#include <stdlib.h>

void init (int *, int );

void init (int *tab, int n)

{

        int i;

        for (i = 0; i < n; i++)

                tab[i] = i;

        return;

}

main()

{

        int i, n = 5;

        int *tab;

        tab = (int*)malloc(n * sizeof(int));

        init(tab,n);

}
```

initialise les éléments du tableau tab.

IV.6. Les qualificateurs de type *const* et *volatile*.

Les qualificateurs de type *const* et *volatile* permettent de réduire les possibilités de modifier une variable.

- ✓ *const*. Une variable dont le type est qualifié par const ne peut pas être modifiée. Ce qualificateur est utilisé pour se protéger d'une erreur de programmation. On l'emploie principalement pour qualifier le type des paramètres d'une fonction afin d'éviter de les modifier involontairement.
- ✓ *volatile*. Une variable dont le type est qualifié par volatile ne peut pas être impliquée dans les optimisations effectuées par le compilateur. On utilise ce qualificateur pour les variables susceptibles d'être modifiées par une action extérieure au programme.

Les qualificateurs de type se placent juste avant le type de la variable, par exemple :

```
const char c;
```

d´esigne un caractère non modifiable. Ils doivent toutefois être utilisés avec précaution avec les pointeurs. En effet :

```
const char *p;
```

d´efinit un pointeur sur un caractère constant, tandis que :

```
char *const p;
```

définit un pointeur constant sur un caractère.

IV.7. La fonction main.

La fonction principale main est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type void, ce qui est toléré par le compilateur. Toutefois l'écriture main() provoque un message d'avertissement.

En fait, la fonction main est de type int. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur.

On peut utiliser comme valeur de retour les deux constantes symboliques EXIT SUCCESS (égale à 0) et EXIT FAILURE (égale à 1) définies dans stdlib.h.

L'instruction return(*statut*); dans la fonction main, où *statut* est un entier spécifiant le type de terminaison du programme, peut être remplacée par un appel à la fonction exit de la librairie standard (stdlib.h).

La fonction exit, de prototype :

```
void exit(int statut);
```

provoque une terminaison normale du programme en notifiant un succès ou un échec selon la valeur de l'entier statut.

Lorsqu'elle est utilisée sans arguments, la fonction main a donc pour prototype :

```
int main(void);
```

On s'attachera désormais dans les programmes à respecter ce prototype et à spécifier les valeurs de retour de main.

La fonction main peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes. En fait, la fonction main possède deux paramètres formels, appelés par convention :

```
argc(argument count) et argv (argument vector).
```

argc est une variable de type int dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1.

argv est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément argv[0] contient donc le nom de la commande (du fichier exécutable), le second argv[1] contient le premier paramètre....

Le second prototype valide de la fonction main est donc :

```
int main ( int argc, char *argv[]);
```

Ainsi, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int a, b;
    if (argc != 3)
    {
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage: %s int int\n",argv[0]);
        return(EXIT_FAILURE);
    }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
    return(EXIT_SUCCESS);
}
```

On lance donc l'exécutable avec deux paramètres entiers, par exemple :

```
a.out 12 8
```

Ici, argv sera un tableau de 3 chaînes de caractères argv[0] , argv[1] et argv[2] qui, dans notre exemple, valent respectivement "a.out", "12" et "8".

Enfin, la fonction de la librairie standard atoi() , déclarée dans stdlib.h, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

V. LA GESTION DES FICHIERS.

Le C offre la possibilité de lire et d'écrire des données dans un fichier. Pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire tampon (*buffer*), ce qui permet de réduire le nombre d'accès aux périphériques (disque...). Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations: l'adresse de l'endroit de la mémoire tampon où se trouve le fichier, la position de la tête de lecture, le mode d'accès au fichier (lecture ou écriture)...

Ces informations sont rassemblées dans une structure dont le type, FILE *, est défini dans stdio.h. Un objet de type FILE * est appelé *flot de données* (en anglais, stream).

Avant de lire ou d'écrire dans un fichier, on notifie son accès par la commande fopen. Cette fonction prend comme argument le nom du fichier, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction fclose.

6.1. Ouverture et fermeture d'un fichier.

6.1.1. La fonction fopen.

Cette fonction, de type FILE* ouvre un fichier et lui associe un flot de données. Sa syntaxe est:

fopen("*nom-de-fichier*","*mode*").

La valeur retournée par fopen est un flot de donn´ees. Si l'ex´ecution de cette fonction ne se d´eroule pas normalement, la valeur retourn´ee est le pointeur NULL. Il est donc recommand´e de toujours tester si la valeur renvoy´ee par la fonction fopen est ´egale `a NULL afin de d´etecter
les erreurs (lecture d'un fichier inexistant...).
Le premier argument de fopen est le nom du fichier concern´e, fourni sous forme d'une chaˆıne de caract`eres. On pr´ef´erera d´efinir le nom du fichier par une constante symbolique au moyen de la directive #define plutˆot que d'expliciter le nom de fichier dans le corps du programme.
Le second argument, *mode*, est une chaˆıne de caract`eres qui sp´ecifie le mode d'acc`es au fichier. Les sp´ecificateurs de mode d'acc`es diff`erent suivant le type de fichier consid´er´e. On

distingue
– les *fichiers textes*, pour lesquels les caract`eres de contrˆole (retour `a la ligne ...) seront interpr´et´es en tant que tels lors de la lecture et de l'´ecriture;
– les *fichiers binaires*, pour lesquels les caract`eres de contrˆole se sont pas interpr´et´es.
Les diff´erents modes d'acc`es sont les suivants:

"r"	ouverture d'un fichier texte en lecture
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin
"rb"	ouverture d'un fichier binaire en lecture
"wb"	ouverture d'un fichier binaire en écriture
"ab"	ouverture d'un fichier binaire en écriture à la fin
"r+"	ouverture d'un fichier texte en lecture/écriture
"w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin
"r+b"	ouverture d'un fichier binaire en lecture/écriture
"w+b"	ouverture d'un fichier binaire en lecture/écriture
"a+b"	ouverture d'un fichier binaire en lecture/écriture à la fin

Ces modes d'acc`es ont pour particularit´es:
– Si le mode contient la lettre r, le fichier doit exister.
– Si le mode contient la lettre w, le fichier peut ne pas exister. Dans ce cas, il sera cr´e´e. Si le fichier existe d´ej`a, son ancien contenu sera perdu.
– Si le mode contient la lettre a, le fichier peut ne pas exister. Dans ce cas, il sera cr´e´e. Si le fichier existe d´ej`a, les nouvelles donn´ees seront ajout´ees `a la fin du fichier pr´ec´edent.
Trois flots standard peuvent ˆetre utilis´es en C sans qu'il soit n´ecessaire de les ouvrir ou de les fermer:
– stdin (standard input): unit´e d'entr´ee (par d´efaut, le clavier);
– stdout (standard output): unit´e de sortie (par d´efaut, l'´ecran);
– stderr (standard error): unit´e d'affichage des messages d'erreur (par d´efaut, l'´ecran).
Il est fortement conseill´e d'afficher syst´ematiquement les messages d'erreur sur stderr afin que ces messages apparaissent `a l'´ecran mˆeme lorsque la sortie standard est redirig´ee.

6.1.2 La fonction fclose

Elle permet de fermer le flot qui a ´et´e associ´e `a un fichier par la fonction fopen. Sa syntaxe est:
fclose(*flot*)
o`u *flot* est le flot de type FILE* retourn´e par la fonction fopen correspondant.
La fonction fclose retourne un entier qui vaut z´ero si l'op´eration s'est d´eroul´ee normalement (et une valeur non nulle en cas d'erreur).

6.2 Les entr´ees-sorties format´ees

6.2.1 La fonction d'´ecriture fprintf

La fonction fprintf, analogue `a printf, permet d'´ecrire des donn´ees dans un fichier. Sa syntaxe est
fprintf(*flot*,"*cha^ıne de contr^ole*", *expression-1, ..., expression-n*)
o`u *flot* est le flot de donn´ees retourn´e par la fonction fopen. Les sp´ecifications de format utilis´ees pour la fonction fprintf sont les mˆemes que pour printf (*cf.* page 30).

6.2.2 La fonction de saisie fscanf

La fonction fscanf, analogue `a scanf, permet de lire des donn´ees dans un fichier. Sa

syntaxe est semblable à celle de scanf : fscanf(*flot*,"*cha^ıne de contr^ole*", *argument-1,..., argument-n*) o`u *flot* est le flot de donn´ees retourn´e par fopen. Les sp´ecifications de format sont ici les mˆemes que celles de la fonction scanf (*cf.* page 32).

6.3 Impression et lecture de caract`eres

Similaires aux fonctions getchar et putchar, les fonctions fgetc et fputc permettent respectivement de lire et d'´ecrire un caract`ere dans un fichier. La fonction fgetc, de type int, retourne le caract`ere lu dans le fichier. Elle retourne la constante EOF lorsqu'elle d´etecte la fin du fichier. Son prototype est

int fgetc(FILE* *flot*);

o`u *flot* est le flot de type FILE* retourn´e par la fonction fopen. Comme pour la fonction getchar, il est conseill´e de d´eclarer de type int la variable destin´ee `a recevoir la valeur de retour de fgetc pour pouvoir d´etecter correctement la fin de fichier (*cf.* page 33).

La fonction fputc ´ecrit *caractere* dans le flot de donn´ees :

int fputc(int *caractere*, FILE **flot*)

Elle retourne l'entier correspondant au caract`ere lu (ou la constante EOF en cas d'erreur).

Il existe ´egalement deux versions optimis´ees des fonctions fgetc et fputc qui sont impl´ement´ees par des macros. Il s'agit respectivement de getc et putc. Leur syntaxe est similaire

`a celle de fgetc et fputc :

int getc(FILE* *flot*);

int putc(int *caractere*, FILE **flot*)

Ainsi, le programme suivant lit le contenu du fichier texte entree, et le recopie caract`ere par caract`ere dans le fichier sortie :

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

#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"
int main(void)
{
FILE *f_in, *f_out;
int c;
if ((f_in = fopen(ENTREE,"r")) == NULL)
{
fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
return(EXIT_FAILURE);
}
if ((f_out = fopen(SORTIE,"w")) == NULL)
{
fprintf(stderr, "\nErreur: Impossible d'ecrire dans le fichier %s\n", \
SORTIE);
return(EXIT_FAILURE);
}
while ((c = fgetc(f_in)) != EOF)
fputc(c, f_out);
fclose(f_in);
```

```
fclose(f_out);
return(EXIT_SUCCESS);
}
```

6.4 Relecture d'un caract`ere

Il est p ossible de replacer un caract`ere dans un flot au moyen de la fonction ungetc :

int ungetc(int *caractere*, FILE **flot*);

Cette fonction place le caract`ere *caractere* (converti en unsigned char) dans le flot *flot*. En particulier, si *caractere* est ´egal au dernier caract`ere lu dans le flot, elle annule le d´eplacement provoqu´e par la lecture pr´ec´edente. Toutefois, ungetc peut ˆetre utilis´ee avec n'importe quel caract`ere (sauf EOF). Par exemple, l'ex´ecution du programme suivant

```
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
int main(void)
{
FILE *f_in;
int c;
if ((f_in = fopen(ENTREE,"r")) == NULL)
{
fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
return(EXIT_FAILURE);
}
while ((c = fgetc(f_in)) != EOF)
{
if (c == '0')
ungetc('.',f_in);
putchar(c);
}
fclose(f_in);
return(EXIT_SUCCESS);
}
```

sur le fichier entree.txt dont le contenu est 097023 affiche `a l'´ecran 0.970.23

6.5 Les entr´ees-sorties binaires

Les fonctions d'entr´ees-sorties binaires permettent de transf´erer des donn´ees dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entr´ee-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des donn´ees d´epend des machines.

Elles sont notamment utiles pour manipuler des donn´ees de grande taille ou ayant un type compos´e. Leurs prototypes sont :

size t fread(void **pointeur*, size t *taille*, size t *nombre*, FILE **flot*);
size t fwrite(void **pointeur*, size t *taille*, size t *nombre*, FILE **flot*);

o`u *pointeur* est l'adresse du d´ebut des donn´ees `a transf´erer, *taille* la taille des objets `a transf´erer, *nombre* leur nombre. Rappelons que le type size t, d´efini dans stddef.h, correspond au type du r´esultat de l'´evaluation de sizeof. Il s'agit du plus grand type entier non sign´e.

La fonction fread lit les donn´ees sur le flot *flot* et la fonction fwrite les ´ecrit. Elles

retournent toutes deux le nombre de donn´ees transf´er´ees. Par exemple, le programme suivant ´ecrit un tableau d'entiers (contenant les 50 premiers entiers) avec fwrite dans le fichier sortie, puis lit ce fichier avec fread et imprime les ´el´ements du tableau.

```
#include <stdio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie"
int main(void)
{
FILE *f_in, *f_out;
int *tab1, *tab2;
int i;

tab1 = (int*)malloc(NB * sizeof(int));
tab2 = (int*)malloc(NB * sizeof(int));
for (i = 0 ; i < NB; i++)
tab1[i] = i;
/* ecriture du tableau dans F_SORTIE */
if ((f_out = fopen(F_SORTIE, "w")) == NULL)
{
fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE);
return(EXIT_FAILURE);
}
fwrite(tab1, NB * sizeof(int), 1, f_out);
fclose(f_out);
/* lecture dans F_SORTIE */
if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE);
return(EXIT_FAILURE);
}
fread(tab2, NB * sizeof(int), 1, f_in);
fclose(f_in);
for (i = 0 ; i < NB; i++)
printf("%d\t",tab2[i]);
printf("\n");
return(EXIT_SUCCESS);
}
```

Les ´el´ements du tableau sont bien affich´es `a l'´ecran. Par contre, on constate que le contenu du fichier sortie n'est pas encod´e.

6.6 Positionnement dans un fichier

Les diff´erentes fonctions d'entr´ees-sorties permettent d'acc´eder `a un fichier en *mode s´equentiel* : les donn´ees du fichier sont lues ou ´ecrites les unes `a la suite des autres. Il est ´egalement possible d'acc´eder `a un fichier en *mode direct*, c'est-`a-dire que l'on peut se positionner `a n'importe quel endroit du fichier. La fonction fseek permet de se positionner `a un

endroit pr´ecis ; elle a pour prototype :
int fseek(FILE *flot, long deplacement, int origine);
La variable *deplacement* d´etermine la nouvelle position dans le fichier. Il s'agit d'un d´eplacement relatif par rapport `a l'origine ; il est compt´e en nombre d'octets. La variable *origine*
peut prendre trois valeurs :
– SEEK SET (´egale `a 0) : d´ebut du fichier ;
– SEEK CUR (´egale `a 1) : position courante ;
– SEEK END (´egale `a 2) : fin du fichier.
La fonction
int rewind(FILE *flot);
permet de se positionner au d´ebut du fichier. Elle est ´equivalente `a
fseek(flot, 0, SEEK SET);
La fonction
long ftell(FILE *flot);
retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).
Par exemple

```
#include <stdio.h>
#include <stdlib.h>
#define NB 50
#define F_SORTIE "sortie"
int main(void)
{
FILE *f_in, *f_out;
int *tab;
int i;
tab = (int*)malloc(NB * sizeof(int));
for (i = 0 ; i < NB; i++)
tab[i] = i;
/* ecriture du tableau dans F_SORTIE */
if ((f_out = fopen(F_SORTIE, "w")) == NULL)
{
fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",F_SORTIE);
return(EXIT_FAILURE);
}
fwrite(tab, NB * sizeof(int), 1, f_out);
fclose(f_out);
/* lecture dans F_SORTIE */
if ((f_in = fopen(F_SORTIE, "r")) == NULL)
{
fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",F_SORTIE);
return(EXIT_FAILURE);
}
/* on se positionne a la fin du fichier */
fseek(f_in, 0, SEEK_END);
```

```
printf("\n position %ld", ftell(f_in));
/* deplacement de 10 int en arriere */
fseek(f_in, -10 * sizeof(int), SEEK_END);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d", i);
/* retour au debut du fichier */
rewind(f_in);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d", i);
/* deplacement de 5 int en avant */
fseek(f_in, 5 * sizeof(int), SEEK_CUR);
printf("\n position %ld", ftell(f_in));
fread(&i, sizeof(i), 1, f_in);
printf("\t i = %d\n", i);
fclose(f_in);
return(EXIT_SUCCESS);
}
```

L'ex´ecution de ce programme affiche `a l'´ecran :

```
position 200
position 160 i = 40
position 0 i = 0
position 24 i = 6
```

On constate en particulier que l'emploi de la fonction fread provoque un d´eplacement correspondant `a la taille de l'objet lu `a partir de la position courante.