Licence SFA - 2ème année

Techniques Scientifiques

pour l'Ingénieur

Compléments sur le langage C

- r 1.3b -

P. Castelan

I. Structures

I.a. Qu'est-ce que c'est ?

Il est possible de regrouper des variables simples ou dimensionnées sous un même nom en utilisant un type appelé structure. Une structure contient plusieurs champs qui sont les variables regroupées. Les structures permettent de manipuler aisément des données complexes.

I.b. La syntaxe

Une structure est un type de variable créé par le programmeur à l'aide de `typedef`. Une fois le type créé on peut créer des variables.

On défini les types structures à l'aide du mot `struct`. Exemple :

```
typedef struct
     {    type nom_champ_1;
          …;
          type nom_champ_n;

     } Nom_du_type_créé;
```

```
typedef struct
     {    char genre;
          float valeur;

     } composant;
```

Le code ci-dessus crée un type de structure appelé composant. Les structures de ce type comporteront 2 champs : `genre` et `valeur` avec les types respectivement déclarés.

On peut alors créer une variable de type composant :

```
int main(void)
{    composant compo1;
     ...
```

Ce qui crée une variable `compo1` de type composant.

On accède aux différents champs de la variable `compo1` en *postfixant* la variable du nom du champ avec un point.

```
compo1.genre = 'R';
compo1.valeur = 1e4; /* compo1 est une résistance de 10 kOhms */
```

L'avantage des structures est qu'elles permettent le stockage et la transmission d'information directement par paquet organisés. En général, on utilise des structures pour modéliser des données complexes tel que des fiches comportant plusieurs champs . De plus il s'agit de variables comme n'importe quelle variable, ainsi l'affectation entre deux structures est possible.

```
 int main(void)
{    composant compo1, compo2;
     …

     compo1 = compo2 ; // tout à fait valide !!
```

Par contre on ne peut multiplier ou additionner des structures entr'elles.

I.c. Résumé

Il faut créer un type avant d'utiliser des structures.

Une structure comporte des champs auxquels on accède via l'opérateur «.».

II. Rappel sur les fonctions

II.a. Qu'est ce que c'est ?

Une fonction est un programme indépendant. Comme tout programme elle reçoit des paramètres et retourne un résultat.

II.b. La syntaxe

```
type  nom_de_la_fonction ( liste de paramètres )
{     liste de variables locales ;

      liste d'instructions ;

      return valeur_retournée ;
}
```

Le type de la fonction est soit un type simple : `int`, `float`, `char`, soit un type de structure créé par le programmeur soit `void`. Une fonction de type `void` ne retourne rien. On parle, alors, souvent de procédure.

Le nom de la fonction doit être clair et concis, il ne doit comporter aucun caractère accentué, et ne pas débuter par un chiffre.

La liste de variables peut être aussi longue que l'on veut et le format est celui de la déclaration des variables : `type param1, type param2` ...

La liste des variables locales a la même forme que dans la fonction `main` (qui est une fonction !)

Nota Bene : Il ne faut pas utiliser le même nom pour un paramètre et une variable locale.

➡ **Les variables locales n'ont d'existence qu'à l'intérieur de la fonction.**

Les paramètres, comme leur nom l'indiquent, sont des *consignes* transmises à la fonction. **Ce sont des variables locales initialisées lors de l'appel à la fonction par la fonction appelante.**

La valeur retournée par le `return` doit être du type de la fonction.

Les fonctions de type `void` n'ont pas de `return`, vu qu'il n'y a rien à retourner (`void` = vide)

II.c. Utilisation

Pour utiliser une fonction il suffit de l'appeler par son nom. La fonction qui appelle (ou fonction appelante) doit fournir autant de valeurs qu'il y a de paramètres à la fonction appelée. Ces valeurs initialisent les paramètres de la fonction appelée. La première valeur est transmise au premier paramètre et ainsi de suite. Il faut que le type de la valeur transmise soit le même que celui du paramètre qui lui correspond.

Exemple :

```
float puissance ( float x, int n )
{     int i;

      for (i = 1 ; i <= n ; i++ )
            x *= x ;// ou x = x*x ;
      return x;
}
```

```
int main(void)
{     float Y,a = 4.23 ;

      Y = puissance ( a , 9 );
...
```

Les valeurs peuvent être le contenu de variables ou des valeurs constantes.

II.d. "Etanchéité"

Les fonctions sont "étanches".

Les variables locales et les paramètres ne sont connus que **dans** la fonction. Toute modification de ces variables ou paramètres reste **interne** à la fonction. Ce n'est pas parce que l'on modifie la valeur d'un paramètre que la valeur transmise lors de l'appel de la fonction sera modifiée ! Par exemple, ci-dessus, on ne peut évidement pas modifier 9 qui est transmis à `n`, mais la modification de la valeur de `x` ne modifie pas `a` non plus ! Cela ne veut pas dire que dans la fonction puissance x et n ne sont pas modifiables, mais que les éventuelles modifications ne modifieront pas les valeurs transmises.

II.e. Les prototypes

Les prototypes servent à déclarer les fonctions. Le C est un langage déclaratif. On doit déclarer tout ce que l'on utilise, y compris les fonctions. Notez bien que les compilateurs acceptent souvent l'oubli des prototypes. **Les correcteurs non !**

Un prototype est constitué de la première ligne –ou en-tête– d'une fonction suivit d'un point virgule.

Il est des cas où la déclaration des fonctions est indispensable. Considérons un programme contenant les fonctions `F1` et `F2` suivantes :

```
int F1( int a, int b)
{     ...
      b = F2( 2*a, -3*b) ;
      ...
}
int F2( int a, int b)
{     ...
      a = F1( -2*a, 3*b) ;
      ...
}
```

Ce programme ne peut se compiler. `F2` n'est pas connue à la compilation de `F1` ce qui provoque une erreur. Permuter `F1` et `F2` ne sert à rien. Alors que déclarer `F1` et `F2` via les prototypes résout la difficulté.

```
int F1( int a, int b) ;
int F2( int a, int b) ;
```

Le fait de déclarer les prototypes fait que les fonctions sont connues avant leur compilation. Par conséquent le compilateur peut tout à fait compiler le programme, lorsqu'il rencontre l'appel à la fonction `F2` dans la fonction `F1`, il sait quels sont ses paramètres et ce qu'elle retourne. Il peut donc «laisser la place nécessaire» pour que l'appel à la fonction `F2` soit possible.

➡ **pour que les prototypes soient utiles, il faut impérativement les placer en un seul groupe en tout début de programme, avant les fonctions.**

III. Rappels sur les variables

III.a. Qu'est ce que c'est ?

Les variables sont des noms que l'on donne à **une zone de la mémoire** de l'ordinateur. Ce n'est qu'une commodité pour le programmeur. L'ordinateur se «débrouille» très bien sans les noms de variable, il utilise les adresses des variables pour accéder à leur contenu.

Il est possible de typer ces zones mémoire pour interpréter le contenu de la zone réservée. Le typage de la mémoire est aussi une commodité pour le programmeur. La mémoire de l'ordinateur ne contient que des zéros et des uns, c'est le programmeur qui précise comment interpréter ces nombres binaires via le type des variables.

III.b. Utilisation des variables

Comme toujours en langage C, on commence par déclarer les variables. La déclaration des variables s'effectue en précisant le type de la variable suivit de son nom.

Il est possible d'affecter des valeurs aux variables –y compris à la déclaration–, d'appliquer des opérations au contenu des variables, de comparer des variables entre elles à l'aide des opérateurs adéquats : référez au cours de M. Mary pour ces opérations.

Il est possible de **changer localement le type** d'une variable à l'aide de l'opérateur de *transtypage* : il suffit de faire précéder le nom de la variable du type local souhaité entre parenthèses. Exemple :

```
float a = 3.17; // declare une variable a de type float et l'initialise à 3,17
printf("A en entier = %d \n", (int)a ); // on transtype a de float vers int
```

Dans l'exemple ci-dessus, a est convertie localement (dans le `printf`) en une variable de type `int` pour correspondre au champ `%d` du `printf`.

III.c. Variables et adresse mémoire

Il est possible d'obtenir l'adresse où l'ordinateur stocke les données à l'aide l'opérateur `&` : exemple

```
scanf( "%d" , &a );
```

Il est possible d'obtenir le **contenu d'une adresse** à l'aide de l'opérateur `*` qui est l'opérateur de déréférencement. Il faut, bien entendu, préciser comment interpréter la suite de zéros et de uns contenus à l'adresse considérée à l'aide du transtypage. Ce qui donne le code suivant :

```
(int *)Adresse_memoire;
/* accède à une adresse et interprète son contenu comme un int. */
```

Toutes les variables n'ont pas la même taille, c'est à dire qu'elles n'occupent pas le même nombre d'octets en mémoire. Cette taille dépend évidement de ce que contient la variable donc de son type. La taille est comptée en octets (*bytes in english*).

Il est possible de connaître de nombre d'octets qu'occupe une variable à l'aide de l'opérateur `sizeof( ).`

```
int a;
printf("un entier occupe %d octets \n",sizeof(a)); // ou sizeof( int )
```

III.d. Rappel : Variables dimensionnées

III.d.a. Syntaxe

Les variables dimensionnées sont déclarées en précisant entre crochets droits le nombre d'éléments souhaités :

```
type nom[nombre d'éléments] ;
```

Ainsi, si l'on veut déclarer une variable de 10 entiers nommée a :

```
int a[10];
```

La numérotation débutant à zéro en C, les éléments sont numérotés de 0 à 9.

On accède au contenu de l'élément k de la variable a à l'aide de crochets droits : `a[k]`

III.d.b. Initialisation à la déclaration

On précise entre accolades les valeurs devant initialiser les éléments de la variable dimensionnée. La première valeur allant dans le premier élément *–celui d'indice 0–* et ainsi de suite. Le fait de ne préciser qu'une partie des éléments de la variable impose l'initialisation à zéro des éléments dont la valeur n'a pas été spécifiée.

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // a[9] vaut 10
int a[10]={0}; // met tout le vecteur à 0, initialiser le 1er élément à 0 suffit !
```

III.d.c. Utilisation de typedef

On peut créer de nouveaux types de variable à l'aide de l'instruction `typedef`. Par exemple :

```
typedef float expe[20];
```

Il est alors possible de créer des variables de type `expe` :

```
expe a={1,2,3}; // crée une variable appelée a de type expe.
```

a est une variable de type `expe` et est initialisée à la déclaration : `a[1]` vaut 2.

III.d.d. Variables à dimensions multiples

Il est possible de créer des variables a plusieurs dimensions, en rajoutant autant de jeux de crochets droits que de dimensions. Par exemple pour une variable a deux dimensions :

```
int a[3][4];
```

➡ **Nota Bene :** `a[1]` **est une variable de 4 éléments entiers numérotés de 0 à 3.**

Il est possible d'initialiser certains éléments de `a[1]` à la déclaration en utilisant des jeux d'accolades imbriqués.

```
int a[3][4] = { {} , {1,2,3 } };
```

Pour initialiser des variables à deux dimensions, il faut deux jeux d'accoaldes. Le premier jeu (le plus extérieur) concerne la première dimension -le premier jeu de crochets droits-, les accolades internes la seconde dimension -le second jeu de crochets droits-. Dans l'exemple ci-dessus, il y aura donc 1 jeu d'accolades externes, 3 jeux d'accolades internes et la possibilité de spécifier pour chacun 4 valeurs. Il suffit qu'une valeur soit spécifiée, pour que toutes les valeurs non spécifiées soient mises à zéro. La variable a est initialisée ainsi :

	a[*][0]	a[*][1]	a[*][2]	a[*][3]
a[0]	0	0	0	0
a[1]	1	2	3	0
a[2]	0	0	0	0

On peut aussi utiliser des déclarations de type pour générer des variables à plusieurs dimensions. Une matrice est un vecteur de vecteur. Ainsi,

```
typedef float vecteur[10];
typedef vecteur matrice[20];
matrice a = { {} , {1,2,3,4} }; permet de définir le type matrice. L'avantage est que a[1]
est un vecteur : a contient 20 éléments de type vecteur qui sont eux même des variables di-
mensionnées de 10 float.
```

Cette écriture, qui est plus claire que la première, est à préférer. Elle offre un contrôle plus aisé du flux des données du programme.

IV. Les pointeurs

IV.a. Qu'est-ce que c'est ?

C'est une **variable** qui contient une adresse. Comme à l'adresse en question il peut y avoir soit un `int`, soit un `float` ou tout autre type de variable, les pointeurs sont typés. C'est à dire qu'il y a des pointeurs pour les `int`, les `float` etc.

Bien entendu, comme un pointeur est une variable, son contenu peut changer, et le pointeur peut successivement contenir les adresses de plusieurs variables différentes, mais toutes du même type. Lorsqu'un pointeur contient le contenu d'une variable, on dit qu'*il pointe vers* elle.

On peut déclarer un pointeur de type `void`, c'est à dire sans type particulier. Il faudra alors utiliser systématiquement le transtypage pour utiliser ce pointeur.

IV.b. Utilisation

On déclare une variable comme étant un pointeur en faisant **précéder** son nom d'une étoile.

```
int a; // un entier
int *p; // un pointeur vers un entier, le type pointeur vers entier est int *
```

Contrairement aux autres variables, un pointeur est initialisé lors de sa déclaration. La valeur par défaut que contient un pointeur est **NULL** –en majuscules–, c'est à dire qu'il ne contient aucune adresse –en fait l'adresse 0–.

On stocke dans un pointeur des adresses, par exemple :

```
p = &a
```

On dit que `a` est la *variable pointée* par `p` et l'on peut alors utiliser le contenu de `p`, qui est une adresse directement :

```
scanf("%d",p) // équivaut à scanf("%d",&a) puisque p contient l'adresse de a.
```

On peut accéder au contenu de la variable pointée en utilisant l'opérateur de déréférencement[1] qu'est l'astérisque '`*`'. L'opérateur de déréférencement se place avant le pointeur, ainsi `*p` est la variable pointée, soit ici `a`.

```
printf ("%d" , *p );// équivaut à printf("%d", a );
```

En résumé :

Variable a		Pointeur p = &a (a est la variable pointée)	
Usage	**Sens**	**Usage**	**sens**
`a`	`contenu de la variable : une valeur`	`p`	`contenu du pointeur : une adresse`
`&a`	`adresse de la variable`	`*p`	`contenu de la variable pointée`

Nota Bene :

Il n'est pas nécessaire de transtyper pour obtenir le contenu d'une adresse mémoire avec un pointeur typé, par défaut l'adresse est interprétée comme étant du type du pointeur. Les pointeurs de type `void *` permettent d'accéder à n'importe quel type de variable. Ils sont non typés, puisque de type `void *`, ils ne pointent donc pas vers des variables (il n'y a pas de variables de type `void` !) mais étant non typés ils nécessitent un transtypage systématique.

IV.c. Pointeurs et variables dimensionnées

➡ **Une variable dimensionnée est un pointeur fixe.**

Ainsi, si l'on déclare :

```
int b[10];
```

Le compilateur réalise les opérations suivantes :

1/ réserve l'espace mémoire pour 10 entiers : 10*sizeof(int).

2/ crée un pointeur b et l'initialise avec l'adresse du premier entier de la zone de mémoire réservée précédemment. b pointe donc vers le premier entier possible, soit b pointe vers b[0] ou encore b=&b[0].

[1] *cf . le paragraphe IIIc*

Afin que le bloc mémoire réservé ne soit pas perdu, il n'est pas possible de modifier le contenu du pointeur `b`.

Une variable dimensionnée est une adresse. Il est possible de gérer cette adresse avec la même syntaxe qu'un pointeur. Toutefois, cette similitude a une limite, il n'est pas possible de modifier l'adresse en question. En conséquence, si, dans une fonction on peut retourner une adresse (mettre une adresse dans un `return` est bien possible !) on ne peut affecter cette adresse a une variable dimensionnée, puisque le lien `b=&b[0]`, ne peut être rompu. De même l'affectation directe entre deux variables dimensionnées n'est pas possible (`a = b` signifiant en fait `&a[0] = &b[0]` ce qui est évidement aussi impossible à réaliser que d'écrire 23 = 42 !).

C'est pourquoi on dit qu'une variable dimensionnée est un pointeur fixe. Elle a tous les attributs du pointeur, mais ne peut peut être modifiée, le lien avec l'adresse du premier élément ne pouvant être rompu.

IV.d. Arithmétique pointeur

Il est possible d'ajouter des valeurs **entières** à un pointeur. Ainsi

```
int *p;
…
p = p + 2 ;
```

est tout à fait valide.

En fait cette opération revient à augmenter l'adresse contenue dans le pointeur `p` de deux fois la taille du type pointé, ici un `int`. Ainsi, si `p` contient l'adresse `$300`, `p+2` pointera vers l'adresse `$300 + 2*sizeof(int)` puisque `p` est de type `int *`, si un entier est codé sur 4 octets, `p+2` contiendra alors l'adresse `$300 + 2*4 = $308`.

➡ **Le C ne vérifie pas que la case mémoire ainsi pointée existe -il n'en n'a pas les moyens !-**

Cette faculté peut être mise à profit pour accéder aux éléments d'une variable dimensionnée ainsi que le montre l'exemple ci-dessous.

```
1     int a[10]
2     int *p;
3     p = a ;
4     *(p + 2 ) = 3
5     printf("%d",a[2] );
6     printf("%d",p[2] ); // C'est possible !
7     printf("%d",*(p+2) );
```

Ligne 3, on fait pointer `p` vers `a`. En effet, `a` est une variable dimensionnée, donc `a = &a[0]` et par conséquent `p` contient désormais `&a[0]`.

Ligne 4, on calcule l'adresse correspondant à l'adresse de base décalée de deux fois la taille d'un entier. On accède à cette adresse via l'opérateur de déréférencement, et on y stocke l'entier 3. Or `p = &a[0]` et en décalant cette adresse de deux, on obtient `&a[0+2] = &a[2] = (p+2)`, donc `*(p+2) = a[2]`.

Ligne 6, il est bien sur possible d'utiliser avec un pointeur la même syntaxe qu'avec une variable dimensionnée pour effectuer les décalages d'adresse. Donc `p[2]` est tout à fait valide.

IV.e. Utilisation de pointeurs dans une fonction

Il est possible d'utiliser des pointeurs en paramètres de fonction. Ce faisant il devient alors possible de transmettre à une fonction une adresse. Il est bien entendu possible de placer des données à une adresse particulière en vue d'effectuer un transfert d'information entre deux parties de programme, voire entre deux programmes différents.

Le programme suivant illustre cette utilisation des pointeurs.

```
01    #include <stdio.h>
02    float ma_fonction(float *a, int n);// prototype
03
04    float ma_fonction(float *a, int n)
05    {     float local;
06          local = *a * n;
07          *a = 56576;
08          n = 123;
09          printf("a et n dans ma_fonction : %f\t%d\n",*a,n);
10          return local;
11    }
12    int main( void)
13    {     int b=2;
14          float x=2.718;
15          printf("%f\n",ma_fonction(&x,b));
16          printf("x et b dans main() : %f\t%d\n",x,b);
17          return 0;
18    }
```

Le programme débute ligne 12 avec la fonction main. Ligne 13 et 14, les deux variables `b` et `x` ont été créées et initialisées. Ligne 15 la fonction est appelée, **et l'on transmet `x` par adresse et `b` par valeur**.

Pour transmettre une variable par adresse, il faut que la fonction ait un de ses paramètres qui soit un pointeur du bon type. Ligne 5, `a` contient l'adresse de `x`, et `n` la valeur de `b`. Donc `a = &x` et `n = 2`. En conséquence, ligne 7 la variable pointée par `a`, soit `x` reçoit le nombre 56576, alors que ligne 8 la variable `n` reçoit 123, ce qui ne modifie en rien le contenu de `b`.

Ligne 16 on affiche `x` et `b`, `x` a une valeur modifiée, `b` non.

Règles de programmation :

- **Lorsque que l'on veut qu'une fonction puisse modifier le contenu d'une variable appartenant à une autre fonction, il faut transmettre cette variable par adresse et non par valeur.**
- **Les variables dimensionnées étant des adresses, elles sont forcément transmises par adresse, leur contenu est donc forcément mis à jour dans la fonction appelante s'il est modifié dans la fonction appelée.**

Ce second point est illustré ci-dessous.

```
01   #include <stdio.h>
02   void saisie(int M[10], int *n);
03   void saisie(int M[10], int *n)
04   {  int i;
05      printf("Entrez la dimension : ");
06      scanf("%d",n);
07      for ( i=0 ; i<*n ; i++)
08      {  printf("A[%d] = ",i);
09         scanf("%d",&M[i]);
10      }
11   }
```

```
12 int main( void)
13 {  int i,N;
14    int a[10]={0};
15    saisie(a,&N);
16    printf("\nVous avez saisi : \n");
17    for ( i=0 ; i<N ; i++)
18       printf("a(%1d) : %d\n",i,a[i]);
19    return 0;
20 }
```

- **La fonction saisie a deux paramètres qui sont tous deux des pointeurs. L'un est explicite, `n`, l'autre implicite, `M`.**

Nota Bene :

Il est bien sur possible d'écrire la fonction saisie sous la forme :

```
void saisie(int *M , int *n)
```

Il reste évidement alors possible d'écrire :

```
scanf("%d",&M[i]);
```

ou

```
scanf("%d",M+i); // M+i est l'adresse de l'élément i d'une variable dimensionnée
```

IV.f. Structures et pointeurs

Il est possible de définir des pointeurs sur des structures, mais il y a une difficulté pour accéder aux champs : l'opérateur `.` a priorité sur l'opérateur `*`. Ainsi pour accéder au `champ1` d'une structure pointée par un pointeur `p` on pourrait imaginer écrire : `*p.champ1`. Ceci n'est pas possible à cause des règles de priorité : avec cette écriture, le compilateur va interpréter d'abord `p.champ1`, puis `*(p.champ1)`. Or le pointeur `p` n'a pas de `champ1`, c'est la variable pointée qui possède le `champ1`. Pour régler ce problème on peut utiliser des parenthèses pour fixer l'ordre des priorité ainsi : `(*p).champ1`. Cette fois-ci le compilateur effectue d'abord `*p`, donc accède à la variable pointée, puis au `champ1` de cette variable, ce qui est correct.

Il existe une autre possibilité pour accéder aux champs d'une variable pointée via son pointeur c'est d'utiliser l'opérateur `->` (signe moins suivi du symbole supérieur) et l'on écrira : `p->champ1` pour accéder au `champ1` de la variable pointée. Ceci est illustré ci-dessous :

```
typedef struct
      {     char type;
            float valeur;

      } composant;
composant c1, *c2;
c1.type = 'R';
c2 = &c1;               // c2 pointe vers c1
*c2.type ='L' ;   // impossible, c2 n'a pas de champ type !
(*c2).type = 'L' ;      // ok, la variable pointée a bien un champ type
c2->type = 'L' ;      // syntaxe simplifiée : moins supérieur
```

V. Fichiers texte

V.a. Généralités

Les supports de stockage sont multiples : CD, DVD, réseau !...

L'accès aux périphériques impose de piloter la quincaille (le hardware) avec un logiciel spécifique à chaque support le driver (ou pilote). Le processus est complexe... mais pas pour nous ! C'est l'OS (windows, linux, MacOS ou autre...) qui fait ce travail pour nous.

Pour lire un fichier il faut créer un lien entre votre programme et l'OS. Cette zone d'échange est appelée *tampon* ou *buffer.*

➡ **Il est possible de réaliser des accès directs aux fichiers, cette année nous ne ferons que des accès «bufferisés».**

V.b. Comment faire ?

La méthode que vous utiliserez comporte 3 étapes :

1/ Création du tampon servant de lien entre votre programme et le support de stockage : **ouverture**.

2/ Accès aux données en lecture ou en écriture voire en lecture et écriture.

3/ Fermeture du fichier.

Pour chacune des étapes décrites ci-dessus le C met à disposition des outils. Ces outils utilisent des pointeurs de type `FILE *` définis dans la bibliothèque `stdio.h`

V.c. Ouverture des fichiers

L'ouverture se fait à l'aide de `fopen ()`

La syntaxe est la suivante :

```
FILE *mavariable;
mavariable = fopen(chemin d'accès, mode d'ouverture du fichier);
```

➡ **Le chemin d'accès absolu ou relatif utilise la syntaxe UNIX exclusivement.**

Même sous windows le chemin à passer à la fonction `fopen()` utilise des '/' et non des '\'. Par exemple on écrira : "c:/travail/toto.txt" et non "c:\travail\toto.txt".

Il existe plusieurs modes d'ouvertures de fichier résumés dans le tableau ci dessous :

mode	ouverture en :
"r"	lecture seule
"w"	écriture seule
"a"	ajout
"r+"	lecture/écriture
"w+"	écriture/lecture

Ainsi si on ne doit que lire le contenu du fichier toto.txt se trouvant dans le dossier travail on procédera comme suit :

```
FILE *mavariable;
mavariable = fopen("/travail/toto.txt", "r" );
```

L'ouverture en mode écriture "w" provoque la création d'un fichier portant le nom `"toto.txt"` si celui n'existe pas. **Si ce fichier existe, il est d'abord effacé.**

Le mode `r+` diffère du mode `w+` en ce sens qu'il ouvre le fichier en lecture s'il existe, alors que le mode `w+` **effacera** d'abord le fichier existant avant de l'ouvrir.

En cas d'erreur la fonction `fopen()` retourne la valeur `NULL`.

V.d. Fichiers Texte

Les fichiers avec lesquels nous travaillerons sont des fichiers de **type texte**. Les données sont stockées sous forme de caractères et sont séparées entre elles par un **séparateur de champ.**

De plus les données sont groupées en lignes et chaque ligne est séparée de la suivante par des **séparateurs de ligne.**

Si les séparateurs de champ peuvent être quelconques, les séparateurs de ligne dépendent de l'OS.

Système d'exploitation	Séparateur de ligne
Windows	CR + LF
MS DOS	CR
Linux / UNIX / MAC OS	LF

Le caractère CR (carriage return) correspond au code ascii 13 et le LF (line feed) au code 10. Ces codes (comme tout caractère de code inférieur à 32) sont invisibles à l'écran. Il sont explicités, ici, pour plus de commodité.

Par exemple :

```
3.1415;2.717LF
1.414;1.73LF
```

Ce fichier a un séparateur de champ qui est le ';' et un séparateur de ligne de type UNIX. On peut voir que chaque ligne contient le même nombre de données (ici 2) séparées par le délimiteur de champ. Les données 3.1415 et 1.414 sont dans la même colonne.

V.e. Lecture

La lecture des données s'effectue avec la fonction `fscanf()`. La syntaxe est la suivante :

```
fscanf(variable_FILE,"format", liste d'adresse de variables);
```

La partie format et la liste d'adresse ont la même syntaxe que pour la fonction `scanf()`.

La seule différence réside dans le fait qu'il ne faut pas oublier de séparer les champs par le séparateur dans la chaîne de format.

Le premier paramètre de la fonction `fscanf` est un pointeur de type `FILE`. Ce pointeur doit avoir été initialisé auparavant à l'aide de la fonction `fopen()`.

La fonction `fscanf()` lit le fichier ligne par ligne et les données séparées par le séparateur sont attribuées aux différentes variables dont les adresses ont été fournies. Le `champ1` à la première variable etc.

V.f. Fin de fichier

La fonction `feof()` indique si la fin du fichier a été atteinte. La syntaxe est la suivante :

```
feof(variable_FILE);
```

Si la fin du fichier est atteinte, la fonction retourne vrai.

Le code suivant réalise la lecture d'un fichier comportant deux colonnes de `float` séparées par un ';'.

```
FILE *f1;
f1 = fopen("toto.txt","r");
while ( !feof(f1)) // tant qu'on  n'est pas à la fin !
{     fscanf(f1,"%f;%f",&a,&b); // a et b sont des float, pas de \n dans fscanf...
}
```

On peut tester `f1` après le `fopen` en la comparant à `NULL`. Si `f1` vaut `NULL` alors le fichier n'est pas ouvert et il faut arrêter le programme avec la fonction `exit` par exemple (`exit` est dans `stdlib.h`) :

```
if ( f1 == NULL)
{     printf("Erreur de fichier\n");
      exit(1) ;
}
```

V.g. Ecriture

Lorsqu'on écrit un fichier la question du nombre de lignes ne se pose pas. Une boucle `for` est donc suffisante.

La fonction que vous utiliserez pour écrire les données est `fprintf`. Sa syntaxe est la suivante :

```
fprintf(variable_FILE,"chaîne de format", liste de variables );
```

La seule différence entre `fprintf` et `printf` est *variable_FILE*.

Enfin il ne faut pas oublier le séparateur de champs dans la chaîne de format, ce peut être un simple espace -c'est le plus courant-, une tabulation '\t' ou tout autre caractère qui ne participe pas au stockage de l'information (ce peut être 9 si vous écrivez en base 8...)

V.h. Fermeture des fichiers

Une fois les fichiers lus ou écrits, il faut "fermer" le fichier. Cela consiste à informer le système d'exploitation que l'on a fini avec le fichier. Informer le système d'exploitation a deux buts :

1/ vider le tampon sur le support de masse

2/ libérer la mémoire utilisée par le tampon.

Nota Bene :

Tant qu'un fichier n'est pas fermé, il est ouvert ! Il est donc susceptible de recevoir des données et le système d'exploitation ne procède à l'écriture physique des données qu'une fois le tampon plein ou qu'on lui signifie la fermeture du fichier. Si vous ne fermez pas un fichier vous avez de fortes chances pour qu'il soit incomplet.

La fonction permettant de fermer un fichier est `fclose()`, sa syntaxe est :

```
fclose(variable_FILE);
```

VI. Les chaînes de caractères

VI.a. Qu'est-ce que c'est ?

Une chaîne de caractère est une variable dimensionnée de type char.

En conséquence il n'est pas possible d'affecter une chaîne à une autre chaîne : une variable dimensionnée est un pointeur fixe !

VI.b. Les deux types de chaînes

Une chaine de caractères contient un nombre limité de caractères. Il faut donc un système pour informer le système du nombre de caractères contenus dans la chaine. Il y a deux approches :

Pascal	6	'C'	'O'	'U'	'C'	'O'	'U'						
C	'C'	'O'	'U'	'C'	'O'	'U'	'\0'						

La chaîne pascal occupe toujours 256 octets, la longueur d'une chaîne C est variable et n'est pas limitée.

Les deux types de chaînes nécessitent l'utilisation d'un caractère pour préciser la longueur de la chaîne.

En Pascal : le premier caractère est utilisé pour indiquer la longueur de celle-ci. Ce qui limite la longueur de la chaîne à 255 caractères. Un caractère est codé sur un octet et il n'y a que 256 valeurs possible avec 8 bits.

En C : le denier caractère est le caractère NULL (code ASCII 0 -zéro- noté \0).

Nous utiliserons des chaînes de type C. Dans certains cas, il n'y a pas d'autre solution que d'utiliser des chaines pascal, mais ceci est hors programme.

VI.c. Chaînes et pointeurs

Le programme suivant permet de comprendre comment utiliser les chaînes de caractères.

```
1 #include <stdio.h>
2
```

```
 3 int main( void)
 4 {  char S[]="bonjour"; // taille non obligatoire
 5
 6   int i;
 7
 8    printf("%s\n",S); // noter le printf %s !
 9    for (i=0 ; i<sizeof(S) ; i++) // noter le sizeof
10          printf("%c : %3d\n",S[i],S[i]);
11
12    S = "titi"; // ne marche pas !! pourquoi ?
13
14    return 0;
15 }
```

Ligne 4 on déclare une variable dimensionnée de type char que l'on initialise.

On peut afficher la chaîne d'un bloc à l'aide du format "`%s`" dans le `printf` ligne 8.

Notez l'utilisation de `sizeof` ligne 9. Dans le cas d'une chaine de caractères, comme chaque caractère occupe un octet, `sizeof` retourne le nombre d'octets donc de caractères.

On peut aussi afficher la chaîne caractère par caractère ligne 10 et afficher chaque case sous forme de caractère ou d'entier : c'est le programmeur qui décide du sens de ce que contient la mémoire…

Si `S` est déclarée :

```
char *S="bonjour";
```

l'affectation ligne 12 devient valide puisque `S` est un pointeur, son contenu (la variable pointée) peut changer. Toutefois, l'adresse de la précédente chaine ("bonjour") est perdue par conséquent la mémoire réservée cette chaîne est "perdue" : la mémoire reste réservée, mais n'est plus accessible puisqu'on ne sait plus où sont les informations.

De plus il n'est plus possible d'utiliser `sizeof` pour déterminer la longueur de la chaine puisque `S` est un pointeur, `sizeof` renverra la taille d'une adresse mémoire et non celle de la chaine...

VI.d. manipulation de chaînes de caractères

Le C met à disposition tout un ensemble de fonctions pour manipuler les chaînes de caractères dans la bibliothèque `string.h`.

Le programme suivant permet d'en présenter quelques-unes.

```
 1 #include<stdio.h>
 2 #include<string.h>
 3 int main( void)
 4 {  char *S="bonjour ";
 5    char *T="Toto";
 6    char big[500];
 7    char *L;
 8
 9    L = strchr(S,'j');      // cherche le caractère j dans S, retourne son adresse
10    printf("%X\t%X\n",S,L); // %X en héxadécimal
11    printf("%s \n",L);
12
13    L = big;
14    strcpy(L,S); // copie S à l'adresse L
15    printf("%s\n",L);
16    strcat(L,T); // Rajoute T à la suite de la chaîne débutant à l'adresse L
17    printf("%s\n",L);
18
19    return 0;
20 }
```

Ce programme retourne :

```
2FB0    2FB3
jour

bonjour
bonjour Toto
```

Il y a beaucoup d'autres fonctions dans la bibliothèque `strings.h` ou `string.h`

VII. Gestion dynamique de la mémoire

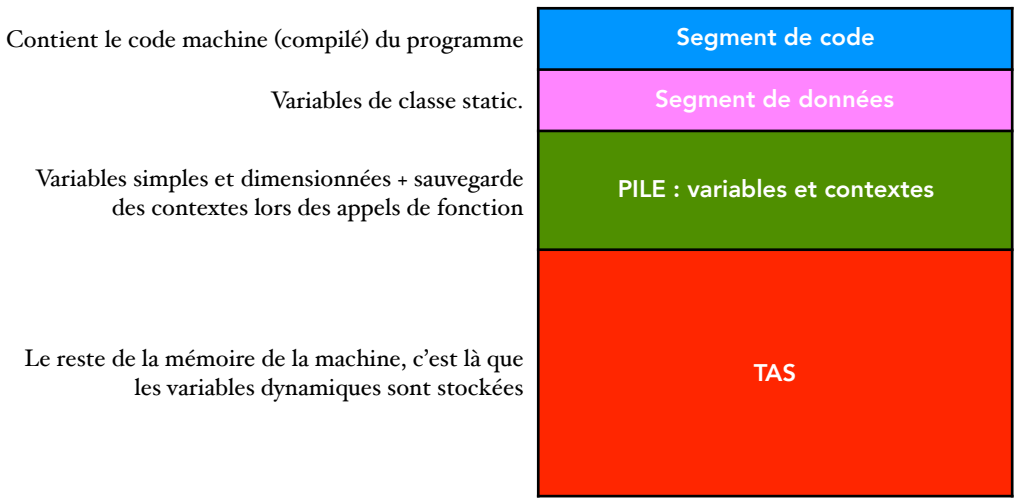
VII.a. Organisation de la mémoire d'un programme

Cette question dépend du mode de compilation de votre programme. Il existe en effet plusieurs modèles en fonction de la taille de l'application que l'on va écrire.

Nous ignorerons ces directives, nous n'aurons pas cette année des programmes de taille telle qu'il faille choisir un modèle différent du modèle par défaut du compilateur gcc.

La figure ci-dessous résume l'organisation de la mémoire d'un programme C.

Organisation de la mémoire d'un programme



Le **segment de code** contient le programme proprement dit. Il s'agit de la suite des instructions pour le microprocesseur.

Le **segment de données** n'est en fait quasiment jamais utilisé pour nos programmes.

La **zone de pile**, contient les environnements, les variables etc. Il faut comprendre que lorsqu'on appelle une fonction, on place les informations nécessaires au bon déroulement du programme pour son retour dans la fonction appelante.

Le **tas** c'est le reste de la mémoire, c'est là que nous créerons les variables dynamiques. Le tas est «commun» à tous les programmes.Tous les programmes accédant au tas, c'est l'OS qui arbitre entre les différents programmes et l'accès au tas (Heap) se fait via des fonction normalisées.

VII.b. Variables dynamiques

L'accès à la mémoire dynamique se fait via des fonctions que le C met à la disposition dans la bibliothèque `stdlib.h`.

Le processus pour réserver de la mémoire est le même que pour les fichiers :

1/ On réserve de la mémoire,

2/ on l'utilise,

3/ on la libère.

VII.c. Réserver de la mémoire : malloc( )

La syntaxe est :

```
pointeur = malloc(nombre d'octets demandés);
```

`malloc` retourne une adresse non typée donc de type `void *`.

Pour stocker des `int`, il faut transtyper l'adresse retournée par la fonction. Par exemple si l'on veut une zone mémoire pour stocker 200 `int` il faut que `malloc` retourne une adresse de type `int *`, ce qui donne le code suivant :

```
int *p;
p = (int *)malloc(200*sizeof(int));
```

S'il n'y a plus de mémoire libre la fonction `malloc` retourne la valeur `NULL`.

Si la réservation a abouti, il est alors possible d'utiliser la mémoire via le pointeur comme une variable dimensionnée par exemple.

VII.d. Libérer la mémoire : free( )

La libération de la mémoire se fait avec la fonction `free`. La syntaxe est la suivante :

```
free(adresse du bloc à libérer) ;
```

Si vous ne libérez pas la mémoire, le fait de quitter le programme libère en principe la mémoire. Mais si le programme plante, il est possible qu'il reste des zones de mémoire que vous n'avez pas libéré et qui bloquent d'autres process, un peu comme dans le cas des chaines de caractères, on perd de la mémoire : le bloc reste réservé, mais comme son adresse n'est pas connue, il devient inaccessible.

Le programme suivant permet de créer un tableau de `n` entiers et l'utilise.

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3 int main(void)
 4 {     int *p,n,k;
 5
 6    printf("Entrez la taille :");
 7    scanf("%d",&n);
 8    p = (int *)malloc(n*sizeof(int));
 9    if (p == NULL)
10    {     printf("Erreur");
11          exit(1);
12    }
13    for ( k=0 ; k<n ; k++ )
14          p[k] = k*k;
15    for ( k=0 ; k<n ; k++ )
16          printf("%d\n", p[k]);
17    free(p);
18    return 0;
19 }
```