Tableaux et Fonctions

Soit un entier **N** strictement positif.
Supposons qu'on veuille saisir **N** nombres réels afin de calculer leur moyenne, ou de trouver leur minimum, ou de les afficher par ordre croissant.
Pour **N** assez petit, on peut déclarer **N** variables réelles pour résoudre le problème suscité. Mais si **N** est assez grand, on se rend compte que cela devient impropre, fastidieux, voire impossible.
Il faudra donc, dans ce cas, utiliser une variable permettant de représenter et de lister les n nombres. Ainsi, on crée un nouveau type de données appelé le **type tableau**.

Définition *:* **Un tableau permet de regrouper une quantité importante d'objets de même type. Ainsi, on appelle tableau une collection séquentielle d'éléments de même type, chacun étant repérée par sa position dans la collection. Cette position est appelée un indice ou bien le rang de l'élément dans le tableau.**

Les chaînes de caractères sont déclarées en C comme tableaux de caractères et permettent l'utilisation d'un certain nombre de notations et de fonctions spéciales.

Les tableaux peuvent être de dimension 1 - on parle alors aussi de vecteurs - de dimension 2, on parle de matrice ou plus.

I. Tableaux à une dimension : vecteur

Un tableau (unidimensionnel) est une variable structurée formée d'un nombre entier N de variables du même type, qui sont appelées les *composantes* du tableau. Le nombre de composantes N est alors la *dimension* du tableau.

En faisant le rapprochement avec les mathématiques, on dit encore que *"A est un vecteur de dimension N".*

1. Déclaration

Pour déclarer un tableau à une dimension, il faut donner:

- son ***nom*** ou identificateur de la variable de type tableau, par exemple TAB,
- sa ***dimension*** : nombre de composantes;
- et le ***type de données*** des éléments qui composent ce tableau.

<TypeElements> <NomTableau>[<Dimension>];
exemple:

int C[10];

Cette déclaration réserve l'emplacement pour 10 éléments de type int. Chaque élément est repéré par sa position dans le tableau, nommée indice. Conventionnellement, en langage C, la première position porte le numéro 0. Ici, donc, nos indices vont de 0 à 9. Le premier élément du tableau sera désigné par t[0], le troisième par t[2], le dernier par t[9].

char D[30];

Mémorisation
Si un tableau est formé de N composantes et si une composante a besoin de M octets en mémoire,
alors le tableau occupera de N*M octets.

2. Initialisation et réservation automatique

2.1 – Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades.

Exemple

```
int A[5]   = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int C[10]  = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

Il faut évidemment veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro.

2.2 – Réservation automatique

Si la dimension n'est pas indiquée explicitement lors de l'initialisation, alors l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

Exemples

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

==> réservation de 5*sizeof(int) octets (dans notre cas: 10 octets)

float B[] = {-1.05, 3.33, 87e-5, -12.3E4};

==> réservation de 4*sizeof(float) octets (dans notre cas: 16 octets)

int C[] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};

==> réservation de 10*sizeof(int) octets (dans notre cas: 20 octets)

3. Accès aux composantes

L'élément N°i sera identifié par :

élément N° 4 du tableau C : ***C[4]***

NB : Considérons un tableau T de dimension N; en langage C :

- l'accès au premier élément du tableau se fait par **T[0]**
- l'accès au dernier élément du tableau se fait par **T[N-1]**

4. Opérations de saisie et d'affichage

```
main()
{
    int A[5];
    int I;  /* Compteur */
    for (I=0; I<5; I++)
    scanf("%d", &A[I]);
    return 0;
}
```

4.2 – Affichage d'un tableau A de cinq éléments

```
main()
{
    int A[5];
    int I;  /* Compteur */
    for (I=0; I<5; I++)
    printf("%d ", A[I]);
    return 0;
    printf("\n");
}
```

Remarque : Avant de pouvoir afficher les composantes d'un tableau, il faut évidemment leur affecter des valeurs.

II. Tableau à deux dimensions : matrice

En C, un tableau à deux dimensions A est à interpréter comme un tableau (uni-dimensionnel) de dimension L dont chaque composante est un tableau (uni-dimensionnel) de dimension C.

On appelle L le *nombre de lignes* du tableau et C le *nombre de colonnes* du tableau. L et C sont alors les deux *dimensions* du tableau. Un tableau à deux dimensions contient donc *L*C composantes*.

On dit qu'un tableau à deux dimensions est ***carré***, si L est égal à C.

En faisant le rapprochement avec les mathématiques, on peut dire que *"A est un vecteur de L vecteurs de dimension C"*, ou mieux:

*"A est une **matrice** de dimensions L et C"*.

1. Déclaration

Pour déclarer un tableau à deux dimensions, il faut donner:

- son ***nom*** ou identificateur de la variable de type tableau, par exemple TAB,
- sa ***dimension de lignes*** : nombre de lignes;
- sa ***dimension de colonnes*** : nombre de colonnes;
- et le ***type de données*** des éléments qui composent ce tableau.

<Type_elements> <NomTabl>[<DimLigne>][<DimCol>];

exemple:

int t[5][3]

Cette déclaration réserve un tableau de 15 (5 x 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

t[3][2] t[i][j] t[i-3][i+j]

Mémorisation

Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de ***l'adresse du premier élément*** du tableau (c.-à-d. l'adresse de la première ***ligne*** du tableau). Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

Un tableau de dimensions L et C, formé de composantes dont chacune a besoin de M octets, occupera L*C*M octets en mémoire.

2. Initialisation et réservation automatique

2.1 – Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades. A l'intérieur de la liste, les composantes de chaque ligne du tableau sont encore une fois comprises entre accolades. Pour améliorer la lisibilité des programmes, on peut indiquer les composantes dans plusieurs lignes.

Exemples

```
int A[3][10] ={{ 0,10,20,30,40,50,60,70,80,90},
{10,11,12,13,14,15,16,17,18,19},
{1,12,23,34,45,56,67,78,89,90}};

float B[3][2] = {{-1.05,   -1.10  },
{86e-5,    87e-5 },
{-12.5E4, -12.3E4}};
```

Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite. Nous ne devons pas nécessairement indiquer toutes les valeurs. Les valeurs manquantes seront initialisées par zéro. Il est cependant défendu d'indiquer trop de valeurs pour un tableau.

2.2 – Réservation automatique

Si le nombre de **lignes L** n'est pas indiqué explicitement lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

```
int A[][10] = {{ 0,10,20,30,40,50,60,70,80,90},
{10,11,12,13,14,15,16,17,18,19},
{ 1,12,23,34,45,56,67,78,89,90}};
```

réservation de 3*10*2 = 60 octets

```
float B[][2] = {{-1.05,   -1.10  },
{86e-5,    87e-5 },
{-12.5E4, -12.3E4}};
```

réservation de 3*2*4 = 24 octets

3. Accès aux composantes

Pour accéder à un élément d'une matrice, on le fait via son numéro de ligne et son numéro de colonne.

L'élément N°i,j sera identifié par :

élément de la ligne 2 et de la colonne 4 du tableau C : ***C[2][4]***

Attention !

Considérons un tableau A de dimensions **L** et **C**. En langage C :

- les indices du tableau varient de **0** à **L-1**, respectivement de **0** à **C-1**.
- la composante de la Nième ligne et Mième colonne est notée: ***A[N-1][M-1]***

4. Opérations de saisie et d'affichage

```
main()
{
    int A[5][10];
    int I,J;
    /* Pour chaque ligne ... */
    for (I=0; I<5; I++)
    /* ... considérer chaque composante */
       for (J=0; J<10; J++)
       scanf("%d", &A[I][J]);
    return 0;
}
```

4.2 – Affichage d'un tableau A de cinq ligne et dix colonnes

```
main()
{
   int A[5][10];
   int I,J;
   /* Pour chaque ligne ... */
   for (I=0; I<5; I++)
```

```
    {
        /* ... considérer chaque composante */
        for (J=0; J<10; J++)
        printf("%7d", A[I][J]);
        /* Retour à la ligne */
        printf("\n");
    }
    return 0;
}
```

III– Chaîne de caractères

caractères). Il existe quand même des notations particulières et une bonne quantité de fonctions spéciales pour le traitement de tableaux de caractères.

1. Déclaration

char <NomVariable> [<Longueur>];

Exemples

```
char NOM [20];
char PRENOM [20];
char PHRASE [300];
```

Espace à réserver

Lors de la déclaration, nous devons indiquer l'espace à réserver en mémoire pour le stockage de la chaîne.

La représentation interne d'une chaîne de caractères est terminée par le symbole **'\0'** (NUL). Ainsi, pour un texte de **n** caractères, nous devons prévoir **n+1** octets.

Malheureusement, le compilateur C ne contrôle pas si nous avons réservé un octet pour le symbole de fin de chaîne; l'erreur se fera seulement remarquer lors de l'exécution du programme.

Mémorisation

Le nom d'une chaîne est le représentant de ***l'adresse du premier caractère*** de la chaîne. Pour mémoriser une variable qui doit être capable de contenir un texte de N caractères, nous avons besoin de N+1 octets en mémoire.

2. Chaîne de caractères constante

- Les chaînes de caractères constantes *(string literals)* sont indiquées entre guillemets. La chaîne de caractères vide est alors: **""**
- Dans les chaînes de caractères, nous pouvons utiliser toutes les séquences d'échappement définies comme caractères constants:

"Ce \ntexte \nsera réparti sur 3 lignes."

- Le symbole **"** peut être représenté à l'intérieur d'une chaîne par la séquence d'échappement **\"**:

"Affichage de \"guillemets\" \n"

- Le symbole **'** peut être représenté à l'intérieur d'une liste de caractères par la séquence d'échappement **\'** :

{'L','\'','a','s','t','u','c','e','\0'}

- Plusieurs chaînes de caractères constantes qui sont séparées par des signes d'espacement (espaces, tabulateurs ou interlignes) dans le texte du programme seront réunies en une seule chaîne constante lors de la compilation:

"un " "deux"
" trois"

sera évalué à

"un deux trois"

Ainsi il est possible de définir de très longues chaînes de caractères constantes en utilisant plusieurs lignes dans le texte du programme.

Observation

Pour la mémorisation de la chaîne de caractères "Hello", le langage C a besoin de **six (!!)** octets.

'x'	est un *caractère constant*, qui a une valeur numérique P.ex: 'x' a la valeur 120 dans le code ASCII.
"x"	est un *tableau de caractères* qui contient deux caractères la lettre 'x' et le caractère NUL: '**\0**'
'x'	est codé dans un octet
"x"	est codé dans deux octets

3. Initialisation de chaine de caractères

En général, les tableaux sont initialisés par l'indication de la liste des éléments du tableau entre accolades:

char CHAINE[] = {'H','e','l','l','o','\0'};

Pour le cas spécial des tableaux de caractères, nous pouvons utiliser une initialisation plus confortable en indiquant simplement une chaîne de caractère constante:

char CHAINE[] = "Hello";

Lors de l'initialisation par **[]**, l'ordinateur réserve automatiquement le nombre d'octets nécessaires pour la chaîne, c.-à-d.: le nombre de caractères **+ 1** (ici: 6 octets). Nous pouvons aussi indiquer explicitement le nombre d'octets à réserver, si celui-ci est supérieur ou égal à la longueur de la chaîne d'initialisation.

Exemples:

4. Accès aux éléments d'une chaîne de caractères

L'accès à un élément d'une chaîne de caractères peut se faire de la même façon que l'accès à un élément d'un tableau. En déclarant une chaîne par:

char A[6];

nous avons défini un tableau A avec six éléments, auxquels on peut accéder par:

A[0], A[1], ... , A[5]

A. 5. Précédence alphabétique et lexicographique

5.1 Précédence alphabétique des caractères

La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé. Pour le code ASCII, nous pouvons constater l'ordre suivant:

. . . ,0,1,2, ... ,9, . . . ,A,B,C, ... ,Z, . . . ,a,b,c, ... ,z, . . .

Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] , ...) et les lettres accentuées (é ,è ,à ,û , ...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules). Leur précédence ne correspond à aucune règle d'ordre spécifique.

Relation de précédence

De la précédence alphabétique des caractères, on peut déduire une ***relation de précédence*** *'est inférieur à'* sur l'ensemble des caractères. Ainsi, on peut dire que

'0' *est inférieur à* **'Z'**

et noter

'0' < 'Z'

car dans l'alphabet de la machine, le code du caractère '0' (ASCII: 48) est inférieur au code du caractère 'Z' (ASCII: 90).

5.2 - Précédence lexicographique des chaînes de caractères

En nous basant sur cette relation de *précédence alphabétique des caractères*, nous pouvons définir une ***précédence lexicographique*** *pour les chaînes de caractères*. Cette relation de précédence suit l'<<ordre du dictionnaire>> et est définie de façon récurrente:

a) La chaîne vide **""** précède lexicographiquement toutes les autres chaînes.

b) La chaîne A = "a1a2a ... ap" (p caractères) précède lexicographiquement la chaîne B = "b1b2 ... bm" (m caractères) si l'une des deux conditions suivantes est remplie:

1) 'a1' < 'b1'

2) 'a1' = 'b1' et

"a2a3 ... ap" précède lexicographiquement "b2b3 ... bm"

Exemples

"ABC" précède **"BCD"**	car **'A'<'B'**
"ABC" précède **"B"**	car **'A'<'B'**
"Abc" précède **"abc"**	car **'A'<'a'**
"ab" précède **"abcd"**	car **""** précède **"cd"**
" ab" précède **"ab"**	car **' '<'a' (le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97)**

Conversions et tests

En tenant compte de l'ordre alphabétique des caractères, on peut contrôler le type du caractère (chiffre, majuscule, minuscule).

Exemples

```
if (C>='0' && C<='9') printf("Chiffre\n", C);
if (C>='A' && C<='Z') printf("Majuscule\n", C);
if (C>='a' && C<='z') printf("Minuscule\n", C);
```

Il est facile, de convertir des lettres majuscules dans des minuscules:

```
if (C>='A' && C<='Z') C = C-'A'+'a';
```

ou vice-versa:

```
if (C>='a' && C<='z') C = C-'a'+'A';
```

6. Travailler avec des chaînes de caractères

Les bibliothèques de fonctions de C contiennent une série de fonctions spéciales pour le traitement de chaînes de caractères. Sauf indication contraire, les fonctions décrites dans ce chapitre sont portables conformément au standard ANSI-C.

6.1. Les fonctions de <stdio.h>

La bibliothèque *<stdio>* nous offre des fonctions qui effectuent l'entrée et la sortie des données. A côté des fonctions **printf** et **scanf** que nous connaissons déjà, nous y trouvons les deux fonctions **puts** et **gets,** spécialement conçues pour l'écriture et la lecture de chaînes de caractères.

- **Affichage de chaînes de caractères**

printf avec le spécificateur de format **%s** permet d'intégrer une chaîne de caractères dans une phrase.

En plus, le spécificateur **%s** permet l'indication de la largeur *minimale* du champ d'affichage. Dans ce champ, les données sont justifiées à droite. Si on indique une largeur minimale négative, la chaîne sera justifiée à gauche. Un nombre suivant un point indique la largeur *maximale* pour l'affichage.

Exemples

```
char NOM[] = "hello, world";
```

printf(":%s:", NOM);	**->**	**:hello, world:**
printf(":%5s:", NOM);	**->**	**:hello, world:**
printf(":%15s:", NOM);	**->**	**: hello, world:**
printf(":%-15s:", NOM);	**->**	**:hello, world :**
printf(":%.5s:", NOM);	**->**	**:hello:**

puts est idéale pour écrire une chaîne constante ou le contenu d'une variable dans une ligne isolée.

Syntaxe: **puts(<Chaîne>)**

Effet: **puts** écrit la chaîne de caractères désignée par <Chaîne> sur *stdout* et provoque un retour à la ligne. En pratique, **puts(TXT);** est équivalent à **printf("%s\n",TXT);**

Exemples

```
char TEXTE[] = "Voici une première ligne.";
puts(TEXTE);
puts("Voici une deuxième ligne.");
```

- **Lecture de chaînes de caractères**

scanf avec le spécificateur **%s** permet de lire un mot isolé à l'intérieur d'une suite de données du même ou d'un autre type.

scanf avec le spécificateur **%s** lit un ***mot*** du fichier d'entrée standard *stdin* et le mémorise à l'adresse qui est associée à **%s**.

Exemple

```
char LIEU[25];
int JOUR, MOIS, ANNEE;
printf("Entrez lieu et date de naissance : \n");
scanf("%s %d %d %d", LIEU, &JOUR, &MOIS, &ANNEE);
```

Remarques importantes

- La fonction **scanf** a besoin des ***adresses de ses arguments:***
- Comme le nom d'une chaîne de caractères est le représentant de l'adresse du premier caractère de la chaîne, **il ne doit pas être précédé de l'opérateur adresse '&' !**
- La fonction **scanf** avec plusieurs arguments présuppose que l'utilisateur connaisse exactement le nombre et l'ordre des données à introduire! Ainsi, l'utilisation de **scanf** pour la lecture de chaînes de caractères est seulement conseillée si on est forcé de lire un nombre fixé de mots en une fois.

gets est idéal pour lire une ou plusieurs lignes de texte (p.ex. des phrases) terminées par un retour à la ligne.

Syntaxe: **gets(<Chaîne>)**

Effet: **gets** lit une ***ligne*** de de caractères de *stdin* et la copie à l'adresse indiquée par <Chaîne>.

Le retour à la ligne final est remplacé par le symbole de fin de chaîne**'\0'**.

Exemple

int MAXI = 1000;
char LIGNE[MAXI];
gets(LIGNE);

6.2. Les fonctions de <string>

La bibliothèque *<string>* fournit une multitude de fonctions pratiques pour le traitement de chaînes de caractères. Voici une brève description des fonctions les plus fréquemment utilisées.

Dans le tableau suivant, <n> représente un nombre du type **int**. Les symboles <s> et <t> peuvent être remplacés par :

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de **char**
- un pointeur sur **char** (cf chp sur les pointeurs)

Fonctions pour le traitement de chaînes de caractères

strlen(<s>)	fournit la longueur de la chaîne ***sans*** compter le '\0' final
strcpy(<s>, <t>)	copie <t> vers <s>
strcat(<s>, <t>)	ajoute <t> à la fin de <s>
strcmp(<s>, <t>)	compare <s> et <t> lexicographiquement et fournit un résultat: négatif si <s> précède <t> zéro si <s> est égal à <t> positif si <s> suit <t>
strncpy(<s>, <t>, <n>)	copie au plus <n> caractères de <t> vers <s>
strncat(<s>, <t>, <n>)	ajoute au plus <n> caractères de <t> à la fin de <s>

Remarques

- Comme le nom d'une chaîne de caractères représente une adresse fixe en mémoire, on ne peut pas 'affecter' une autre chaîne au nom d'un tableau; il faut bien copier la chaîne caractère par caractère ou utiliser la fonction **strcpy** respectivement **strncpy**:

strcpy(A, "Hello");

- La concaténation de chaînes de caractères en C ne se fait pas par le symbole '+' comme en langage algorithmique ou en Pascal. Il faut ou bien copier la deuxième chaîne caractère par caractère ou bien utiliser la fonction **strcat** ou **strncat**.

6.3. Les fonctions de <stdlib>

La bibliothèque *<stdlib>* contient des déclarations de fonctions pour la conversion de nombres en chaînes de caractères et vice-versa.

Les trois fonctions définies ci-dessous correspondent au standard ANSI-C et sont portables. Le symbole <s> peut être remplacé par :

- une chaîne de caractères constante
- le nom d'une variable déclarée comme tableau de **char**
- un pointeur sur **char**

Conversion de chaînes de caractères en nombres

atoi(<s>)	retourne la valeur numérique représentée par <s> comme **int**
atol(<s>)	retourne la valeur numérique représentée par <s> comme **long**
atof(<s>)	retourne la valeur numérique représentée par <s> comme **double**

Règles générales pour la conversion:

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

6.4. Les fonctions de <ctype>

Les fonctions de *<ctype>* servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés. Les fonctions de *<ctype>*sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes. Dans la suite, <c> représente une valeur du type **int** qui peut être représentée comme caractère.

Fonctions de classification et de conversion

Les fonctions de ***classification*** suivantes fournissent un résultat du type **int** différent de zéro, si la condition respective est remplie, sinon zéro.

La fonction:	***retourne une valeur différente de zéro,***
isupper(<c>)	si <c> est une majuscule **('A'...'Z')**
islower(<c>)	si <c> est une minuscule **('a'...'z')**
isdigit(<c>)	si <c> est un chiffre décimal **('0'...'9')**
isalpha(<c>)	si **islower(<c>)** ou **isupper(<c>)**
isalnum(<c>)	si **isalpha(<c>)** ou **isdigit(<c>)**
isxdigit(<c>)	si <c> est un chiffre hexadécimal (**'0'...'9'** ou **'A'...'F'** ou **'a'...'f'**)
isspace(<c>)	si <c> est un signe d'espacement (' ', **'\t'**, **'\n'**, **'\r'**, **'\f'**)

Les fonctions de ***conversion*** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère; la valeur originale de <c> reste inchangée:

tolower(<c>) retourne <c> converti en minuscule si <c> est une majuscule

toupper(<c>) retourne <c> converti en majuscule si <c> est une minuscule

B. LES FONCTIONS

Face à un problème complexe, tel que la réalisation d'un programme, on cherche à séparer les difficultés et à trouver une méthode de résolution pour aboutir à l'écriture d'un algorithme. C'est cette phase préliminaire que l'on appelle analyse.

L'analyse descendante est une méthode consistant à décomposer un problème complexe en sous problèmes indépendants les uns des autres, et dont la complexité est moindre que le problème initial. Chacun des sous problèmes peut lui-même être décomposé à son tour, jusqu'à ce que chacun des sous problèmes ainsi définis soient réalisables - c'est à dire qu'on puisse écrire les algorithmes résolvant ces problèmes.

La modularité des programmes ainsi obtenue assure :

- une meilleure qualité de programmation
- un code plus court et par suite une diminution du risque d'erreurs de syntaxe,
- une mise au point aisée et rapide des programmes,
- une facilité de maintenance accrue,
- une facilité d'intégration de modules à d'autres programmes,
- une centralisation de la résolution d'un problème autour d'un programme principal.

Il est immédiat que la qualité et l'élégance de la solution finale dépendront fondamentalement de la qualité de la décomposition choisie.

Les modules crées sont appelés les sous programmes ; en C, on parle de fonctions.

L'intérêt des sous-programmes est :

- *de permettre d'écrire un programme en suivant l'analyse descendant ;*
- *d'éviter d'écrire plusieurs fois une séquence d'instructions destinée à être exécutée plusieurs fois ; il suffit alors d'en faire une fonction et de l'appeler autant de fois que nécessaire ;*
- *de réutiliser facilement des traitements déjà écrits sous forme de fonctions;*
- *de partager le travail entre programmeurs lors de la résolution d'un problème informatique complexe.*

I. Définition d'une fonction

Avant d'être utilisée, une fonction doit être définie car pour l'appeler dans le corps du programme il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions qu'elle contient. La définition d'une fonction s'appelle "*déclaration*".

La déclaration d'une fonction se fait selon la syntaxe suivante:

type_de_donnee Nom_De_La_Fonction(type1 argument1, type2 argument2, ...)
{
liste d'instructions
}

```
float fexple (float x, int b, int c)

{
```

```
float val ; /* déclaration d'une variable "locale" à fexple

val = x * x + b * x + c ;

return val ;

}
```

Remarques:

- La première ligne de cette définition est l'*en-tête* de la fonction. Dans cet en-tête, type_de_donnee représente le type de valeur que la fonction est sensée retourner (char, int, float,...)
- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé *void*
- Si aucun type de donnée n'est précisé (cela est très vilain!), le type *int* est pris par défaut
- le nom de la fonction suit les mêmes règles que les noms de variables:
  le nom doit commencer par une lettre
  un nom de fonction peut comporter des lettres, des chiffres et les caractères _ et & (les espaces ne sont pas autorisés!)
  le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)
- 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 paramètres, 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;
}
```

NB : non seulement l'instruction return définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (n'oubliez pas qu'en C tous les modules sont des fonctions, y compris le programme principal).

Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en pro- duit un. C'est d'ailleurs ce que nous avons fait fréquemment avec printf ou scanf. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

II. Appel d'une fonction

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom (une fois de plus en respectant la casse) suivie d'une parenthèse ouverte (éventuellement des arguments)puis d'une parenthèse fermée. 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.

III. 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.*

III.1. Prototype d'une fonction

Le prototype d'une fonction est une description d'une fonction qui est définie plus loin dans le programme. On place donc le prototype en début de programme (avant la fonction principale *main()*).

Cette description permet au compilateur de "vérifier" la validité de la fonction à chaque fois qu'il la rencontre dans le programme, en lui indiquant:

- Le type de valeur renvoyée par la fonction
- Le nom de la fonction
- Les types d'arguments

Contrairement à la **définition** de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas le nom des paramètres (seulement leur type).

Syntaxe

Type_de_donnee_renvoyee Nom_De_La_Fonction(type_argument1, type_argument2, ...);

Exemple

void Affiche_car(char, int);int Somme(int, int);

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.

C. IV- Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même *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 (***cf.*** options -O de gcc), 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.

3. 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 fonction()
{
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

Voyez cet exemple de programme.

```
#include <stdio.h>

int i ;

  main()

{ void optimist (void) ;

for (i=1 ; i<=5 ; i++) optimist() ;

}

void optimist(void)

{

printf ("il fait beau %d fois\n", i) ;

}
```

Le programme affiche :

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

Ainsi, ici, le programme principal affecte à i des valeurs qui se trouvent utilisées par la fonction optimist.

Remarque :

La norme ANSI ne parle pas de variables globales, mais de variables externes. Le terme « global » illustre plutôt le partage entre plusieurs fonctions tandis que le terme « externe » illustre plutôt le par- tage entre plusieurs fichiers source. En C, une variable globale est partagée par plusieurs fonctions ; elle peut être (mais elle n’est pas obligatoirement) partagée entre plusieurs fichiers source.

3.1. La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (n'oubliez pas que, pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
main() {

.... }

int n ;
float x ;
fct1 (...)
{
.... }

fct2 (...) {

.... }
```

Les variables n et x sont accessibles aux fonctions fct1 et fct2, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En effet, pour d'évidentes raisons de lisibilité, on préférera regrouper les déclarations de toutes les variables globales au début du programme source.

3.2. La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.

De plus, ces variables se voient **initialisées à zéro**, avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

4. 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 :

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

Le programme 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.

4.1. Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent.

Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant.

On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur cette gestion dynamique de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 11.2) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

4.2. Les variables locales statiques

Il est toutefois possible de demander d'attribuer un emplacement permanent à une variable locale et qu'ainsi sa valeur se conserve d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **static** (le mot static employé sans indication de type est équivalent à static int).

En voici un exemple :

Exemple d'utilisation de variable locale statique

```
#include <stdio.h>
main()
{  void fct(void) ;
int n ;
for ( n=1 ; n<=5 ; n++)
fct() ;
```

```
}
void fct(void)
{  static int i ;
i++ ;

printf ("appel numéro : %d\n", i) ;
}
```

```
appel numéro : 1
appel numéro : 2
appel numéro : 3
appel numéro : 4
appel numéro : 5
```

La variable locale i a été déclarée de classe « statique ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée i qui n'aurait alors aucun rapport avec la variable i de fct.

V- Arguments d'une fonction

Il est possible de passer des arguments à une fonction, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la fonction puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments. Le passage d'arguments à une fonction se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la fonction.

Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre au risque, sinon, de générer une erreur lors de la compilation...

Un argument peut être une constante, une variable, une expression ou une autre 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 :

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);
}
```

Ce programme 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 donc modifiés par la fonction.

Par exemple :

```
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);
}
```

Le programme initialise les éléments du tableau tab.

VI- 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 lorsqu'on utilise l'option -Wall de gcc :

```
% gcc -Wall prog.c
prog.c:5: warning: return-type defaults to `int'
prog.c: In function `main':
prog.c:11: warning: control reaches end of non-void function
```

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 :

```
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.

VIII- Pointeur et fonction

Passage de paramètres par référence

Pour le compilateur, **p* est la variable pointée par *p*, cela signifie que l'on peut, pour le moment du moins, utiliser indifféremment **p* ou *x*". En précisant "pour le moment du moins", j'avais déjà l'intention de vous montrer des cas dans lesquels ce n'était pas possible. C'est à dire des cas dans lesquels on connait l'adresse d'une variable mais pas son nom.

Permutation de deux variables

A titre de rappel, observez attentivement le programme suivant :

```
#include<stdio.h>
void echange(int x, int y)
{
        int t = x; x = y; y = t;
}
void main()
{
        int a = 1;
        int b = 2;
        printf("a = %d, b = %d\n", a, b);
        echange(a, b);
        printf("a = %d, b = %d\n", a, b);
}
```

A votre avis, affiche-t-il

a = 1, b = 2
a = 2, b = 1

ou bien

a = 1, b = 2
a = 1, b = 2

Méditons quelque peu : la question que l'on se pose est "Est-ce que le sous-programme *echange* échange bien les valeurs des deux variables *a* et *b*" ? Il va de soi qu'il échange bien les valeurs des deux variables *x* et *y*, mais comme ces deux variables ne sont que des **copies** de *a* et *b*, cette permutation n'a aucun effet sur *a* et *b*. Cela signifie que la fonction *echange* ne fait rien, on aurait pu écrire à la place un sous-programme ne contenant aucune instruction, l'effet aurait été le même. Ce programme affiche donc

a = 1, b = 2
a = 1, b = 2

Remarques

Par exemple, l'instruction *scanf*("%*d*", & *x*) permet de placer une valeur saisie par l'utilisateur dans *x*, et *scanf* est un sous-programme. Vous conviendrez donc que **la variable *x* a été passée en paramètre par référence**. Autrement dit, que la valeur de *x* est modifiée dans le sous-programme *scanf*, donc que la variable permettant de désigner *x* dans le corps de ce sous-programme n'est pas une copie de *x*, mais la variable *x* elle-même, ou plutôt un **alias de la variable *x***.

Vous pouvez d'ores et déjà retenir que

nomsousprogramme(…, &x, …)

sert à passer en paramètre la variable *x* par référence. Et finalement, c'est plutôt logique, l'instruction

nomsousprogramme(…, *x*, …)

passe en paramètre la **valeur** de *x*, alors que

nomsousprogramme(…, &*x*, …)

passe en paramètre l'**adresse** de *x*, c'est-à-dire un moyen de retrouver la variable *x* depuis le sous-programme et de modifier sa valeur.

Cependant, si vous écrivez *echange*(&*a*, &*b*), le programme ne compilera pas.

En effet, le sous-programme *echange* prend en paramètre des *int* et si vous lui envoyez des adresses mémoire à la place, le compilateur ne peut pas "comprendre" ce que vous voulez faire... Vous allez donc devoir modifier le sous-programme *echange* si vous voulez lui passer des adresses mémoire en paramètre.

<u>Utilisation de pointeurs</u>

On arrive à la question suivante : dans quel type de variable puis-je mettre l'adresse mémoire d'une variable de type entier ? La réponse est *int**, un pointeur sur *int*. Observons le sous-programme suivant :

```
void echange(int* x, int* y)
{
        int t = *x;
        *x = *y;
        *y = t;
}
```

x et *y* ne sont pas des *int*, mais des pointeurs sur *int*. De ce fait le passage en paramètre des deux adresses &*a* et &*b* fait pointer *x* sur *a* et *y* sur *b*. Donc **x* est un alias de *a* et **y* est un alias de *b*. Nous sommes, comme décrit dans l'introduction de ce chapitre dans un cas dans lequel on connait l'adresse d'une variable, mais pas son nom : dans le sous-programme *echange*, la variable *a* est inconnue (si vous l'écrivez, ça ne compilera pas...), seul le pointeur **x* permet d'accéder à la variable *a*.

Il suffit donc, pour écrire un sous-programme prenant en paramètre des variables passées par référence, de les déclarer comme des pointeurs, et d'ajouter une * devant à chaque utilisation.

VIII- Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction.
Un pointeur sur une fonction ayant pour prototype :
type fonction(type_1,...,type_n);
est de type ***type* (*)(*type_1,...,type_n*);**
Ainsi, une fonction operateur_binaire prenant pour paramètres deux entiers et une fonction de type int, qui prend elle-même deux entiers en paramètres, sera définie par :

*int operateur_binaire(int a, int b, int (*f)(int, int))*

Sa déclaration est donnée par
int operateur_binaire(int, int, int(*)(int, int));

Pour appeler la fonction operateur_binaire, on utilisera comme troisième paramètre effectif l'identificateur de la fonction utilisée, par exemple, si somme est une fonction de prototype : *
int somme(int, int);

on appelle la fonction operateur_binaire pour la fonction somme par l'expression
operateur_binaire(a,b,somme)

Notons qu'on n'utilise pas la notation &somme comme paramètre effectif de operateur_binaire.

Pour appeler la fonction passée en paramètre dans le corps de la fonction operateur_binaire, on écrit (*f)(a, b).

Par exemple

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

Pour l'appel : printf("%d\n",operateur_binaire(a,b,somme));

Quelques fonctions utiles:

La fonction **int strlen** (**const char *** string) : renvoie la longueur de la chaîne pointée par *string* sans compter le caractère '\0'

La fonction **char * strcpy** (**char *** dest, **const char *** src) copie la chaîne pointée par *src* (y compris le caractère '\0' final) dans la chaîne pointée par *dest*.

La fonction **int strcmp** (**const char *** s1, **const char *** s2) compare les deux chaînes *s1* et *s2*. Elle renvoie un entier : négatif si *s1*<*s2,* nul si *s1*=*s* ou positif si *s1* > *s2*

La fonction **char * strcat** (**char * dest**, **const char *** src) ajoute la chaîne *src* à la fin de la chaîne dest en écrasant le caractère '\0' `a la fin de *dest*, puis en ajoutant un nouveau caractère '\0' final.

int atoi(const char *CH) retourne la valeur numérique représentée par CH comme int

long atol(const char *CH) retourne la valeur numérique représentée par CH comme long

double atof(const char *CH) retourne la valeur numérique représentée par CH comme double

long strtol(const char * restrict nptr, char ** restrict endptr, int base) retourne la chaine *nptr* considérées comme étant en base de numération *base* comme un long. Le reste de la chaine qui n'a pas pu etre convertie est pointée par *endptr, positionne errno en cas d'erreur.

De même **strtod** convertit une chaîne en double.