

programme sont dit “dynamiques”. On se limite ici aux tableaux statiques à une seule dimension, analogue aux vecteurs manipulés en mathématiques. Ils se déclarent sous la forme :

```
type_d_élément variable_tableau[taille];
```

Par exemple, pour déclarer un tableau de 10 entiers : `int tab[10];`

Les éléments sont toujours numérotés on dit (indités) à partir de 0. Pour un tableau de taille N, les indices des éléments vont donc de 0 à N-1. Attention il n'existe pas dans le langage d'opérations fonctionnant sur la totalité du tableau : on ne peut pas en une seule opération copier un tableau dans un autre, ni comparer 2 tableaux, ni demander à un tableau quelle est sa taille!!!

Exemples :

```
int main() {
int t1[10], t2[10];
t1[0] = 12; // range 12 dans la 1ère case de t1
t1[9] = 5; // range 5 dans la dernière case de t1
t2[1] = t1[0]; // copie la 1ère case de t1 dans la 2ème case de t2
t1[10] = 9; // ERREUR t1[10] n'existe pas ! t1 est indicé de 0 à 9
t1 = t2; // ERREUR on ne peut pas affecter le contenu de t2 dans t1
{
int i;
for (i=0; i < 10; i++)
t1[i] = t2[i];
} // OK, copie de t2 dans t1
```

3 Lien entre pointeurs et tableaux.

On peut considérer un nom de tableau comme l'adresse en mémoire de son premier élément. Un nom de tableau peut donc être assimilé à une sorte de pointeur constant (qu'on ne peut pas modifier). On peut donc accéder à ses éléments par une opération de dépointage. Exemples :

```
int main() {
int t1[10], t2[10];
*t1 = 12; // range 12 dans la 1ère case (car t1 est l'adresse de cette case)
*(t1 + 9) = 5; // range 5 dans la dernière case de t1 (9 cases après la première)
*(t2 + 1) = *t1; // copie la 1ère case de t1 dans la 2ème case de t2
*(t1 + 10) = 9; // ERREUR t1[10] n'existe pas !
{
int i;
for (i=0; i < 10; i++)
*(t1+i) = *(t2+i);
} // copie de t2 dans t1
```

Le code de copie d'un tableau dans un autre, donné juste au dessus en exemple peut aussi être exprimé en copiant les adresses des tableaux dans des variables pointeurs :

```
int main() {
int t1[10], t2[10];
// on suppose que t2 est rempli par des données, à copier dans t1
int i, *p1=t1, *p2=t2;
for (i=0; i < 10; i++)
*p1++ = *p2++;
}
```

A noter que l'on ne gagne pas vraiment en lisibilité! C'est possible mais à éviter...

4 Variables chaînes de caractères

Les chaînes de caractères, que je noterai simplement “chaînes”, sont un type de données très utilisé, notamment pour l'affichage et la saisie de messages, noms, bref pour les données textuelles.

En C une variable chaîne n'est rien d'autre qu'un tableau de caractères à 1 dimension, et qui contient un caractère spécial marquant la fin de la chaîne. Si l'on veut stocker un message de 20 caractères, on doit donc utiliser un tableau d'au moins 21 caractères (donc 1 caractère de plus), pour réserver la place de ce marqueur de

fin de chaîne. Le tableau peut sans problème être plus grand : grâce au marqueur de fin, s'il y a des caractères superflus en fin de tableau, situés derrière le marqueur de fin, ils seront ignorés par les fonctions de manipulation et d'affichage de chaîne.

Le marqueur de fin est par convention le caractère (non affichable ni imprimable) de code 0. On peut le noter `'\0'`, mais comme le type caractère est un vu par C comme un type numérique, on peut aussi l'écrire 0, mais attention en aucun cas `'0'` ni `"0"!!! ATTENTION`, sans ce marqueur, un tableau n'est pas une chaîne et ne peut pas être manipulé par les fonctions standards sur les chaînes!

Toutes les possibilités et restrictions précédemment mentionnées pour les tableaux sont aussi valables pour les chaînes : pas de copie ni de comparaison, utilisation possible de pointeurs.

```
int main() {
char ch1[10]="012345678"; // attention à bien respecter le nombre d'éléments!
// il faut penser au marqueur de fin de chaîne qui utilise 1 caractère!!!
char ch2[10];

ch2[0] = ch1[0]; // ch2[0] vaut maintenant '0'
*(ch2 + 1) = 'B'; // ch2[1] vaut 'B'
printf("%s\n", ch1); // OK
printf("%s\n", ch2); // ERREUR on n'a pas mis de marqueur de fin dans ch2
ch2[2] = 0; // ch2 est maintenant une vraie chaîne
printf("%s\n", ch2); // OK, affiche "0B"
}
```

La présence du marqueur de fin permet de connaître la longueur de la chaîne (qui peut être plus petite que celle du tableau qui la contient). Exemple de copie de chaîne :

```
int main() { // copie raisonnable
char ch1[10]="012345678";
char ch2[10];
int i = 0;
while (ch1[i] !=0)
    ch2[i] = ch1[i++]; // on incrémente i pour l'itération suivante
ch2[i] = 0; // on place la fin de chaîne dans ch2
}
```

Comme 0 est aussi interprété comme le booléen FAUX dans les tests, on peut encore compacter l'écriture en utilisant des pointeurs :

```
int main() { // copie raccourcie
char ch1[10]="012345678";
char ch2[10];
char *p1 = ch1, *p2 = ch2;
while (*p1) // donc tant que (*p1 != 0)
    *p2++ = *p1++; // on incrémente pour l'itération suivante
*p2 = 0; // on place la fin de chaîne dans ch2
}
```

Et comme l'affectation est vue comme une expression renvoyant la valeur affectée, on peut même condenser en :

```
int main() { // copie pour "geek"
char ch1[10]="012345678";
char ch2[10];
char *p1 = ch1, *p2 = ch2;
while (*p2++ = *p1++) // donc tant que (*p1 != 0)
    ; // il n'y a plus rien à faire ici

// la copie du marqueur a été faite dans la boucle
}
```

Evidemment, là encore ce qui est faisable n'est pas pour autant recommandé : la lisibilité est le critère à privilégier pour 95% du code.

VI Fonctions

Quelques éléments essentiels des fonctions en C :

- il a peu de différence syntaxique entre fonction et "procédure" (à la ADA) : une procédure est simplement une fonction qui renvoie `void` (c'est à dire rien, néant);
- un appel ou une définition de fonction est **toujours** suivie d'une paire de parenthèses. Ces parenthèses entourent les paramètres, ou rien du tout s'il n'y a pas de paramètres;
- les paramètres effectifs (lors de l'appel) sont toujours passés par valeur : pas d'équivalent du mode "out" de ADA;
- pas de définition de fonction à l'intérieur d'une autre définition de fonction.

1 Syntaxe

Forme d'une définition de fonction :

```
type_retourné nom_de_fonction ( type_param_1 nom_param_1,
                               type_param_2 nom_param_2,
                               ...
                               )
{
    déclarations_de_variables_locales
    liste_d_instructions
}
```

- si la fonction ne doit rien retourner, le type retourné est `void`
- si le type retourné n'est pas `void`, alors il doit y avoir une instruction `return expression;` retournant le résultat;
- si le type retourné est `void`, on peut trouver l'instruction `return;` sans expression associée, qui sert uniquement à quitter immédiatement la fonction (pas le programme).

Forme d'un appel de fonction :

```
nom_de_fonction ( param_effectif_1 , param_effectif_2, ... )
```

- tout paramètre entier plus court que `int` est traduit en `int`
- tout paramètre `float` est traduit en `double`

Déclaration préliminaire Une difficulté syntaxique se présente lorsque le programmeur veut appeler une fonction qui se trouve être déclarée **plus bas** dans le code que l'endroit de l'appel : n'ayant pas encore rencontrée la définition de la fonction que l'on appelle, le compilateur va générer un message d'erreur et le programme ne sera pas complètement compilé. Or on ne peut pas toujours placer les définitions de fonctions avant les appels, notamment dans le cas de deux fonctions s'appelant récursivement l'une l'autre.

Dans cette situation on va placer une "déclaration préliminaire" de la fonction en tête du programme avant l'appel. Cette déclaration consiste simplement à dupliquer l'entête de fonction sans son corps accolades, suivi d'un point-virgule :

```
type_retourné nom_de_fonction ( type_param_1 nom_param_1,
                               type_param_2 nom_param_2,
                               ...
                               ) ;
```

Le compilateur passe alors d'abord sur la déclaration préliminaire, lorsqu'il rencontre l'appel de la fonction il peut vérifier que le type et le nombre des paramètres correspondent bien, et il effectuera à nouveau cette vérification lorsqu'il rencontrera la définition complète de la fonction. Exemple :

```
// déclarations préliminaires en tête du code
int fonction_un(int n);
int fonction_deux(int n);

// définitions et appel des fonctions
int fonction_un(int n) {
    if (n > 0) {
```

```

    printf ("Dans fonction_un\n");
    fonction_deux(n-1);
}
}

int fonction_deux(int n) {
    if (n > 0) {
        printf ("Dans fonction_deux\n");
        fonction_un(n-1);
    }
}

int main() {
    fonction_un(5);
    return 0;
}

```

Ces déclarations préliminaires sont très souvent utilisées, même lorsqu'elles ne sont pas strictement indispensables, car elles permettent de rendre le code compilable indépendamment de la position respective des appels et des définitions de fonctions. Elles sont de plus nécessaires pour découper une application en plusieurs fichiers (voir la section sur la modularité).

2 Quelques exemples de fonctions

Calcul de factorielle :

```

int factorielle(int n) {
    int i, resultat=1;
    for (i=2; i <= n; i++)
        resultat *= i;
    return resultat;
}

```

Calcul de factorielle récursif : proposé pour l'exemple, car la récursivité est sans intérêt pour ce problème.

```

int factRec(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factRec(n-1);
}

```

Valeur numérique d'un caractère chiffre entre '0' et '9' :

```

int valNum(char c) {
    if (c >= '0' && c <= '9')
        return c - '0'; /* les chiffres ont des codes successifs */
    else /* erreur */
        return -1; /* par convention */
}

```

Procédure pour passer une ligne sur la console :

```

void passeLigne() {
    printf("\n");
}

```

3 Simuler les paramètres en mode "in out"

Passage par copie. Les paramètres sont toujours passé par copie. Ainsi dans le programme suivant, la valeur affichée sera 3 et pas 27 :

```

#include <stdio.h>
void valAuCube(int n) {
    n = n * n * n; /* si n valait 3 alors il vaut 27 */
}

int main() {
    int a = 3;
    valAuCube(a);
    printf("%d", a);
    return 0;
}

```

En effet, c'est le paramètre formel de nom `n` (qu'on peut considérer comme une variable locale à la fonction `valAuCube`) qui a été modifié. La variable `a` du programme principal est restée inchangée.

Passage par adresse. Par contre, si l'on passe *l'adresse* de `a`, alors on va pouvoir changer sa valeur depuis une fonction :

```

#include <stdio.h>
void valAuCubeBis(int *n) /* n est un pointeur sur une variable de type int */
{
    /* si à l'adresse contenue dans n il y avait 3 alors il y a 27 */
    *n = (*n) * (*n) * (*n);
}

int main() {
    int a = 3;
    valAuCubeBis(&a); /* a est passé par adresse */
    printf("%d", a);
    return 0;
}

```

Syntaxiquement, la modification consiste à placer `&` dans l'appel devant le paramètre effectif, et à placer `*` devant chaque apparition du paramètre formel correspondant (en fait nous utilisons dans la fonction des variables pointeurs, pointant sur la variable du programme principal).

La question à se poser lorsqu'on écrit une fonction est la suivante : "les paramètres seront ils des variables que l'on souhaite modifier dans la fonction?". Si la réponse est non, on utilise un passage par valeur, si c'est oui on utilise un passage par adresse, appelé encore passage par pointeur (si on voulait être plus précis on dirait que l'on passe *par copie l'adresse* de la variable). Notez que vous avez déjà utilisé ces appels par adresse avec la fonction de saisie `scanf()` qui évidemment doit modifier le contenu des variables passées en paramètres.

Cas particulier des tableaux Comme on l'a vu plus haut un nom de tableau peut être vu comme l'adresse de la zone mémoire où est stocké le contenu du tableau. Par conséquent passer un tableau en paramètre c'est passer son adresse, et tout passage de tableau est toujours un passage par adresse. Donc on en déduit que :

- on peut toujours modifier un tableau reçu en paramètre dans une fonction ;
- il est inutile de préfixer avec l'opérateur d'adressage `&` un nom de tableau que l'on veut passer en mode "in out", puisque par défaut c'est déjà une adresse que l'on manipule. Les compilateurs un peu anciens signalent une erreur dans ce cas, certains plus récents sont plus compréhensifs.

Ainsi lorsque l'on veut saisir une chaîne pas besoin de `&`, car une chaîne est stockée dans un tableau :

```

int main() {
    char chaine[80];
    printf("saisir une chaîne :\n");
    scanf("%s", chaine); // sans &
    printf("%s", chaine);
}

```

4 Les macros du pré-processeur

Le pré-processeur permet de simuler des appels de fonctions que l'on nomme *macros* ou macro-instructions : en fait le corps de la fonction sera inséré dans le texte du programme à chaque appel (d'autres langages de

programmation ont une notion ressemblante : les fonctions *inline*). Le contrôle syntaxique de ces macros est minimal, **il faut donc les éviter** et en tous les cas les réserver à des usages très simples. Exemple :

```
#define CUBE(n) (n * n * n)
```

```
int main() {
    int a = 3;
    printf("%d", CUBE(a));
    return 0;
}
```

le texte du programme sera transformé par le pré-processeur en :

```
int main() {
    int a = 3;
    printf("%d", (a * a * a));
    return 0;
}
```

L'utilisateur de macros doit conserver à l'esprit les points suivants :

- les pseudo-paramètres sont spécifiés entre parenthèses, séparés par des virgules *sans espaces* s'il y en a plusieurs, et sans type, juste après le nom de macro ;
- le corps de la macro est *tout* ce qui suit sur la ligne ; si la ligne devient trop longue, on peut placer un `\` et continuer sur la ligne suivante ;
- comme on ne sait pas a priori dans quel contexte le corps de la macro va être recopiée, il est prudent de le mettre entre parenthèses, pour éviter les problèmes de priorité d'opérateurs ;
- l'appel est remplacé par une copie : on gagne du temps de calcul (un tout petit peu), mais le programme est plus long ;
- une macro n'étant pas une vraie fonction, elle n'aura pas d'adresse mémoire lors de l'exécution du code donc sera inutilisable avec un pointeur de fonction ; par ailleurs aucune récursivité n'est possible.

VII Tableaux dynamiques, tableaux à 2 dimensions

1 Tableaux dynamiques

Les tableaux vus précédemment avaient une taille fixée une fois pour toute lors de leur déclaration dans le code source du programme. C'est une méthode tout à fait légitime pour de nombreuses applications, mais qui trouve ses limites lorsqu'on ne connaît pas la taille du tableau à manipuler au moment où l'on écrit le code (par exemple on fait du calcul de matrices mais que la taille des matrices ne sera connue qu'à l'exécution). Il existe donc un procédé pour créer des tableaux "dynamiquement" lors de l'exécution du programme (on parle d'allocation dynamique).

Pour cela on inclut la bibliothèque `stdlib.h` ou encore `alloc.h`. On ne déclare pas un tableau proprement dit, mais une variable pointeur, et on alloue (c.à.d. réserve) un bloc de mémoire de la taille nécessaire pour stocker le tableau. La fonction d'allocation standard est `malloc()`, qui prend en argument la taille du bloc mémoire à réserver, et retourne l'adresse du premier élément du bloc qu'il faudra conserver pour accéder au tableau. Pour connaître la taille du bloc mémoire à passer en paramètre à `malloc` il suffit de multiplier la taille du tableau (en nombre d'éléments) par la taille occupée par un élément en mémoire, laquelle s'obtient par le mot-clé `sizeof`. Enfin la fonction `malloc` retourne un pointeur de type générique `void*`, et il faut une conversion de type entre parenthèses pour éviter les messages d'avertissements du compilateur lorsqu'on sauve cette adresse. Exemple d'allocation dynamique d'un tableau d'entiers :

```
#include <stdlib.h>
int main() {
    int taille;
    int *tab; // c'est un pointeur d'entiers
    printf("Entrez la taille du tableau : ");
    scanf("%d", &taille); // on suppose que taille est > 0
    if (taille <= 0) // erreur
        return 1; // on quitte la fonction "main"
    // ici la taille est positive
    // allocation : on réserve "taille" entiers, noter la conversion de type
    // entre parenthèse avant "malloc"
    tab = (int*) malloc(taille * sizeof(int));
```

```

// on peut maintenant utiliser tab comme un tableau
tab [0] = 12;

// après utilisation , on libère la mémoire
free(tab);
return 0;
}

```

Comme montré dans l'exemple, on peut rendre la mémoire au système d'exploitation ("désallouer") lorsque l'utilisation du tableau est terminée, avec la fonction `free()` qui prend en paramètre le pointeur qui a été utilisé comme tableau.

2 Tableaux à 2 dimensions

Nous nous limiterons pour les tableaux multidimensionnels à ceux à 2 dimensions, sachant que les principes de base s'étendent naturellement. La déclaration d'un tableau statique bidimensionnel est de la forme :

```
type_d_élément variable_tableau[taille_dim_1][taille_dim_2];
```

L'accès aux éléments se fait de manière très semblable aux tableaux unidimensionnels. Exemple :

```
tab2D[0][0] = 12; affecte 12 à la case d'indice (0,0).
```

Néanmoins la structure des tableaux 2D est assez spécifique : on a affaire en fait à un tableau de lignes, c'est à dire un tableau de tableaux unidimensionnels (ce qui fournit les 2 dimensions souhaitées). Soit un élément de tableau de type T, on a vu que le nom d'un tableau de tels éléments est vu comme une adresse de T, c'est à dire un T*. Si on prend maintenant un tableau de tels tableaux, il y a à nouveau passage à l'adresse et on obtient un objet de type T** (un pointeur de pointeur de T) : c'est le type que nous devons gérer lors d'une allocation dynamique. Nous donnons ci-dessous un exemple d'allocation statique et un d'allocation dynamique de 2 tableaux d'entiers bidimensionnels.

```

#include <stdlib.h>
int main() {
    int t1[5][10]; // tableau 2D statique d'entiers

    // exemples d'utilisation
    t1[0][0] = 5;
    t1[4][9] = t1[0][0];

    int **t2; // pour un tableau dynamique 2D d'entiers
    // allouer le tableau de tableaux
    t2 = (int**) malloc(5 * sizeof(int *));
    // allouer chacun des tableaux lignes
    {
        int i;
        for (i = 0; i < 5; i++)
            t2[i] = (int *) malloc(10 * sizeof(int));
    }
    // on peut maintenant utiliser t2 comme on utiliserait t1
    t2[0][0] = 5;
    t2[4][9] = t2[0][0];

    // rendre la mémoire au système après utilisation
    {
        int i;
        for (i = 0; i < 5; i++)
            free(t2[i]);
    }
    free(t2);
}

```