Pointeurs et tableaux

Un tableau est assimilé à l'adresse de son premier élément

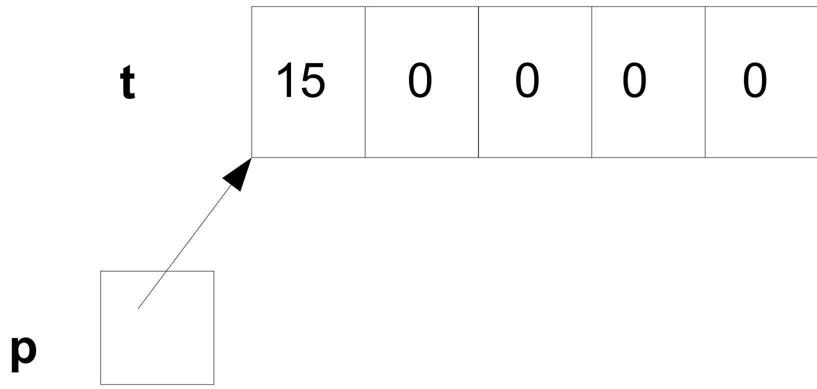
&t[0] est égal à t

Une variable de type tableau de t est un pointeur sur t -
t pointeur constant sur le début du tableau

On peut déclarer un tableau d'entiers avec le type int*

```
int t[5]={0,0,0,0,0};
int * p;
printf("t=%p, &t[0] =%p\n", t, &t[0]);
p=t;
*p=15;
printf("p=%p, t[0] =%p\n", p, t[0]);
```

```
Affichage :
t=001CCE68,&t[0]=001CCE68
p=001CCE68, t[0]=15
```


t
15
0
0
0
0
p

t pointeur constant sur le début du tableau

Conséquence : on ne peut modifier t.
Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle)

```
int A[5] = {0,0,0,0,0};
char B[5] =  {10,10,10,10,10};

A = B;           /* IMPOSSIBLE -> ERREUR !!! */
```

Dans cet exemple, nous essayons de copier l'adresse de B dans A
Impossible car l'adresse représentée par le nom d'un tableau reste toujours constante.

Arithmétique des Pointeurs

- pointeur +/- entier

T t[100]; int i; T *p;

p+i est de type T*
pointe i objets de type t après p

il faut bien faire attention avec ce genre d'opération à ne pas sortir du bloc mémoire, car le C n'effectuera aucun test pour vous.

- compatibilité avec les tableaux

t[i] est évalué comme *(t+i)

en particulier si p pointe sur t[i] alors p+k pointe sur t[i+k]

- les opérations ++ et - - s'appliquent aux pointeurs

p++ pointe sur l'objet suivant de type T

Arithmétique des Pointeurs (2)

- différence de pointeurs (de même type) nombre d'objets compris entre les deux adresses

(p+i)-p vaut i

&t[i+1] - &t[i] vaut 1

- comparaisons possibles entre pointeurs : ==, !=

(p) idem (p!=NULL) (dans une condition de if ou de boucle)
(!p) idem (p==NULL)

- en paramètre de fonction : T * tab est équivalent à T tab[]

Exemple copie de tableaux (de s dans t)

```
void copie1(int s[], int t[], int taille){
int i;
for (i=0;i<taille;i++)
   t[i]=s[i];
}

void copie2(int s[], int t[], int taille){
int i;
for (i=0;i<taille;i++)
   *(t+i)=*(s+i);
}

void copie3(int * s, int * t, int taille){
int i;
for (i=0;i<taille;i++) {
   *t=*s;
    t++;
    s++;
}
}
```

Les 3 fonctions requièrent que s et t soient des tableaux alloués par l'appelant.

Un autre exemple

on cherche un 0 dans la zone pointée par tab sur une longueur de nb

```
void cherche_zero(int* tab, int nb){
int *p;
int i=nb;
for (p=tab; i>0; i- -, p++ )
if ( *p == 0 ) return 1;
return 0;
}

void f(){
int a[100];
...
if ( cherche_zero(&a[10], 15 ) ) ...
}
```

Ici on se sert de la fonction pour chercher un 0 dans le tableau à partir de la case 10 à la case 24.

Tableaux dynamiques

Un des intérêts des pointeurs et de l'allocation dynamique = permettre de décider de la taille d'une variable au moment de l'exécution, comme par exemple pour les tableaux.

Pour allouer un tableau de n entiers (n étant connu à l'exécution) :
 déclarer une variable de type int *
+ allouer une zone mémoire pour n entiers :

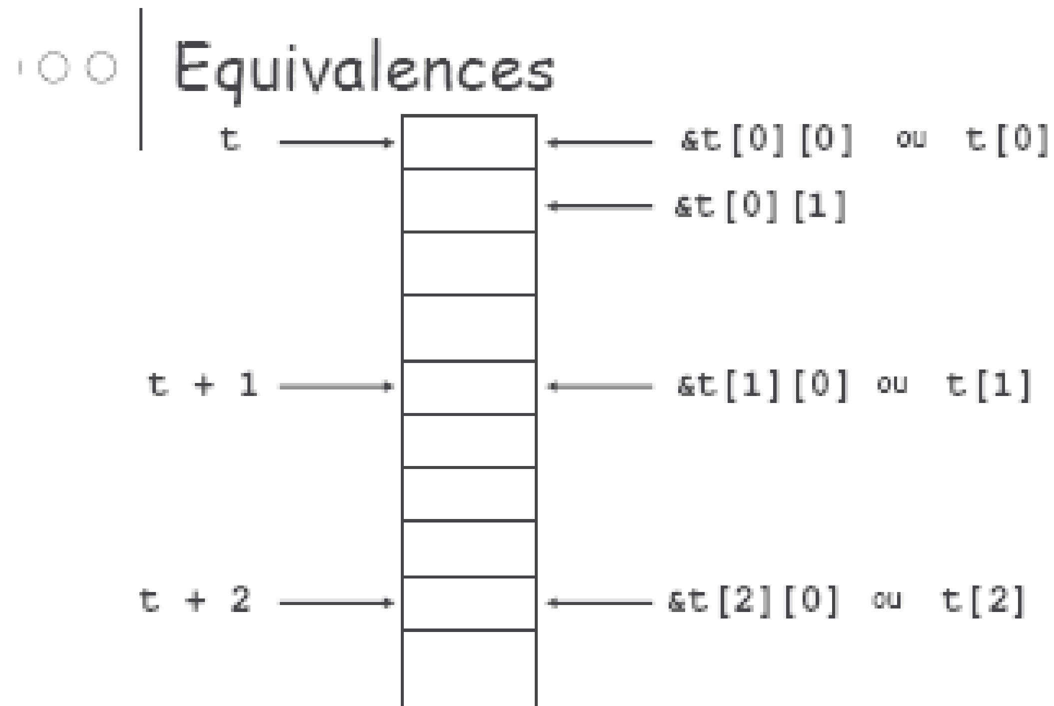
```
int n;
int * tableau ;

scanf("n=%i ", &n);    /* entrée au clavier de la valeur de n */
tableau = malloc(n*sizeof(int));
if (tableau  != NULL)
{
    /* opérations sur le tableau */
     ...
    free( tableau );
}
```

Cas des tableaux multidimentionnels

Ici tableaux à deux dimensions

int t[3][4]

- t tableau de 3 blocs de 4 entiers
- t pointeur sur le début de la zone donc t équivalent à &t[0][0] ou à t[0]



Chaînes de caractères

Chaîne (string) = suite de caractères terminée par le caractère \0.

En C, pas de type chaîne spécifique :

-> Représentation des chaînes de car. par un tableau de car.

Déclaration : on utilise un tableau de char : char[],
En argument : on passe un pointeur dans un tableau : char*.

Exemple : `char ch[10] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};`
les cases restantes sont mises à \0

`char s[3];`
sans initialisation : dangereux car pas de \0

Initialisation simplifiée : ajout automatique de \0

```
char ch[10] = "bonjour";
```

Longueur d'une chaîne

```
int lg(char * s){
int i=0;
while (s[i])
  i++;
return i;
}
```

Les chaînes sont passées aux fonctions par référence
sous forme de pointeur vers le premier caractère.
Type de l'argument : char*

Parcours du tableau jusqu'au marqueur de fin \0
Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères (inutile de la passer en paramètre quand on manipule des chaînes de caractères).

Autre exemple

```
/*arg : c                type : char -> char
post : retourne le caractère majuscule correspondant si c
est une lettre minuscule, retourne c sinon*/
char en_majuscule(char c){
if ( c >= ’a’ && c <= ’z’ )
return c - ’a’ + ’A’;
else
return c;
}

/* modifie le contenu de s*/
/* met en majuscules toutes les lettres minuscules de s*/
void chaine_en_majuscule(char* s){
char *p;
for ( p=s; *p; p++ )
*p = en_majuscule(*p);
}
```

Fonctions standards sur les chaînes

Fonctions standards : documentées dans man string.

Quelques fonctions utiles : #include <string.h>

strlen   longueur d'une chaine
strcmp compare deux chaînes lexicographiquement
strcpy   copie de chaîne
strcat   concaténation de chaînes
strncpy
strncat versions limitées à n caractères
strncmp
strchr   recherche d'un caractère dans une chaîne
strstr   recherche d'une sous-chaîne dans une chaîne

Exemples de prototypes :

```
size_t strlen(const char *s);
char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
```

Exemples d'utilisation :

```
char d[50]= "bonjour ";
char *s = "à tous !";
char * p;
p=strcat(d, s);          p pointe sur "bonjour à tous!"
...

void make_path(const char* dir, const char* file){
char buf[1024]
if ( strlen(dir) + strlen(file) + 2 > sizeof(buf) ) exit(0);
strcpy(buf, dir);
strcat(buf, "/");
strcat(buf, file);
...
```

Chaînes constantes

On peut utiliser une chaîne entre " en dehors des initialisations de tableaux.

```
printf("toto\n");
char* s = "toto\n";
```

Effets et contraintes :
compilation : crée un tableau anonyme bien initialisé,
le tableau est global et constant,

```
const char anonymeXXX[] = "toto" ;
```

exécution : renvoie un pointeur sur ce tableau.

```
printf(anonymeXXX) ; char* s = anonymeXXX ;
```

Rq : à l'exécution, aucune allocation ni copie de chaîne n'a lieu,
le tableau ne doit pas être modifié.

Attention !

Il existe une différence importante entre les deux déclarations:

```
char A[] = "Bonjour !";   /* un tableau  */
char *B  = "Bonjour !";   /* un pointeur */
```

A est un tableau qui a exactement la taille pour contenir la chaîne de caractère
et le '\0'.
Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire. (A est un pointeur constant)

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée o
affichée, mais pas modifiée.

```
char *A = "chaîne";
char *B = "Deuxième chaîne";
A = B;
```

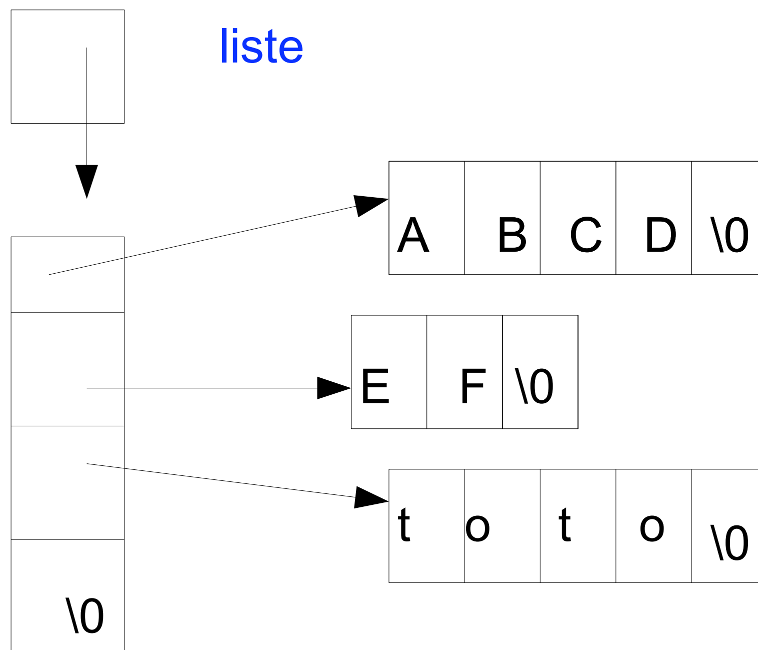
Tableau de chaînes de caractères

Pour une gestion dynamique : le déclarer comme un pointeur sur chaîne donc de type char**

char ** liste

Procédure de saisie de cet tableau de chaînes :
char saisie(void);**
Elle retournera le pointeur sur le tableau saisi.

On aura la situation suivante :



```
char ** saisie(void){
char** l ;
char** p;
char ligne[81];
/* on saisit au plus 10 chaînes de caractères*/
/* allocation de la liste de pointeurs : un pointeur de chaîne
occupe une zone mémoire de taille sizeof(char*) */
l = (char**)calloc(11, sizeof(char*));
/* calloc initialise la zone allouée à 0 */
assert(l!=NULL);
for (p=l; (p);p++){
    /* on lit la chaîne dans le buffer ligne*/
    scanf("Entrez une chaîne %s", ligne);
    /* on alloue la zone mémoire juste nécessaire*/
    *p= (char *)malloc(strlen(ligne)+1);
    /*on recopie la chaîne stockée dans le buffer*/
    strcpy(*p, ligne);
}
*p=NULL;
return l;
}
```