Programmation Avancée

Gestion de la mémoire

Mémoire d’un programme

- Deux *espaces* mémoires dispo
  - Pile: (*stack* en Anglais) mémoire à durée limitée.
  - Tas : (*heap* en Anglais) mémoire disponible le temps d’exécution du programme.

Pile

- Petite quantité de mémoire réservée pour exécuter une fonction.
- Disparaît quand la fonction a terminé son exécution.
- Les paramètres d’une fonction sont passées en copie. (i.e. l’intégralité de la donnée est recopiée ; sauf pour les tableaux).
- Déclaration des variables simples.

Tas

- Les données stockées sur le Tas sont accessibles pendant l’exécution du programme.
- On ne communique que l’adresse pas la donnée .
- La déclaration des variables est plus complexes :
  - Déclaration,
  - Allocation.
- Des erreurs peuvent intervenir (p. ex. échec de l’allocation).
- Le programmeur doit **gérer** la mémoire : allocation et **libération.**

Allocation de mémoire

- Pour allouer de la mémoire sur le tas on doit:
  - Déclarer un pointeur `type * variable=NULL;`
  - Utiliser une fonction d'allocation:
    - `malloc` : demande une allocation mémoire.
    - `calloc` : même chose, mais en initialisant à 0.
    - `realloc`: essaie d'agrandir une zone déjà réservée.

Fonction `malloc()`:

- Signature:

  `void * malloc( size_t taille_totale );`

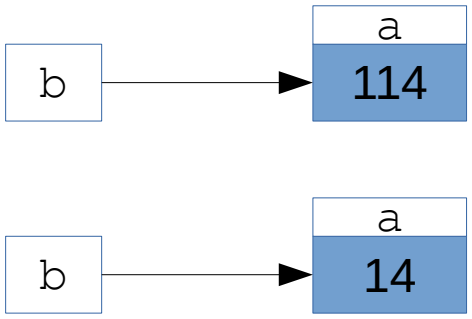
- Définie dans `stdlib.h`
- Paramètre:
  - Le nombre d'**octets** à allouer au total.
  - Un entier positif long(même si c'est indiqué **`size_t`**).
- Retour:
  - Renvoie un pointeur sur la zone allouée.
    `NULL` si ça a échoué.
  - La zone allouée, est un espace contiguë en mémoire (i.e. les adresses se suivent)

Connaître la taille d’un type

- La fonction `sizeof` permet de déterminer la taille d’un type de donné.
- Signature: `size_t sizeof( type) ;`
- Renvoie le nombre d’octets utilisés.
- Fonctionne pour tous les types: `int, float, char, long int`, ... et les `struct` et les `union`.
- Fonctionne également avec les pointeurs: p. ex
  `sizeof (int *)`

Manipulation de pointeurs

```
int a = 114 ;
int * b = NULL ;
b=&a ;
*b=14 ;
```



- La variable `a` prend la valeur 114
- La var. `b` est un pointeur sur un entier
- `b` pointe sur la zone mémoire de `a`.
- On modifie la zone pointée par `b`. Donc `a` vaut maintenant 14

Exemple simple

- Allocation d’un entier:

```
int * p = NULL ;
p=malloc(sizeof(int)) ;
if(p==NULL){return -1 ; }
*p = 25 ;
```

- On réserve seulement ce dont on a besoin.
- Si l’allocation échoue (plus de mémoire ou autre) malloc renvoie NULL. Pour pouvoir utiliser le pointeur on doit s’assurer qu’il est valide.

Exemple différent

- Si dans le code précédent, on change l'allocation par:

  `p=malloc(21) ;`

- Est-ce que le code est toujours fonctionnel ?
- Peut-on modifier la valeur pointée par `p` en faisant

  `*(p)=14 ;` ?

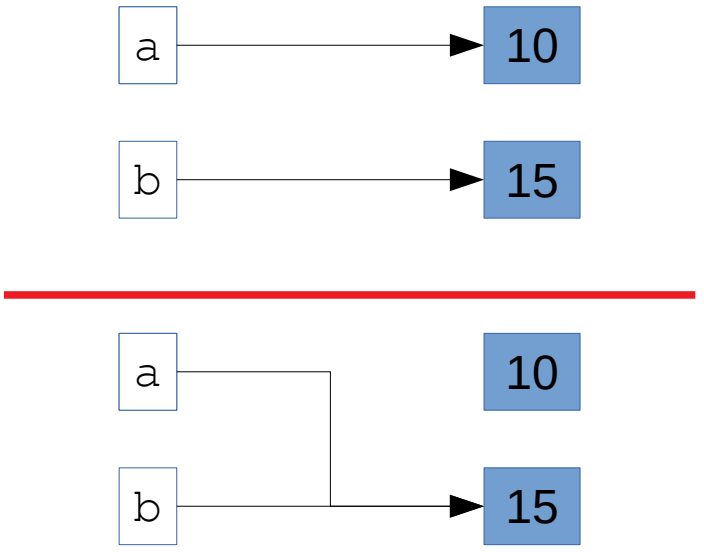
Perte de mémoire

```
int * a =
malloc(sizeof(int));
int * b =
malloc(sizeof(int));

/* tests de vérif des
pointeurs */

*(a)=10 ;
*(b)=15 ;


a=b ;
```



Perte de mémoire (suite)

- Dans l'exemple précédent on a:
  - Deux pointeurs `a` et `b`.
  - Chacun pointe sur sa zone mémoire attribuée par `malloc`.
- Quand on fait `a=b` on perd l'adresse de la zone mémoire pointée initialement par `a`.
- Si cette opération n'est pas une erreur on doit libérer la zone de `a` initialement pointée.

Libération de mémoire

- En C, la libération de la mémoire revient au programmeur.
- Ne pas libérer la mémoire peut engendrer des problèmes au niveau de l’exécution du programme (consommation excessive de ressource; empêcher les autres programmes de fonctionner correctement)

Fonction `free`

- La fonction `free` permet de libérer la mémoire.
- Signature: `void free(void * pointer) ;`
- Le paramètre est l’adresse de base de la zone mémoire pointé: le pointeur.
- Pas besoin d’indiquer la taille.
- Le pointeur doit correspondre à l’adresse attribuée par le `malloc` initial.

Correction

```
int * a = malloc(sizeof(int));
int * b = malloc(sizeof(int));

/* tests de vérif des pointeurs */

*(a)=10 ;
*(b)=15 ;

free(a) ;
a=b ;
```

Les Tableaux:

- Affecter une seule valeur (`int, float` ou autre) sur le tas, n’a pas grand intérêt.
- On peut par contre affecter avec `malloc` un espace mémoire conséquent composé de plusieurs octets.
- On va faire cela pour les tableaux.
- On doit déterminer le nombre d’octets à allouer.

Tableau: Création

```
int taille= 25 ;

int * T = malloc(taille*sizeof(int)) ;

if(T==null){return -1 ;}
```

- On réserve un nombre d’octets égal à la taille d’un entier fois le nombre d’éléments du tableau. Ici 25.

Utilisation du tableau

- Comme `T` est un pointeur d'entiers deux notations possibles:
  - La première avec les crochets. p. ex `T[5]=16 ;`
  - La seconde avec la notation *. p. ex `*(T+5)++ ;`
- Pour la notation `*`, comme `T` est un pointeur, i.e. une adresse de base `T+k`, va opérer un décalage de `k` fois le nombre d'octets d'un entier (en général 4)
  On accèdera donc à la zone mémoire de `T[k]`
- On doit rajouter l'étoile pour accéder à la valeur.

Libération d’un tableau à une dimension:

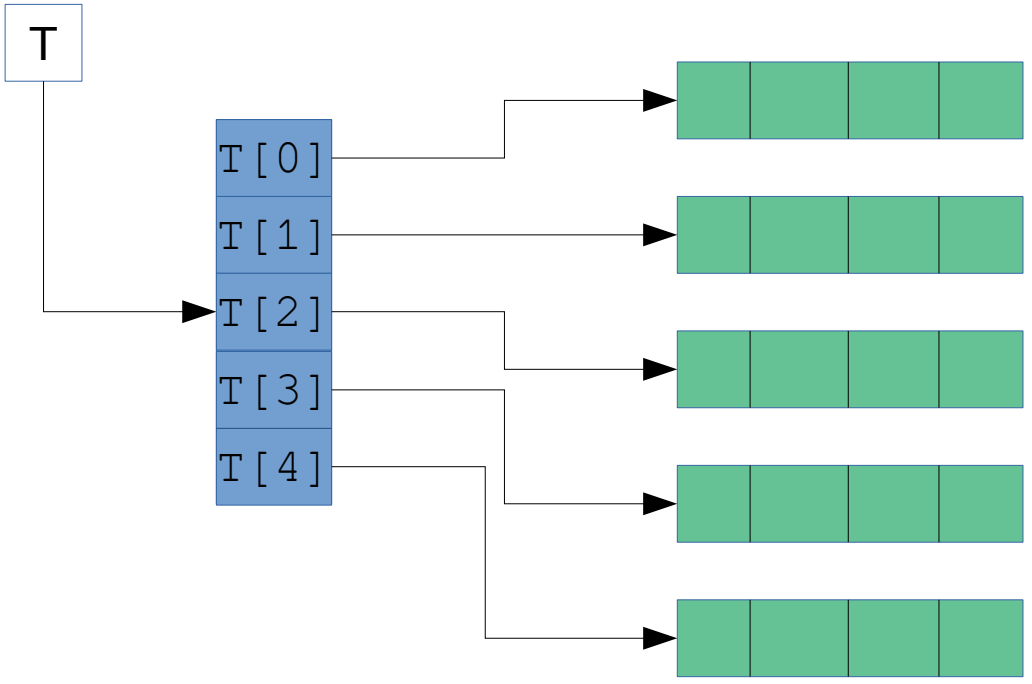
- Pas de difficulté car une seule zone mémoire a été allouée.
- `free(T) ;`

Tableau à 2 (k) dimensions

```
int  i;
int ** T  = NULL ;
T= malloc(5 * sizeof(int *)) ;
/* test de succès */
for(i=0;i<5;i++){
  T[i]=malloc(4 * sizeof(int)) ;
  /* test de succès  */
}
```

Tableau à 2 (k) dimensions (suite)

- Le premier malloc alloue un tableau de pointeurs `int *`
- Chaque case du premier tableau va pointer sur un tableau de 4 entiers. (les `mallocs` effectués dans la boucle for).
-

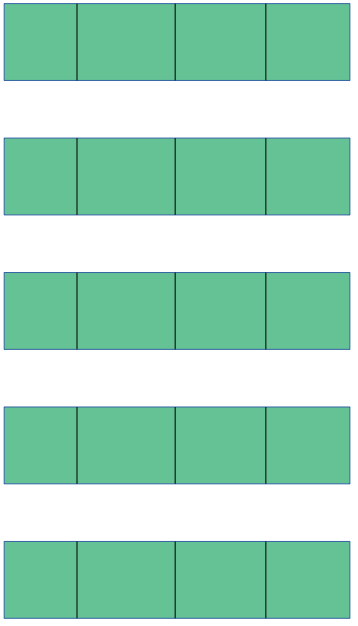

T
T[0]
T[1]
T[2]
T[3]
T[4]

Libération Tableau 2 dimensions

```
free(T);
```

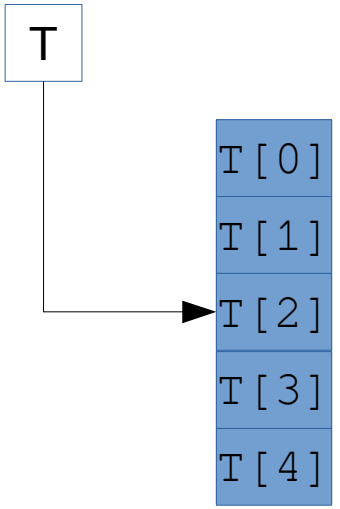
T

- Si on libère `T` en premier
  tous les tableaux de taille 4 existent encore.
- Il faut commencer par les zones qui ne pointent sur aucune autre zone.



Libération Tableau 2 dimensions (suite)

```
for(i=0;i<5;i++)
{
   free(T[i]) ;
}
```



- On libére en premier les derniers tableaux créés.

Libération Tableau 2 dimensions (suite)

- On termine avec le premier tableau créé.

```
free(T);
T= NULL ;
```

- Ainsi aucune espace n’est perdu.
- On peut généraliser à k dimensions.

Notation * pour les tableaux à 2 dim

```
for(i = 0;i<5;i++){
   for(j=0;j<4 ;j++){
      T[i][j]=i*4+j ;
      printf(‘’%d \n’’, *(*(T+i)+j)) ;
      }
      }
```

Allocations dynamique de struct

```
struct coord{ int x; int y ;}

struct coord * point =
malloc(sizeof(struct coord)) ;

…

free(point)
```

Accès aux attributs d’une struct

- Notation *

```
(*point).x=25 ; (*point).y=27 ;
```

- Notation →:

```
point  -> x =42 ; point -> y= 566 ;
```