

Allocation dynamique de mémoire

- ◆ Jusqu'à présent, pour stocker un grand nombre de données nous avons utilisé des tableaux.
 - Mémoire statique
 - Taille du tableau est fixe
 - Nous devons nous assurer de déclarer le tableau d'une taille suffisante.
 - Gaspillage d'espace mémoire
 - Structure non adaptative

Allocation dynamique de mémoire

- ◆ Nous avons brièvement discuté de tableaux dynamique
 - Rappel : les instructions nécessaires pour créer un tableau dynamique de 100 entiers sont :

```
#include<stdlib.h>
int * tab;
tab = (int *)malloc(100 * sizeof(int));
```
- ◆ Dans ce cas, la différence avec un tableau statique est le moment de l'allocation de la mémoire (à l'exécution)

La fonction malloc()

- ◆ La fonction est déclarée dans `stdlib.h` comme suit :

```
void * malloc(unsigned size);
```

 - Le paramètre `size` correspond au nombre d'octets de mémoire à allouer
 - La fonction retourne l'adresse en mémoire du début du bloc de `size` octets
 - Le type de retour étant un pointeur sur `void`, nous devons utiliser un `type cast ()` pour forcer un changement de type à l'affectation
- ◆ Pour libérer la mémoire, il faut utiliser `free(ptr)` ; où `ptr` est l'adresse en mémoire du début du bloc

Structure (rappel)

- ◆ L'utilisation d'une structure permet d'avoir plusieurs champs de données de types différents pour un seul élément
- ◆ Déclarations :

```
struct nom {
    déclarations des champs;
};
```

Ou

```
typedef struct {
    déclarations des champs;
}nom;
```
- ◆ Il est possible d'avoir un tableau dynamique de structures

Listes chaînées

- ◆ Une liste chaînée est une structure de données qui a l'avantage de s'adapter facilement aux besoins en espace mémoire.
- ◆ Schéma d'une liste chaînée

```
liste ----> 20 -----> 15 -----> 10 X
```

 - Un élément d'une liste est une structure qui contient une portion informations et un champs pointeur sur un autre élément du même type de structure
 - La tête de la liste est un pointeur (ici `liste`) qui pointe sur le premier élément de la liste

Listes chaînées

- Chaque élément de la liste sait où est le suivant grâce à son champs qui est un pointeur (souvent nommé `suitant`)
- Le champs pointeur (`suitant`) du dernier élément de la liste vaut `NULL`
- Seul `liste` sait où débute la liste
- ◆ Exemple de déclaration de la structure d'un élément

```
typedef struct Elem
{ int      valeur ;
  struct Elem * suivant ;
} Element ;
```

 - Ici `struct Elem` est nécessaire car à l'intérieur de la déclaration des champs, le type `Element` n'est pas connu

Listes chaînées

- ◆ Déclaration de la liste

```
struct Elem * liste;  
Ou  
Element *liste;  
Ou  
typedef Element * Pointeur;  
Pointeur liste;
```

- liste est un pointeur sur un élément (le premier) de la liste chaînée

Listes chaînées

- ◆ Parcourir tous les éléments d'une liste

```
while(liste != NULL)  
{  
    traitement sur l'élément courant  
  
    liste = liste->suivant;  
}  
  
- while(liste != NULL) peut être remplacé par  
  while(liste)
```

Listes chaînées

- ◆ Dans le cadre de ce cours, nous allons voir 3 sortes de listes chaînées

- LIFO (Last In First Out) qui peut être vue comme une pile
 - Voir chapitre 8 p. 157
- FIFO (First In First Out), une file d'attente
 - Voir chapitre 8 p. 160
- Liste chaînée triée, où les éléments sont en ordre
 - Voir listeTrie.c