Fiche n°17 : Gestion de la mémoire en C

Nous revenons au langage C pour donner quelques compléments concernant la gestion dynamique de la mémoire.

Nous connaissons déjà la fonction **malloc** qui réserve un bloc mémoire et en retourne l'adresse, et nous avons déjà évoqué la fonction **free** qui libère un bloc mémoire dont on lui transmet l'adresse en paramètre.

```
double* p = (double*)malloc(sizeof(double));
*p=10.0;
printf("%lf\n",*p);
free(p);
```

Réserve un bloc permettant de stocker une valeur de type **double**.

Assigne une valeur à la variable dynamique ainsi créée et affiche cette valeur.

Libère le bloc mémoire

La fonction **void* calloc(unsigned int n, unsigned int size)** est une variante qui réserve un bloc mémoire de taille n * size et qui initialise ce bloc avec des 0. Elle est plus spécialement dédiée à la création de tableaux de **n** cellules de taille **size**.

La fonction **void* realloc(void* p, unsigned int size)** Permet de réallouer un bloc plus grand ou plus petit que celui pointé par **p**. Le nouveau bloc n'est pas nécessairement situé à la même adresse que l'ancien. L'ancien est détruit. Les valeurs de l'ancien bloc sont recopiées dans le nouveau, à hauteur de la place disponible. Si le paramètre **p** vaut **NULL**, alors **realloc** devient équivalent à **malloc**.

Typiquement, **realloc** est utilisée pour re dimensionner un tableau dynamique.

```
int* tab;

tab=(int*)calloc(5,sizeof(int));
tab[2]=11;
tab=(int*)realloc(tab,10*sizeof(int));
tab[8]=22;
free(tab);
```

Création d'un tableau dynamique de 5 **int**.

Utilisation du tableau…

Agrandissement du tableau.

Utilisation du tableau agrandi…

Destruction du tableau.

Exemple d'application.

Réalisez une pile d'entiers extensible dont la taille s'adapte à la quantité de valeurs empilées. Réalisez et testez sur machines les fonction permettant de créer une pile, d'empiler une valeur et de dépiler une valeur. Vous devez bien sûr utiliser la fonction **realloc**.

Fiche n°18 : Structures autoréférentes en C

On appelle strucure autoréférente une structure dont l'un des champs est un pointeur une structure de même type. Ce principe est utilisé par exemple pour représenter des listes chaînées, des arbres ou des graphes.

Un exemple : pile d'entiers chaînée.

```c
typedef struct _maillon
{
    int val;
    struct _maillon* suiv;

}maillon;
```

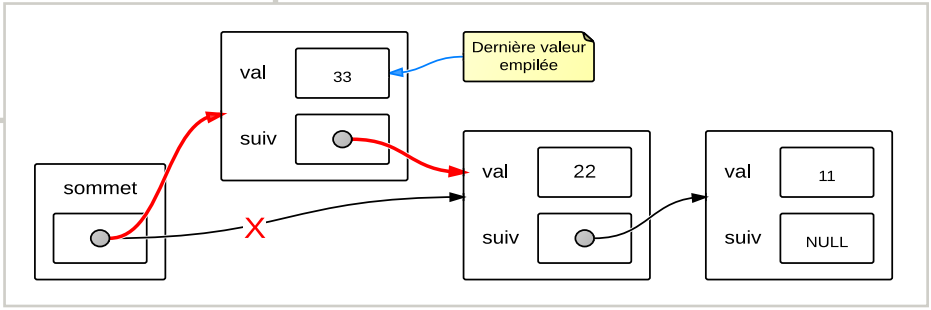
Chaque maillon comporte un entier et un lien vers le maillon suivant.

Une pile comporte un lien vers le premier maillon, appelé sommet.

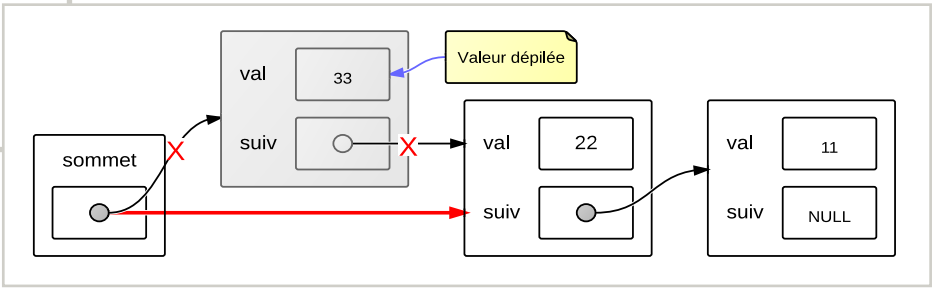
```c
typedef struct
{
    maillon* sommet;

}pileInt;
```

```c
void empile(pileInt* p, int x)
{
  maillon* newm;
  newm = (maillon*)malloc(sizeof(maillon));
  newm->suiv=p->sommet;
  newm->val=x;
  p->sommet=newm;
}
```




```c
int depile(pileInt* p)
{
    maillon* s = p->sommet;
    int val = s->val;
    p->sommet=s->suiv;
    free(s);
    return val;
}
```




```c
void testPile()
{
    pileInt p;
    initPile(&p);
    empile(&p,11);
    empile(&p,22);
    printf("%d %d\n",depile(&p),depile(&p));
}
```

Test consistant à empiler deux valeurs, puis à les dépiler et les afficher.

Fiche de préparation n°5

Vous devez réfléchir à une autre représentation d'une pile d'entiers basée sur le modèle suivant :

Une pile est soit vide, soit constituée d'un sommet et d'une queue, qui est elle même soit une pile vide, soit une constituée…

Il faut donc déterminer comment représenter une pile vide et comment représenter une pile non vide. Une solution consiste à représenter une pile non vide par un pointeur sur une structure comportant les champs appropriés, et une pile vide par un pointeur ayant la valeur NULL. Une autre solution consiste à représenter toute pile, même vide, par une structure comportant notamment un champ particulier dont la valeur indique si la pile set vide ou pas. Tentez d'implanter l'un ou l'autre de ces modèles et de déterminer leurs avantages et inconvénients.