CHAPITRE 2 : ALLOCATION DYNAMIQUE DE LA MÉMOIRE

I. Introduction :

Nous avons insisté plusieurs fois sur le fait que l'on devait déclarer les variables avant de les utiliser: l'un des rôles de cette déclaration est précisément de permettre au compilateur d'allouer les zones mémoire nécessaires pour le stockage des données. Ceci impose au programmeur de connaître le nombre de données nécessaires au programme : dans le cas des tableaux et des matrices, nous avons ainsi souvent déclaré des tableaux d'une taille maximale fixée, et utilisé une partie seulement de ce tableau, mais il est totalement impossible, à moins de modifier la valeur de la taille du tableau et donc de recompiler le programme, de traiter des données de taille supérieure !

Il est de nombreux cas où le nombre de données est inconnu a priori, et où il n'est pas non plus possible de réserver une place mémoire maximale fixée. Le langage C offre au programmeur la possibilité d'allouer dynamiquement et à la demande, des emplacements en mémoire pour stocker les données.

II. LES OUTILS DE BASE DE LA GESTION DYNAMIQUE: malloc ET free :

Commençons par étudier les deux fonctions les plus classiques de gestion dynamique de la mémoire, à savoir **malloc** et **free**.

II-1. La fonction malloc :

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection *****, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée **NULL** définie dans **stdio.h** ou **malloc.h**. En général, cette constante vaut 0. Le test **p== NULL** permet de savoir si le pointeur p pointe vers un objet.

On peut initialiser un pointeur p par une affectation sur p. Par exemple, on peut affecter à p l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à *p. Mais pour cela, il faut d'abord réserver à *p un espace-mémoire de taille adéquate. L'adresse de cet espace-mémoire sera la valeur de p. Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle allocation dynamique. Elle se fait en C par la fonction **malloc** de la librairie standard **stdlib.h** ou ***alloc.h***. Sa syntaxe est :

Syntaxe :

void * malloc (*nombre-octets*)

Cette fonction demande au système d'exploitation un bloc de taille *nombre-octets* (en octets) et renvoie un pointeur vers l'adresse du bloc alloué. S'il se produit une erreur, typiquement il n'y a plus de mémoire disponible, cette fonction renvoie la valeur **NULL**. La fonction retourne une adresse non typée **(void *)**, ceci n'est pas un problème, en effet, le type void * est compatible avec tous les pointeurs, par conséquent une conversion de type à l'aide d'un **cast** n'est pas obligatoire. Cependant, pour des raisons historiques, il était d'usage d'effectuer cette conversion sur le retour de la fonction dans le type de la variable qui stockera cette adresse. Ainsi, si on voulait allouer dynamiquement un entier, on faisait une conversion de type vers un pointeur d'entier.

L'argument ***nombre-octets*** est souvent donné à l'aide de la fonction **sizeof()** qui renvoie le nombre d'octets utilisés pour stocker un objet.

Exemple 1:

```
int * p=NULL;
p = (int*)malloc(sizeof(int));

  /* en principe sizeof(int)=4 */

/* On aurait pu écrire également
p = (int*)malloc(4);*/
```

Exemple 2:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main()
{
        int i = 3;
        int *p;
        printf("valeur de p avant initialisation = %ld\n",p);
        p = (int*)malloc(sizeof(int));
        printf("valeur de p après initialisation = %ld\n",p);
        if (p !=NULL)
        {*p = i;
                printf("valeur de *p = %d\n",*p);
        }
        else
                printf("Memoire insuffisante \n");

  getch();
}
```

Définit un pointeur p sur un objet *p de type int, et affecte à *p la valeur de la variable i.
résultat du programme :
valeur de p avant initialisation = 0
valeur de p après initialisation = 5368711424
valeur de *p = 3

La fonction **malloc** permet également d'allouer un espace pour plusieurs objets contigus en mémoire (Les tableaux). On peut écrire par exemple pour un pointeur p de type int* :

```
p =(int*) malloc(nombreElements * sizeof(int));
```

Remarque:

nombreElements doit représenter le nombre de cases de notre tableau, et donc doit être un **entier positif** (différent de zéro).

Exemple 3:

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int  i = 3;
int  j = 6;
int *p;
p = (int*)malloc(2*sizeof(int));
if (p )                    /* ou if (p !=NULL )   */
{
        *p = i;
        *(p+1) = j;
        printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
else
         printf("Allocation impossible  \n");

}
```

Définit un pointeur p sur un objet *p de type int, réserve 2 places d'objets de type int à l'adresse p (8 octets).
résultat du programme :
*p = 5368711424 *p = 3 p+1 = 5368711428 *(p+1) = 6*

Exemple 4 (Pointeurs et structures):

```
#include <stdio.h>
#include <stdlib.h>
#define UE_CARD 6
typedef struct {
 char *nom, *prenom;
 short age;
 short notes[UE_CARD];
} Eleve;

int main() {
 Eleve *promo1 = NULL;
int count1 = 0;
printf("Entrer le nombre d'élèves: ");
scanf("%d", &count1);
promo1 = (Eleve*) malloc(count1 * sizeof(Eleve));

... lire les données ...

 if (promo1) { /* afficher les données */
Eleve* p;
for (p = promo1; p < promo1 + count1; p++) {
printf("Eleve %s %s\n", p->nom, p->prenom);
printf("age : %d \n", p->age);
.....
}
return 0;
}
```

Exemple 5 (Pointeurs et chaînes de caractères):

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
main()
{               int i;
                char *chaine1, chaine2, *res, *p;
                chaine1 = "chaine ";
                chaine2 = "de caracteres";
                res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
                if (res)
                {
                p =res;
                for (i = 0; i < strlen(chaine1); i++)
                                        *p++ = chaine1[i];
                for (i = 0; i < strlen(chaine2); i++)
                                        *p++ = chaine2[i];
                printf("%s\n",res);

                }
                else
                 printf("Allocation impossible  \n");



}
```

II-2. La fonction free :

L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin. Le rôle de la fonction **free** est de libérer un emplacement préalablement alloué.

Syntaxe :

free (*pointeur***)**

Exemple :

```
#include <stdio.h>
#include <alloc.h>
int main(void)
{
char * adr1, * adr2, * adr3 ;
adr1 =( char *) malloc (100) ;
printf ("allocation de 100 octets en %p\n", adr1) ;
adr2 = ( char *) malloc (50) ;
printf ("allocation de 50 octets en %p\n", adr2) ;
free (adr1) ;
printf ("libération de 100 octets en %p\n", adr1) ;
adr3 = ( char *) malloc (40) ;
printf ("allocation de 40 octets en %p\n", adr3) ;
return 0 ;
}
```

résultat du programme :
allocation de 100 octets en 06AC
allocation de 50 octets en 0714
libération de 100 octets en 06AC
allocation de 40 octets en 06E8

Remarque :

Vous notez que la dernière allocation a pu se faire dans l'espace libéré par le précédent appel de free.

III. LES AUTRES FONCTIONS DE LA GESTION DYNAMIQUE: calloc ET realloc :

III-1. La fonction calloc :

La fonction **calloc** de la librairie **stdlib.h** a le même rôle que la fonction **malloc** mais elle initialise en plus l'objet pointé *p à zéro. Sa syntaxe est :

Syntaxe :

void * calloc (nombre-objets , taille-objets)

Exemple :

```
p =(int*)calloc(N,sizeof(int));

/*  est strictement équivalente à */
                    p = (int*)malloc(N * sizeof(int));
                              for (i = 0; i < N; i++)
                                        *(p+i) = 0;
```

III-2. La fonction realloc :

Il arrive que l'on ait besoin de modifier la taille d'un bloc mémoire que nous avions alloué grâce à la fonction **malloc** ou **calloc**. Pour cela, il nous faudra utiliser la fonction **realloc**, dont la syntaxe est:

Syntaxe :

void * realloc(pointeur , nouvelle_taille);

Le premier paramètre que prend cette fonction correspond à un pointeur vers un espace mémoire qui avait été dynamiquement alloué au préalable, via la fonction **malloc** ou une fonction équivalente. Si ce pointeur est nul, la fonction **realloc** agira de la même manière que la fonction **malloc**, que nous avons étudié plus haut.

Le second paramètre correspond à la nouvelle taille que nous souhaitons pour notre bloc mémoire. Celle-ci peut être inférieure, ou supérieure, selon vos besoins, à la taille d'origine.

En cas de succès, la fonction **realloc** retourne un pointeur sur la nouvelle zone mémoire, qui peut ou non être la même que celle qui était déjà à votre disposition, selon l'état dans lequel la mémoire se trouvait. En cas d'échec à la réallocation, la fonction retourne **NULL** ; notez que, dans ce cas, vous ne devez pas considérer que les données pointées par le pointeur que vous aviez passé en premier paramètre soient toujours valides !

Remarque :

- Si la " nouvelle_taille " est supérieure à l'ancienne, il y a conservation des données.
- Si la " nouvelle_taille " est inférieure à l'ancienne, il y a conservation des données, jusqu'à la nouvelle taille .