Chapitre 15

Allocation dynamique de Mémoire

Nous allons à présent aborder un chapitre qui peut paraître quelque peu déroutant au départ ; mais ce que nous y apprendrons se révèle rapidement être extrêmement utile, et important. Je ne peux donc que vous inciter à persévérer si vous ne saisissez pas tout dès la première lecture, car il est rare de ne pas être amené à rencontrer des "allocations dynamiques de mémoire" dans les programmes que l'on trouve sur le net, ou que l'on écrit, à partir du moment où l'on arrive à des programmes de taille non négligeable.

I:\ Allocation Dynamique de Mémoire ?

Il est relativement fréquent, lorsque l'on commence à développer des programmes réellement interactifs, d'avoir besoin, par moment, de disposer d'un espace mémoire dont la taille n'est soit pas connue à l'écriture du programme, soit trop grande pour pouvoir être allouée sur la pile.
Par exemple, puisque la pile ne fait que environ 16Ko, il n'est pas possible de déclarer un tableau de la manière qui suit, dans le corps d'une fonctions :

```
unsigned long tab[5000];
```

En effet, un `unsigned long` occupe 4 octets en mémoire ; cette déclaration demande 5000*4, soit 20000 octets... sur une pile qui n'en fait que 16000 et des miettes.
Cette déclaration posera donc, comme vous pouvez vous y attendre, des problèmes lors de l'exécution de votre programme.

La solution à ce problème est de demander de la mémoire ailleurs que sur la pile, c'est-à-dire, sur ce qu'on appelle, en programmation, le tas ("Heap" en anglais, par opposition à "Stack", la pile).
Sur nos calculatrice, la taille du tas est au maximum d'environ 180Ko, lorsque toute la mémoire RAM est libre ; naturellement, si l'utilisateur a laissé des variables en mémoire RAM sans les archiver, moins de mémoire sera disponible ; autrement dit, une allocation dynamique de mémoire peut échouer ; il vous faudra donc **toujours**, lorsque vous en effectuerez une, vérifier si elle a réussi.

D'autre part, la mémoire RAM est découpée en blocs de 64Ko ; il n'est donc pas possible d'allouer plus de 64Ko (moins quelques octets) à la fois.

Effectuer une allocation dynamique de mémoire, c'est demander au système de vous réserver un bloc de mémoire, que vous pourrez utiliser à votre convenance.
Cela dit, c'est à vous d'indiquer, avant la fin de votre programme, que vous souhaitez libérer ce bloc mémoire. Dans le cas contraire, il restera marqué comme réservé, et le système d'exploitation de la TI ne pourra plus l'utiliser ; au final, si vous allouez beaucoup de mémoire sans jamais la libérer, votre TI n'en n'aura plus de libre... ni pour vos programmes, ni pour son propriétaire... Et, dans ce cas de figure, la seule solution permettant de récupérer la mémoire perdue est d'effectuer un reset de la calculatrice.
Vous vous doutez bien que cette solution n'est pas appréciable pour les utilisateurs de votre programme... qui ne l'utiliseront probablement pas longtemps, s'il cause de telles fuites de mémoire.
Donc, lorsque vous allouez dynamiquement de la mémoire, pensez à toujours la libérer lorsque vous n'en n'avez plus besoin !

II:\ Comment faire, sur TI

Maintenant que nous avons vu en quoi l'allocation dynamique de mémoire consiste, ainsi qu'à quoi elle sert, nous allons étudier les différentes fonctions mises à notre disposition pour travailler avec le tas.

A: Allocation Mémoire

Tout d'abord, allouons de la mémoire.
Pour cela, nous utilisons la fonction `malloc`, dont le prototype est le suivant :

```
void *malloc(unsigned long taille);
```

Cette fonction prend en paramètre la quantité de mémoire, en octets, que l'on souhaite allouer, et retourne, en cas de succès, un pointeur vers la zone mémoire qui nous est nouvellement réservée. En cas d'échec, c'est un pointeur `NULL` qui nous est renvoyé.
La fonction `malloc` est un alias, conforme à la norme ANSI-C, de la fonction `HeapAllocPtr`, dont le nom provient du TIOS. Naturellement, cette fonction est définie exactement de la même manière que la première :

```
void *HeapAllocPtr(unsigned long taille);
```

Généralement, j'ai tendance à préférer la fonction `malloc`, du fait que c'est une fonction standard de la bibliothèque C, sous tous les compilateurs non préhistoriques. Cela dit, d'autres programmeurs, en particulier ceux ayant commencé la programmation sur TI par le langage d'Assembleur, préférent `HeapAllocPtr`. Encore une fois, c'est à vous de choisir selon vos préférences.

Je me permet d'insister sur la nécessité de vérifier que l'allocation a réussi : si la fonction d'allocation a renvoyé `NULL`, le bloc mémoire que vous souhaitiez réserver ne vous a pas été alloué ; il ne faut pas, en ce cas, travailler avec le pointeur retourné, sous peine de plantage (écriture vers l'adresse nulle, ou vers une zone mémoire non définie).

Par exemple, pour allouer un tableau de 5000 `unsigned long`, comme nous souhaitions le faire au début de ce chapître :

```
unsigned long *tab;

tab = malloc(5000*sizeof(unsigned long));
if(tab != NULL)
{
    // Utilisation de l'espace mémoire alloué
    // ...
    // /!\ Penser à libérer la mémoire !!!
}
else
{
    // Echec de l'allocation mémoire
}
```

Notez l'utilisation de `5000*sizeof(unsigned long)` : malloc attend une taille en octets... Puisque nous voulons allouer l'espace mémoire nécessaire pour 5000 `unsigned long`, il nous faut multiplier 5000 par la taille d'un `unsigned long`. Et c'est cette taille qui est retournée par `5000*sizeof(unsigned long)`.
N'oubliez pas ceci, ou il sera fréquent que nous allouiez moins de mémoire que ce que vous pensiez... et que cela soit à l'origine de plantages de votre programme.

B: Réallocation

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

```
void *realloc(void *pointeur, unsigned long 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 mùmoire 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 !

A titre d'illustration, voici un petit exemple de réallocation de l'espace mémoire que nous avons alloué un peu plus haut :

```
tab = realloc(tab, 6000*sizeof(unsigned long));
if(tab != NULL)
{
     // Utilisation de l'espace mémoire ré-alloué
     // ...
     // /!\ Penser à libérer la mémoire !!!
}
else
{
     // Echec de la ré-allocation mémoire
}
```

La remarque que j'ai fait pour `malloc` au sujet de `sizeof(unsigned long)` est valide ici aussi, bien entendu, puisque la taille attendue par `realloc` est exprimée en octets.

C: Libération de mémoire allouée

J'ai déjà parlé plusieurs fois de l'importance qui réside dans le fait de bien libérer l'espace mémoire que vous avez alloué... Il serait peut-être temps d'étudier comment cela se fait.
Pour libérer un espace mémoire que vous aviez demandé précédemment à l'aide de la fonction `malloc`, il convient d'utiliser la fonction `free`, dont le prototype est le suivant :

```
void free(void *pointeur);
```

L'emploi de cette fonction est des plus simple : elle prend en paramètre un pointeur sur l'espace mémoire à libérer, pointeur qui correspond à la valeur de retour de `malloc`, et ne retourne rien.
De la même façon que `HeapAllocPtr` est un alias de `malloc`, la fonction `free` est un alias de la fonction `HeapFreePtr`.

```
void HeapFreePtr(void *pointeur);
```

Ici encore, choisir entre l'une ou l'autre de ces deux fonctions est une histoire de goûts et d'habitudes... Personnellement, je préfère la fonction `free`, du fait qu'il s'agit d'une fonction ANSI-C ; mais libre à vous d'agir autrement.

Je me permet d'attirer votre attention sur le fait qu'il est nécessaire que le paramètre que vous passez à la fonction `free` soit valide, et corresponde bien au pointeur qui avait été retourné par une fonction d'allocation mémoire.
Si le pointeur que vous passez en paramètre n'est pas valide, il est fort probable que votre programme plantera.

Et voici un petit exemple, même si je doute de son utilité :

```
unsigned long *tab = malloc(5000*sizeof(unsigned long));
if(tab != NULL)
{
	// Utilisation de l'espace mémoire alloué
	// ...
	free(tab);
}
else
{
	// Echec de l'allocation mémoire
}
```

Comme précédemment, nous allouons un espace mémoire ; et si l'allocation a réussi, nous libérons le bloc mémoire que nous avions obtenu.

III:\ Exemple

Pour finir, voici un petit exemple de programme au sein duquel nous réalisons une allocation dynamique de mémoire :

```
#include <tigcclib.h>

void _main(void)
{
    short nombre_elements = random(10)+1;
    clrscr();

    unsigned long *tab = malloc(nombre_elements*sizeof(unsigned long));
    if(tab != NULL)
    {
        short i;
        for(i=0 ; i<nombre_elements ; i++)
        { // Remplissage du tableau...
            tab[i] = random(100000);
        }

        printf("%d éléments :\n", nombre_elements);
        for(i=nombre_elements-1 ; i>=0 ; i--)
        { // Affichage du tableau (en ordre inverse)...
            printf("%lu\n", tab[i]);
        }

        free(tab);
    }
    else {printf("Echec de l'allocation mémoire\n");}

    ngetchx();
}
```

Exemple Complet

Tout d'abord, nous utilisons la fonction `random`, qui retourne un nombre choisi aléatoirement entre 0 et la valeur passée en argument (exclue), de manière à obtenir un nombre valant entre 1 et 10 compris.
Ensuite, nous allouons une zone mémoire pouvant contenir ce nombre d'`unsigned long`.
Après quoi, nous initialisons chacune des cases de ce tableau à une valeur aléatoire, choisie entre 0 et 99999 compris, puis nous parcourons ce tableau en sens inverse, dans le but d'afficher les valeurs qu'il contient à présent.
Et enfin, nous libérons l'espace mémoire que nous avions alloué.
Naturellement, nous n'avons effectué nos manipulations sur l'espace mémoire alloéu uniquement si l'allocation mémoire n'avait pas échoué.

Pour cet exemple comme pour celui que nous avions étudié il y a quelques chapîtres de ça, il aurait été possible de se passer d'un tableau... Mais, après tout... c'était le but de notre exemple.

Notons, qu'il existe de nombreuses autres fonctions permettant de travailler avec la mémoire ; on peut par exemple déterminer combien de mémoire est disponible, verrouiller des blocs en mémoire, et pas mal d'autres choses encore.
Cela dit, ces fonctionnalités ne sont pas ANSI, et il n'est pas nécessaire, à mon avis, de les connaître, sauf si l'on souhaite effectuer des manipulations bien particulières ; si le besoin s'en fait sentir plus loin au cours de ce tutorial, nous étudierons certaines de ces fonctions.
Bien entendu, en plus de cela, rien ne vous empêche de consulter la documentation du fichier `alloc.h`.