Université Sultan Moulay Slimane

Faculté Polydisciplinaire **Khouribga**

Sciences Mathématiques et Informatique

Structures de Données

Chapitre 2 : Gestion dynamique de mémoire

Pr. Ibtissam Bakkouri

i.bakkouri@usms.ma

Année Universitaire : **2022/2023**

Plan

Allocation dynamique

La primitive d'allocation réclame des blocs de mémoire dans la zone mémoire appelée le tas et la met à la disposition du programme.

Cette opération est dite allocation dynamique parce que l'espace mémoire est réservé lors de l'exécution du programme et non lors de la compilation. Par analogie, les entités créées de cette façon sont dites dynamiques.

L'allocation dynamique de mémoire permet la réservation d'un espace mémoire pour son programme au moment de son exécution. Ceci est à mettre en opposition avec l'allocation statique de mémoire.

Libération dynamique

Un des intérêts de l'allocation dynamique est de libérer l'espace mémoire alloué lorsqu'il n'est plus nécessaire. Cet espace pourra donc être réaffecté ultérieurement lors de nouvelles demandes dynamiques.

Lorsque celle-ci est devenue inutile, la primitive de libération permet de détruire l'entité dynamique : le mémoire alloué est libéré et redonné au système.

Chaque allocation avec doit impérativement être libérée (détruite) avec sous peine de créer **une fuite de mémoire**.

Fuite de mémoire

La fuite de mémoire est une zone mémoire qui a été allouée dans le tas par un programme qui a omis de la désallouer avant de se terminer. Cela rend la zone inaccessible à toute application (y compris le système d'exploitation) jusqu'au redémarrage du système. Si ce phénomène se produit trop fréquemment la mémoire se remplit de fuites et le système finit par tomber faute de mémoire.

Fonction malloc

La bibliothèque standard fournit trois fonctions vous permettant d'allouer de la mémoire: **malloc()**, **calloc()** et **realloc()** et une vous permettant de la libérer: **free()**. Ces quatre fonctions sont déclarées dans l'en-tête <**stdlib.h**>.

Syntaxe de malloc() :

```
void *malloc(size_t taille);
```

La fonction malloc() vous permet d'allouer un objet de la taille fournie en argument (qui représente un nombre de multiplets) et retourne l'adresse de cet objet sous la forme d'un pointeur générique. En cas d'échec de l'allocation, elle retourne un pointeur nul.

Allocation d'une variable

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{ int *p = malloc(sizeof(int));
if (p == NULL)
{ printf("Échec de l'allocation");
return EXIT_FAILURE;
} *p = 10;
printf("%d", *p);
return 0; }
```

Nous réservons un objet de la taille d'un int, nous y stockons ensuite la valeur dix et l'affichons. Pour cela, nous utilisons un pointeur sur int qui va se voir affecter l'adresse de l'objet ainsi alloué et qui va nous permettre de le manipuler comme nous le ferions s'il référençait une variable de type int.

Allocation d'un tableau

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
void main() {
int *p = malloc(sizeof(int) * 10);
if (p == NULL)
{ printf("Échec de l'allocation");
return EXIT_FAILURE;
} for (unsigned i = 0; i < 10; ++i)
{ p[i] = i * 10;
printf("p[%u] = %d", i, p[i]); } }
```

Nous réservons un objet de la taille d'un int, nous y stockons ensuite la valeur dix et l'affichons. Pour cela, nous utilisons un pointeur sur int qui va se voir affecter l'adresse de l'objet ainsi alloué et qui va nous permettre de le manipuler comme nous le ferions s'il référençait une variable de type int.

Fonction calloc

La fonction **calloc()** attend deux arguments: le nombre d'éléments à allouer et la taille de chacun de ces éléments.

void *calloc(size_t nombre, size_t taille);

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
#define INT_NUMBER 10
int main() {
int i; int * pointer = (int *) calloc( INT_NUMBER, sizeof(int) );
for ( i=0; i<INT_NUMBER-1; i++ ) { pointer[i] = i; }
for (i=0; i<INT_NUMBER; i++ ) { printf( "%d ", pointer[i] ); }
free( pointer);
return 0; }
```

Fonction realloc

La fonction **realloc()** libère un bloc de mémoire précédemment alloué, en réserve un nouveau de la taille demandée et copie le contenu de l'ancien objet dans le nouveau.

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(void) { int *p = malloc(sizeof(int[10]));
if (p == NULL) { printf("Échec de l'allocation"); return EXIT_FAILURE;
}
for (unsigned i = 0; i < 10; ++i) p[i] = i * 10;
int *tmp = realloc(p, sizeof(int[20]));
```

Fonction realloc

```
if (tmp == NULL) { free(p); printf("Échec de l'allocation");
return EXIT_FAILURE; }
p = tmp;
for (unsigned i = 10; i < 20; ++i) p[i] = i * 10;
for (unsigned i = 0; i < 20; ++i) printf("p[%u] = %d", i, p[i]);
free(p); return 0; }
```

Remarquez que nous avons utilisé une autre variable, tmp, pour vérifier le retour de la fonction realloc(). En effet, si nous avions procéder comme ceci.

p = realloc(p, sizeof(int[20]));

Fonction free

La libération de mémoire est extrêmement simple: elle se fait via la fonction **free()**. Voici sa signature:

void free(void * bloc);

La fonction libère un bloc mémoire précédemment alloué via les fonctions **malloc()** ou **calloc()**. La libération de mémoire ne devrait pas poser de problème.

Tableaux à un indice

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
void main() {
int *p = malloc(sizeof(int) * 10);
if (p == NULL)
{ printf("Échec de l'allocation");
return EXIT_FAILURE;
} for (unsigned i = 0; i < 10; ++i)
{ p[i] = i * 10;
printf("p[%u] = %d", i, p[i]); } }
```

Nous réservons un objet de la taille d'un int, nous y stockons ensuite la valeur dix et l'affichons. Pour cela, nous utilisons un pointeur sur int qui va se voir affecter l'adresse de l'objet ainsi alloué et qui va nous permettre de le manipuler comme nous le ferions s'il référençait une variable de type int.

Tableaux à deux indices

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{ int *p = malloc(sizeof(int[3][3]));
if (p == NULL) { printf("Échec de l'allocation");
return EXIT_FAILURE; }
for (unsigned i = 0; i < 3; ++i) for (unsigned j
= 0; j < 3; ++j) { p[(i * 3) + j] = (i * 3) + j;
printf("p[%u][%u] = %d", i, j, p[(i * 3) + j]); }
free(p); return 0; }
```

Comme pour un tableau simple, il vous est possible d'allouer un bloc de mémoire dont la taille correspond à la multiplication des longueurs de chaque dimension, elle-même multipliée par la taille d'un élément.

Travail à Rendre

Exercice 1 :

Écrire une fonction qui alloue la mémoire d'un tableau. Ensuite, écrire une fonction qui libère ce tableau. Afficher ce tableau pour tester vos fonctions.

Exercice 2 :

Écrire une fonction qui alloue la mémoire d'une matrice. Ensuite, écrire une fonction qui libère cette matrice. Afficher cette matrice pour tester vos fonctions.