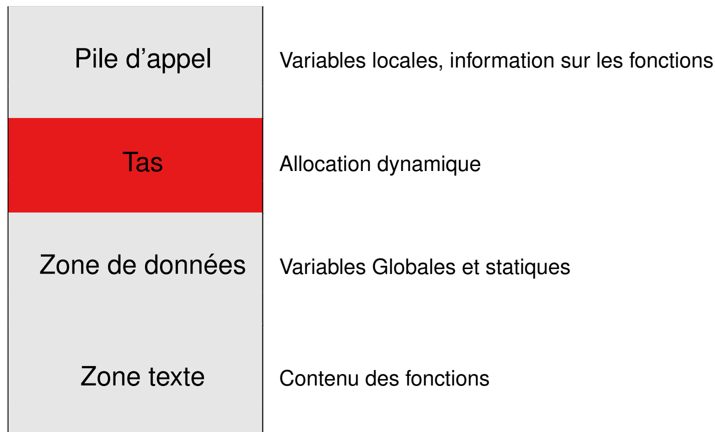
Cours de Programmation Impérative
Allocation dynamique

Julien David

A101 - david@lipn.univ-paris13.fr

Previously on "Programmation Impérative"

Épisode sur les zones de mémoire



Épisode sur les pointeurs

La puissance des pointeurs

- Ils permettent d'accéder à n'**importe quel** bloc de mémoire, situé dans n'**importe quelle** zone de mémoire du programme.

Or jusqu'ici. . .

- . . .on s'en servait pour se balader dans la pile

Avec les notions actuelles

On ne peut pas (pour l'instant)

- créer/réserver de l'espace mémoire avec une fonction.
- faire varier la taille d'un tableau
- définir des structures contenant des tableaux dont la taille n'est pas précisée.

Réserver de l’espace mémoire avec une fonction

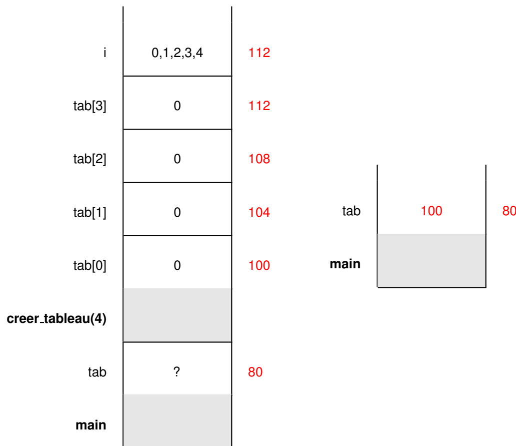
```c
int * creer_tableau(int n){
  int tab[n];
  int i;
  for(i=0;i<n;i++)
    tab[i]=0;
  return tab;
}

int main(){
  int * tab=creer_tableau(4);
  return EXIT_SUCCESS;
}
```

Reserver de l'espace mémoire avec une fonction

```c
int * creer_tableau(int n){
  int tab[n];
  int i;
  for(i=0;i<n;i++)
    tab[i]=0;
  return tab;
}

int main(){
  int * tab=creer_tableau(4);
  return EXIT_SUCCESS;
}
```



Reserver de l'espace mémoire avec une fonction

Pour réserver de l'espace mémoire

- On a besoin d'utiliser une autre zone de la mémoire.

Définir des structures de taille variable

Actuellement, pour stocker plusieurs données de même type, on utilise des tableaux.

- On ne peut pas faire varier leur taille au cours du programme.
- Lorsqu'une structure contient un tableau, on est obligé de définir sa taille à l'avance.

Faire varier la longueur d’un tableau

```
1 int main(){
2   char chaine[10];
3   scanf("%s",chaine);
4   printf("%s\n",chaine);
5   return EXIT_SUCCESS;
6 }
```

- Si l’utilisateur rentre une chaine trop grande, il y aura un débordement.
- L’idée se généralise à n’importe qu’elle structure, on ne peut pour l’instant avoir que des objets de taille fixe.

L'allocation dynamique

Notion Fondamentale : l'allocation dynamique

L'allocation dynamique ou allocation dynamique *sur le tas* consiste à allouer de la mémoire dans le *tas* (la zone de mémoire).

Les avantages

- On peut allouer ou libérer la mémoire à tout moment.
- On peut accéder à la zone de mémoire à partir de n'importe qu'elle fonction.
- On peut décider de "modifier la taille" d'une zone de mémoire.

Les inconvénients

- Contrairement à l'allocation dans la pile, la mémoire n'est pas libérée automatiquement.
- L'allocation mémoire est un appel à une fonction système, souvent coûteuse en temps.

L'allocation dynamique

Notion Fondamentale : l'allocation dynamique

L'allocation dynamique ou allocation dynamique *sur le tas* consiste à allouer de la mémoire dans le *tas* (la zone de mémoire).

Les avantages

- On peut allouer ou libérer la mémoire à tout moment.
- On peut accéder à la zone de mémoire à partir de n'importe qu'elle fonction.
- On peut décider de "modifier la taille" d'une zone de mémoire.

Les inconvénients

- Contrairement à l'allocation dans la pile, la mémoire n'est pas libérée automatiquement.
- L'allocation mémoire est un appel à une fonction système, souvent coûteuse en temps.

L’allocation dynamique

Notion Fondamentale : l’allocation dynamique

L’allocation dynamique ou allocation dynamique *sur le tas* consiste à allouer de la mémoire dans le *tas* (la zone de mémoire).

Les avantages

- On peut allouer ou libérer la mémoire à tout moment.
- On peut accéder à la zone de mémoire à partir de n’importe qu’elle fonction.
- On peut décider de ”modifier la taille” d’une zone de mémoire.

Les inconvénients

- Contrairement à l’allocation dans la pile, la mémoire n’est pas libérée automatiquement.
- L’allocation mémoire est un appel à une fonction système, souvent coûteuse en temps.

Les fonctions d'allocations

L'allocation

La fonction `void * malloc(size_t size);`

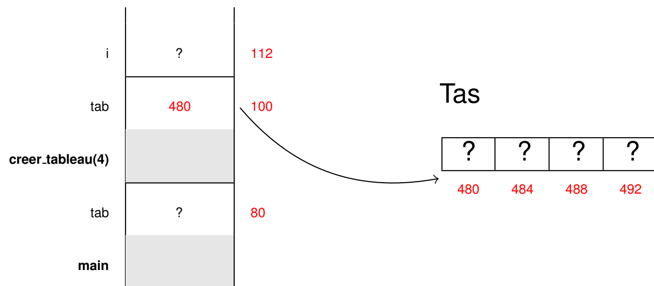
- prend en entrée la taille du bloc de mémoire que l'on souhaite obtenir,
- alloue le bloc de mémoire dans le tas,
- renvoie l'adresse du bloc de mémoire.

Syntaxe d'un appel à la fonction `malloc`

```
malloc(taille case * nombre cases);
```

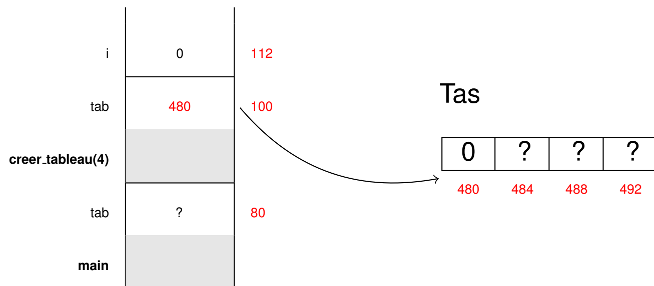
Les fonctions d'allocations

```
1  int * creer_tableau(int n){
2    int *tab=malloc(sizeof(int)*n);
3    int i;
4    for(i=0;i<n,i++)
5      tab[i]=0;
6    return tab;
7  }
8
9  int main(){
0    int * tab=creer_tableau(4);
1    return EXIT_SUCCESS;
2  }
```



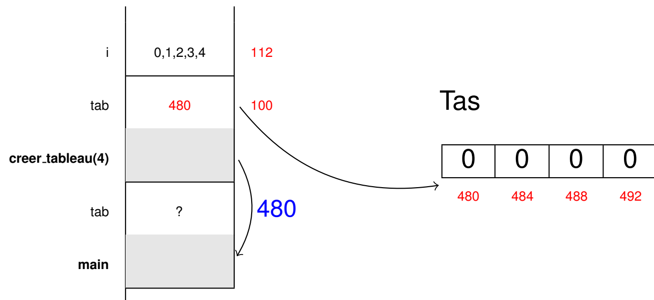
Les fonctions d'allocations

```
1  int * creer_tableau(int n){
2    int *tab=malloc(sizeof(int)*n);
3    int i;
4    for(i=0;i<n,i++)
5      tab[i]=0;
6    return tab;
7  }
8
9  int main(){
0    int * tab=creer_tableau(4);
1    return EXIT_SUCCESS;
2  }
```



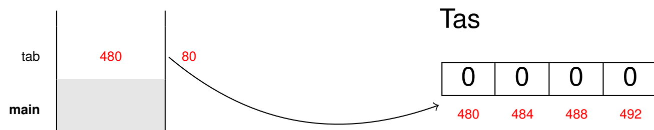
Les fonctions d'allocations

```
1  int * creer_tableau(int n){
2    int *tab=malloc(sizeof(int)*n);
3    int i;
4    for(i=0;i<n,i++)
5      tab[i]=0;
6    return tab;
7  }
8
9  int main(){
0    int * tab=creer_tableau(4);
1    return EXIT_SUCCESS;
2  }
```



Les fonctions d'allocations

```
1 int * creer_tableau(int n){
2   int *tab=malloc(sizeof(int)*n);
3   int i;
4   for(i=0;i<n,i++)
5     tab[i]=0;
6   return tab;
7 }
8
9 int main(){
0   int * tab=creer_tableau(4);
1   return EXIT_SUCCESS;
2 }
```



La fonction `creer_tableau` a une complexité $\Theta(n)$ en temps et en espace.

Les fonctions d'allocations

L'allocation

La fonction `void *calloc(size_t nmemb, size_t size);`

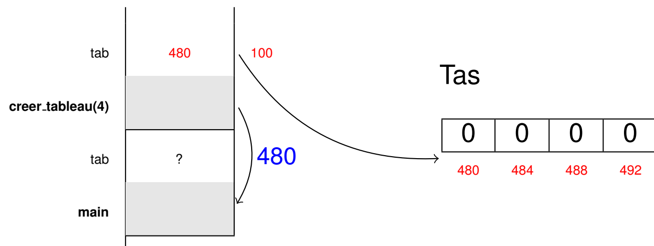
- prends en entrée le nombre de case à allouer et la taille de chaque case,
- alloue le bloc de mémoire dans le tas.
- Initialise chaque case à 0.
- Renvoie l'adresse du bloc de mémoire.

Syntaxe d'un appel à la fonction `calloc`

```
calloc(nombre cases,taille case);
```

Les fonctions d'allocations

```
1 int * creer_tableau(int n){
2   int *tab=calloc(n,sizeof(int));
3   return tab;
4 }
5
6 int main(){
7   int * tab=creer_tableau(4);
8   return EXIT_SUCCESS;
9 }
```



La fonction `creer_tableau` a une complexité $\Theta(n)$ en temps et en espace.

Malloc/Calloc

Pourquoi avoir deux fonctions ?

- La fonction `calloc` sert uniquement dans le cas où l'on veut initialiser toutes les cases à 0.
- elle évite de devoir initialiser les cases une à une avec une boucle `for`.
- le code est donc plus simple.
- il arrive souvent qu'on ne veuille pas initialiser les cases à 0.

Allocation/Désallocation

Allocation/Désallocation

- Chaque bloc de mémoire que vous allouez doit être libéré lorsque vous n'en avez plus besoin.
- À la fin d'un programme, le nombre d'allocations doit être égal au nombre de désallocations.

Les fonctions d'allocations

Désallocation

La fonction `void free(void *ptr);`

- prends en entrée un pointeur contenant l'adresse d'un bloc mémoire,
- rend l'espace mémoire à nouveau disponible (libère la mémoire).

Les fonctions d'allocations

```c
int * creer_tableau(int n){
  int *tab=malloc(sizeof(int)*n);
  int i;
  for(i=0;i<n,i++)
    tab[i]=0;
  return tab;
}

int main(){
  int * tab=creer_tableau(4);
  free(tab);
  return EXIT_SUCCESS;
}
```

Les fonctions d'allocations

Réallocation

La fonction `void *realloc(void *ptr, size_t n);`

- prends en entrée un pointeur contenant l'adresse d'un bloc mémoire et une taille `n`,
- alloue l'espace mémoire de taille `n` (`malloc`),
- copie le contenu de la zone mémoire situé à l'adresse `ptr` vers la nouvelle zone mémoire,
- libère l'espace mémoire mémoire localisé par `ptr` (`free`),
- renvoit l'adresse de la nouvelle zone mémoire.

Lorsqu'une zone mémoire devient trop petite. . .

- on ne l'agrandit pas ,
- on en crée une nouvelle.

Structures de taille variable

On peut à présent définir des structures de longueur variable.

Exemple

- Dans le TD5 : on aurait bien eu besoin de créer des garages de longueurs variables.

Mais attention

- si la fonction qui permet de créer une instance d'une structure contient un `malloc...`
- on doit définir une fonction qui permet de `détruire` cette instance.

Structures de taille variable

```
#ifndef _GARAGE_
#define _GARAGE_H
#include"voiture.h"


struct garage_s{
  voiture_t * places; //tableau de voitures.
  int nb_places_max; //Nombre maximum de voitures.
  int nb_voitures; //Nombre d'emplacements occupes.
};
typedef garage_s garage_t;

/**
   Fontion qui prends en entree un nombre n,
   alloue un tableau de n voitures et renvoit le garage.
   @param n nombre de places dans le garage
   @return un garage initialise
 */
garage_t * creer_garage(int n);

/**
   Fontion qui libere la memoire allouee pour un garage g
   @param g un garage
 */
void detruire_garage(garage_t * g);

#endif
```

Structures de taille variable

```
1  garage_t * creer_garage(int n){
2    garage_t * res=malloc(sizeof(garage_t));
3    res->places=malloc(sizeof(voiture_t)*n);
4    res->nb_places_max=n;
5    res->nb_voitures=0;
6    return res;
7  }
8
9  void detruire_garage(garage_t * g){
0    free(g->places);
1    free(g);
2  }
```

Copie de variables structurées

Copier une variable

- Il n'est plus possible de copier une variable structurée dans une autre en utilisant l'opérateur $=$,
- on ne ferait que copier la valeur du pointeur.
- or deux variables **ne doivent pas** partager le même espace mémoire.

Solution

- Il faut donc écrire une fonction pour recopier la valeur d'une variable structurée.

Copie de variables structurées

```
1 garage_t * copie_garage(garage_t * g){
2   garage_t * res=creer_garage(g->nb_places_max);
3   int i;
4   for(i=0;i<g->nb_voitures;i++)
5     res->places[i]=g->places[i];
6   res->nb_voitures=g->nb_voitures;
7   return res;
8 }
```

Tableaux à plusieurs dimensions

Rappel sur les tableaux

- Un tableau à une dimension est un pointeur.
- un tableau d'entier à une dimension est de type `int *`

Un tableau à deux dimensions

- Chaque case du tableau à deux dimensions est un tableau à une dimension.
- C'est donc un pointeur de pointeur.
- un tableau d'entier à deux dimensions est de type `int **`

Tableaux à plusieurs dimensions

```
1  int ** creer_tableau(int h, int l){
2    int i,j;
3    int ** tab=malloc(sizeof(int*)*h);
4    for(i=0,i<h;i++){
5      tab[i]=malloc(sizeof(int)*l);
6      for(j=0,j<l;j++)
7        tab[i][j]=0;
8    }
9    return tab;
0  }
1
2  void detruire_tableau(int ** tab, int h){
3    int i;
4    for(i=0,i<h;i++)
5      free(tab[i]);
6    free(tab);
7  }
```

On va pouvoir attaquer . . .

le