Programmation C et Structures de données (Prog 2)

Cours 4

Allocation dynamique de mémoire

Pierre Fouilhoux & Christophe Tollu
pierre.fouilhoux@lipn.fr et ct@lipn.univ-paris13.fr

30 janvier 2024

Les trois types d'allocation mémoire

On appelle **allocation mémoire** le fait que de la mémoire vive soit réservée pour stocker des données (variables, tableaux...) ou effectuer des opérations (fonctions...).

Allocation statique/dynamique

- L'**allocation statique** est réalisée automatiquement par les programmes en langage compilé (comme le C et non comme Python) : il s'agit principalement des réservations mémoires correspondant aux variables (variables scalaires, tableaux défini par [] etc) et des zones mémoires stockant le code des fonctions.
- L'**allocation dynamique sur la pile** est réalisée automatiquement lors de l'appel des fonctions : c'est la réservation mémoire nécessaire à l'exécution des fonctions avec leur variables locales
- L**allocation dynamique sur le tas** est réalisée par des commandes spécifiques du programme utilisées par le programmeur (malloc, calloc...)

Zones de mémoire

Variables locales, information sur les fonctions

Allocation dynamique

Variables globales et statiques

Contenu des fonctions

Allocation statique/dynamique

- L'allocation statique et l'allocation dynamique dans la pile, qui sont gérés automatiquement, sont
  - très efficace en temps d'exécution.
  - pas très efficace ou pratique en vitesse d'exécution
- L'allocation dynamique sur la pile est gérée par le programmeur
  - c'est une source de difficultés (donc d'erreurs potentielles)
  - mais cela permet une utilisation efficace de l'espace mémoire pour l'algorithmique, par exemple pour les structures de données avancées (liste, arborescence...).

Sans allocation dynamique sur la pile

Sans allocation dynamique sur la pile

- créer/réserver de l'espace mémoire (un tableau par exmple) par une fonction pour l'extérieur d'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

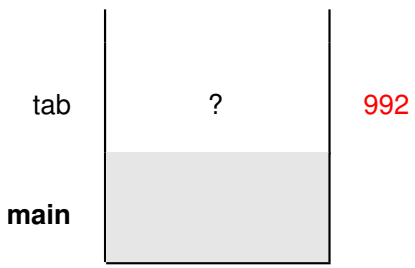
!! Idée à ne pas suivre !!

```
1  #include <stdlib.h>
2
3  int * creer_tableau(int n) {
4    int tab[n], i;
5    for(i=0; i<n; ++i)
6      tab[i]=i;
7    return tab; /*INTERDIT: NE PAS RETOURNER UNE ADRESSE DE TABLEAU DEFINIE PAR []*/
8  }
9  int main(){
0    int * tab = creer_tableau(4);
1    return EXIT_SUCCESS;
2  }
```

En effet, à l'issue de l'appel de la fonction, la réservation mémoire du tableau est libérée : ainsi l'adresse mémoire retournée par la fonction ne doit pas être utilisée dans le main.

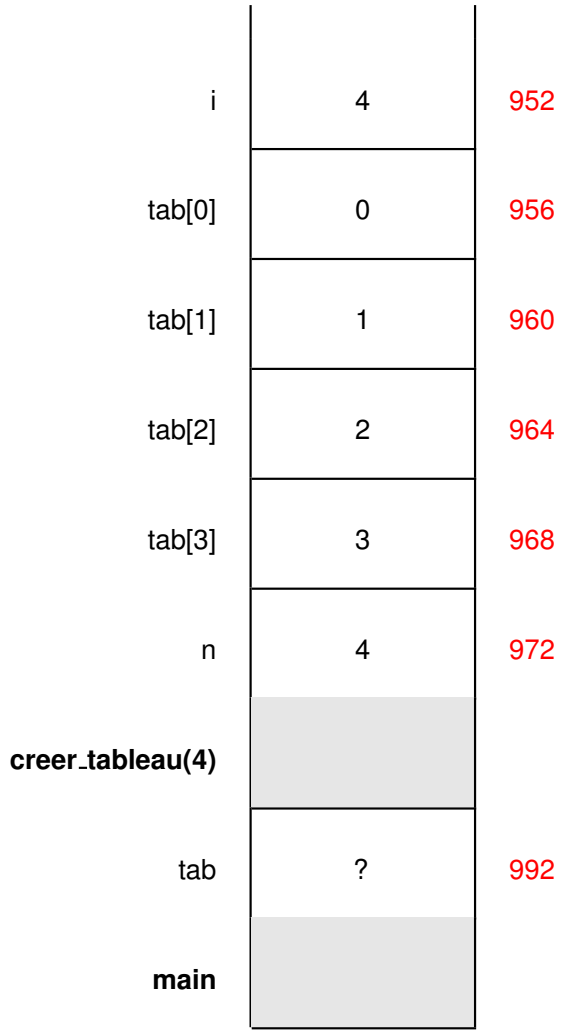
Réserver de l'espace mémoire avec une fonction

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



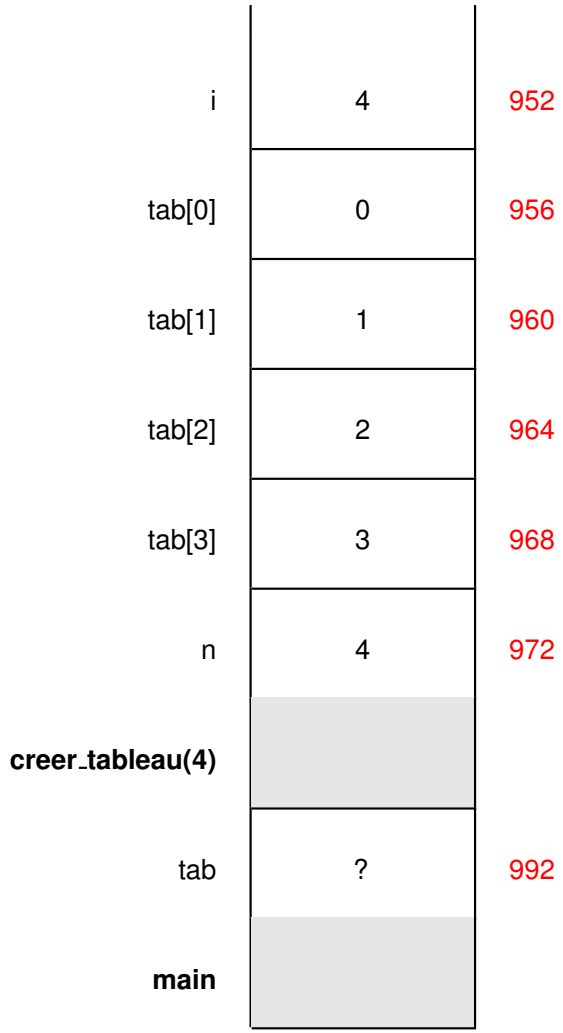
Réserver de l'espace mémoire avec une fonction

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



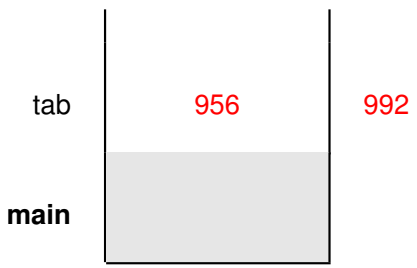
Réserver de l'espace mémoire avec une fonction

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



Réserver de l'espace mémoire avec une fonction

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



La variable *tab* pointe à une adresse "désallouée"!!!

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

Code peu robuste : c'est-à-dire qu'il peut provoquer des erreurs suivant ce que tape l'utilisateur du programme.

/* Saisie d'une chaîne de caractères d'au plus 9 caractères */

```
1 int main()
2 {
3     char chaine[10];
4     scanf("%s",chaine);
5     printf("%s\n",chaine);
6     return EXIT_SUCCESS;
7 }
```

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

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. . .
- . . . on va s'en servir pour allouer dans le tas.

L'allocation dynamique

Notion Fondamentale : l'allocation dynamique

L'allocation dynamique sur le tas : que l'on appelle souvent *allocation dynamique* tout court) consiste à allouer de la mémoire dans la zone de mémoire appelée *tas*.

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 quelle fonction.
- On peut décider de "modifier la taille" d'un bloc 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 sur le tas : que l'on appelle souvent *allocation dynamique* tout court) consiste à allouer de la mémoire dans la zone de mémoire appelée *tas*.

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 quelle fonction.
- On peut décider de "modifier la taille" d'un bloc 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 sur le tas : que l'on appelle souvent *allocation dynamique* tout court) consiste à allouer de la mémoire dans la zone de mémoire appelée *tas*.

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 quelle fonction.
- On peut décider de "modifier la taille" d'un bloc 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 taille);` *(stdlib.h)*

- prend en argument 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.

Le type *void** indique le type "générique" pointeur,
c'est-à-dire que tout pointeur typé (comme *int*, char **...) est de type *void **.

Définir la taille d'un bloc mémoire

/* Déclaration d'une zone de mémoire correspondant à une variable */

```
1 double *f ; /* Declaration d'une variable pointeur sur double */
2
3 f = malloc(sizeof(double)); /* Allocation d'une zone memoire
4                                de la taille de stockage d'un flottant */
```

/* Déclaration d'une zone de mémoire correspondant à un tableau */

```
1 double *f ; /* Declaration d'une variable pointeur sur double */
2
3 f = malloc(sizeof(double) * 100 ); /* Allocation d'une zone memoire
4                                       de la taille de stockage de 100 flottants */
```

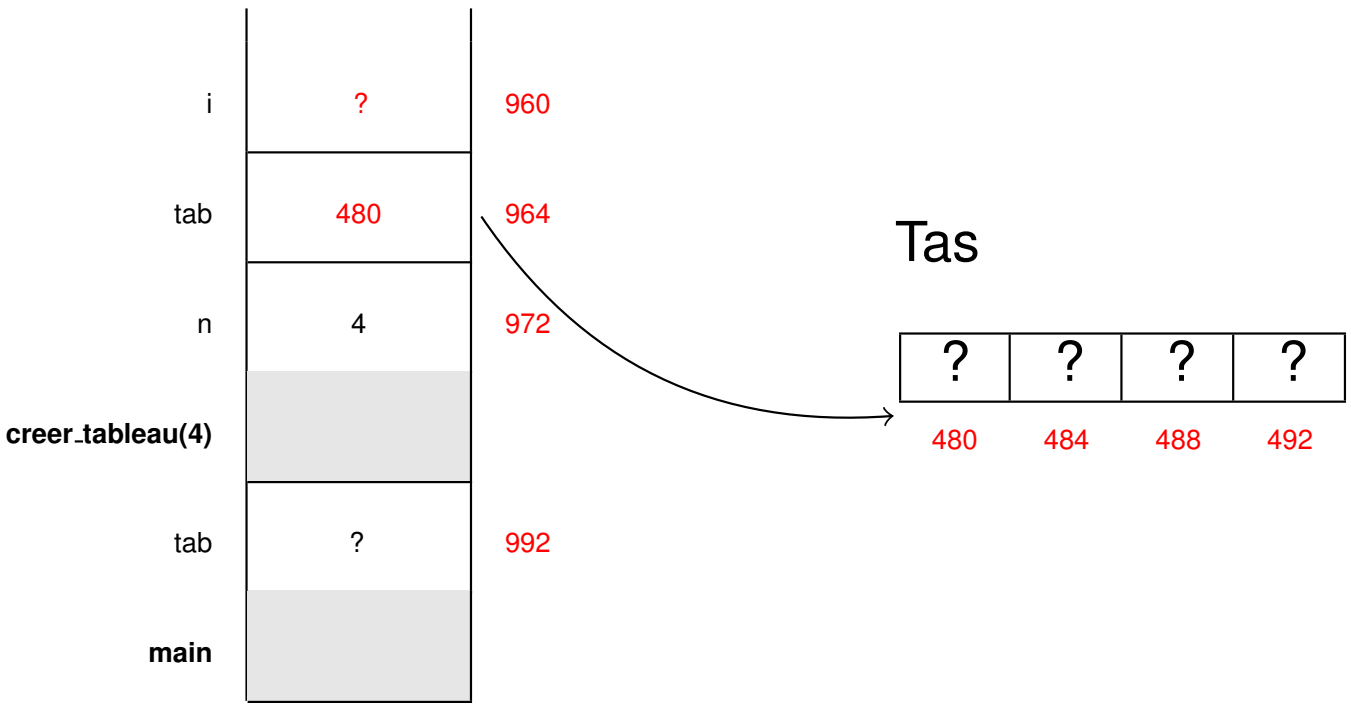
Définir la taille d'un bloc mémoire

Durée de vie d'une allocation dynamique

- L'espace mémoire réservée par malloc reste allouée jusqu'à
  - la fin du programme
  - ou la libération par le programmeur (voir plus loin)
- Ainsi une allocation dans une fonction existe en dehors de la fonction !

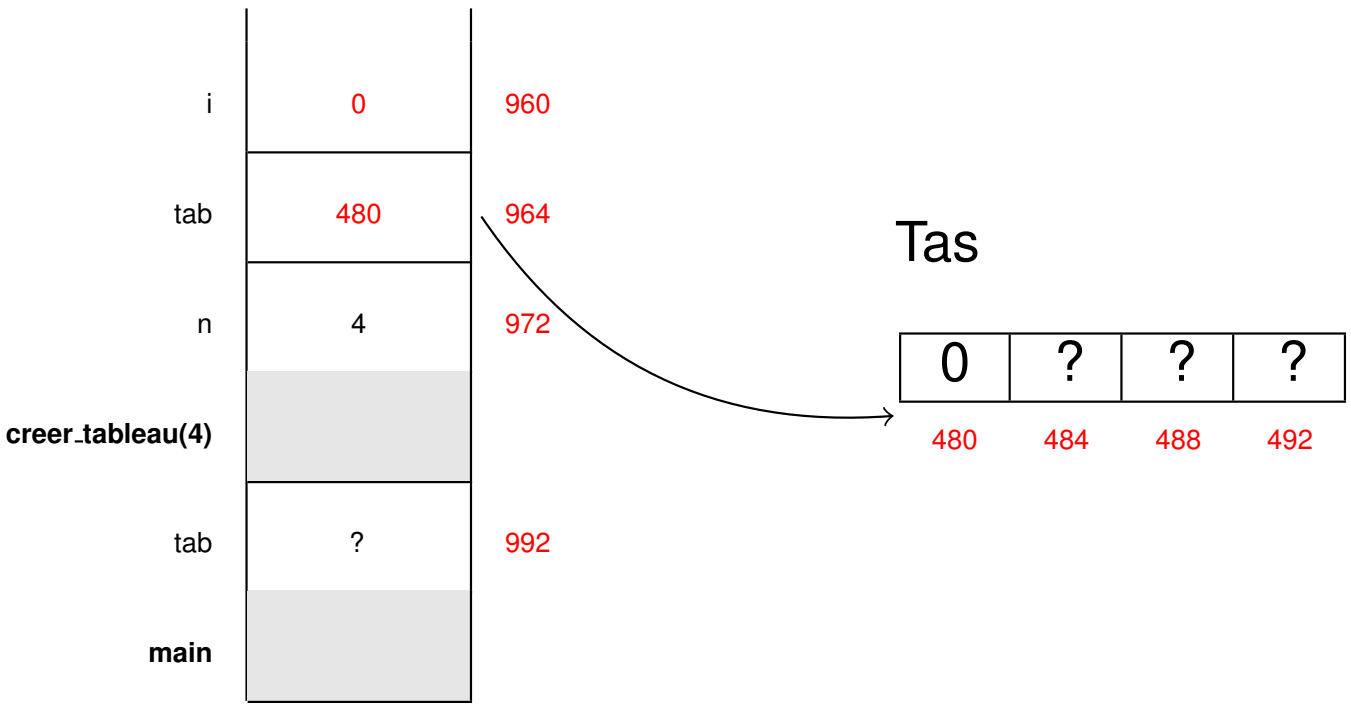
Les fonctions d'allocations

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



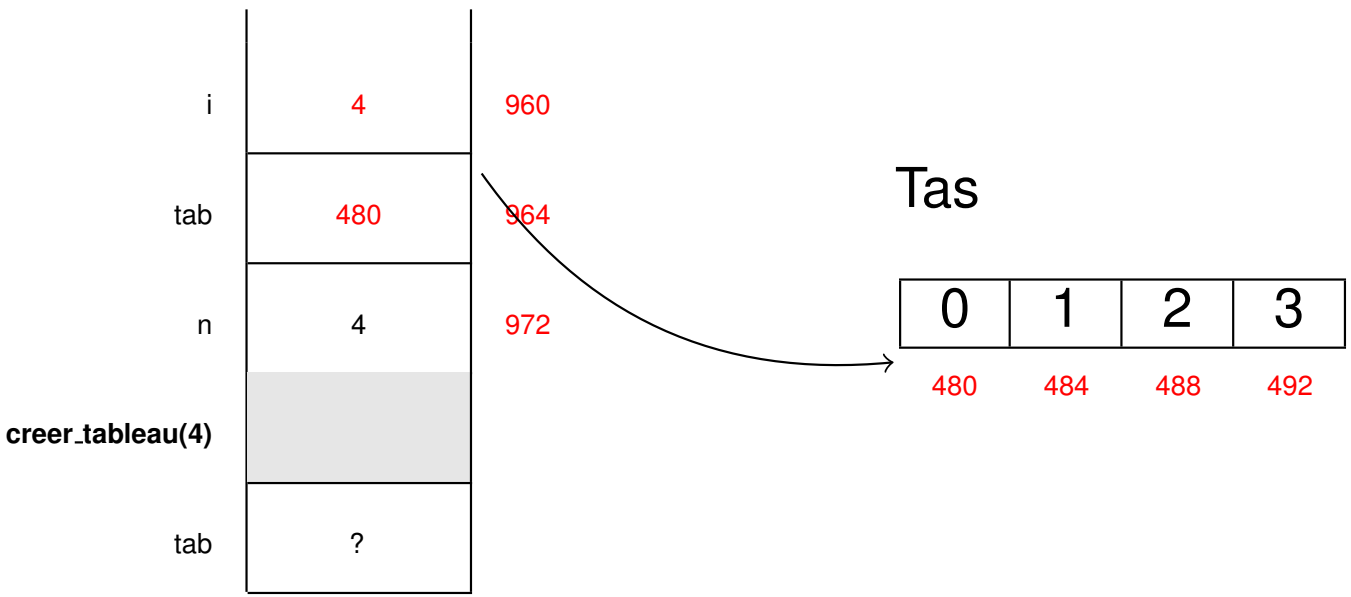
Les fonctions d'allocations

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



Les fonctions d'allocations

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



Les fonctions d'allocations

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

Le tableau existe après la fin de l'appel de la fonction et il est référé par le main dans la variable *tab*.

Si l'allocation mémoire échoue...

- Si l'allocation mémoire a échoué, `malloc` renvoie **un pointeur nul** (`NULL`) ;
- cela peut arriver en cas de réservation d'une grande zone mémoire
- en bonne programmation, il faut s'assurer qu'aucun problème de cette sorte n'apparaît !

Par exemple, dans le main :

```
1   int *tab = malloc(sizeof(int)*10000);
2   if (tab == NULL){
3       fprintf(stderr, "Echec de l'allocation\n");
4       return EXIT_FAILURE;
5   }
```

Dans une fonction, il faut prévoir que la fonction qui réalise une allocation renvoie un message d'erreur.

Les fonctions d'allocations

L'allocation

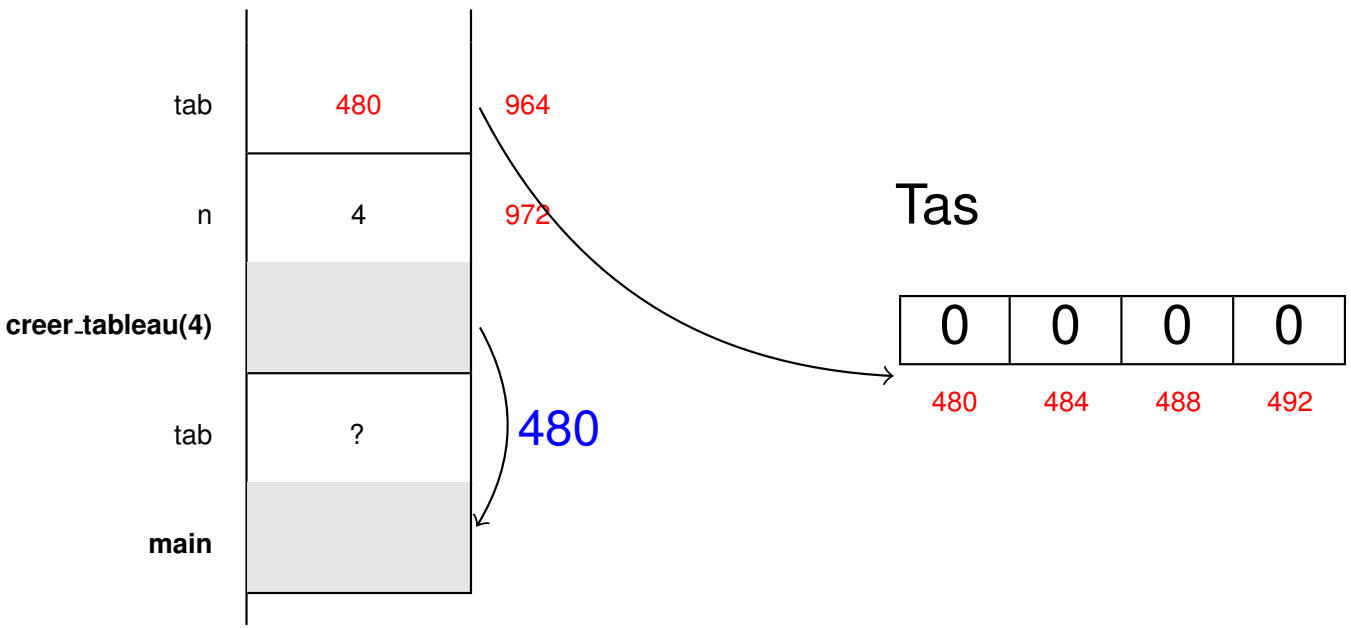
La fonction

`void *calloc(size_t nbre_cases, size_t taille_case);` *(stdlib.h)*

- prend en argument **le nombre de cases** 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.

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 }
```



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 (parfois) plus simple mais 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.

La fonction de désallocation

Désallocation

La fonction `void free(void *ptr);` *(stdlib.h)*

- prend en argument un pointeur contenant l'adresse d'un bloc mémoire, auparavant alloué par `malloc` ou `calloc` (ou `realloc`)
- rend l'espace mémoire à nouveau disponible (libère la mémoire).

```
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    ...
2    free(tab);
3    tab=NULL;
4    return EXIT_SUCCESS;
5  }
```

Les fonctions d'allocations

Réallocation

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

- prend en argument un pointeur contenant l'adresse d'un bloc mémoire et une taille `n`,
- alloue un bloc mémoire de taille `n`,
- copie le contenu du bloc mémoire situé à l'adresse `ptr` vers le nouveau bloc mémoire,
- libère l'espace mémoire adressé par `ptr`,
- renvoie l'adresse du nouveau bloc mémoire.

Lorsqu'on constate que les données sont trop nombreuses

- un bloc mémoire peut être trop petit :
  la commande *realloc* recopie la zone dans un block plus grand
- ATTENTION : cette commande peut utiliser beaucoup de temps !

Structures (avec des champs) de taille variable

On peut à présent définir des structures de longueur variable.

Exemple

On peut avoir besoin de créer des noms d'individus (ou des titres de livres) de longueur variable.

```
1 struct contact_s{
2   char * nom;
3   char * numtel;
4   int departement;
5 };
```

Structures (avec des champs) de taille variable

```
1  /** Fonction qui cree et initialise un individu champ par champ */
2  /** et renvoie son adresse */
3  struct contact_s * creer_contact(char *nom, char *numtel, int departement) {
4    struct contact_s * res = malloc(sizeof(struct contact_s));
5    res->nom = malloc(sizeof(char)*(strlen(nom)+1));
6    strcpy(res->nom, nom);
7    res->numtel = malloc(sizeof(char)*(strlen(numtel)+1));
8    strcpy(res->numtel, numtel);
9    res->departement=departement;
0    return res;
1  }
2
3  void affichage_contact(struct contact_s *C){
4    printf("%s dep. %d: %s\n", C-> nom, C->departement, C->numtel);
5
6  }
7  int main(){
8    struct contact_s* C = creer_contact("Raoul Boisjoli", "06.43.30.20.12", 63);
9
0    affichage_contact(C);
1
2    free(C);
3    C=NULL;
4    return EXIT_SUCCESS;
5  }
```

Structures (avec des champs) de taille variable

Pointeur de structure

- Dans le main précédent, on manipule un pointeur sur struct

```
1    struct contact_s* C = creer_contact("Raoul Boisjoli", "06.43.30.20.12", 63);
```

- C'est un choix qui permet d'avoir des struct parfois gros et en quantité importante dans des structures légères (on verra les tableaux de pointeurs sur struct !)

Désallocation !

- Mais il faut penser à désallouer le bloc

```
1 void detruire_contact (struct contact_s * c) {
2   free(c->nom);
3   free(c->numtel);
4   free(c);
5 }
```

- **N.B.** C'est le bloc mémoire le plus tardivement créé qui est libéré le plus tôt.

Affectation par copie d'une variable

Premier niveau d'erreur

- Si on manipule un pointeur

```
1    struct contact_s* C = creer_contact("Raoul Boisjoli", "06.43.30.20.12");
```

- La commande

```
1    struct contact_s* C2;
2    C2=C;
```

affecte un deuxième pointeur sur le même struct

Pour copier le contenu d'un struct dans une autre, on doit faire une affection plus "profonde" (deep copy).

Affectation par copie d'une variable

```
1   struct contact_s* C1 = creer_contact("Raoul Boisjoli", "06.43.30.20.12", 63);
2   struct contact_s* C2 = C1;
3   C1->departement=64;
4   affichage_contact(C1);
5   affichage_contact(C2);
```

Affiche : Raoul Boisjoli dep. 64 : 06.43.30.20.12
Raoul Boisjoli dep. 64 : 06.43.30.20.12

En effet, on affiche deux fois la même zone mémoire.

Affectation par copie d'une variable

Deuxième niveau d'erreur

- Avec

```
1    struct contact_s* C = creer_contact("Raoul Boisjoli", "06.43.30.20.12");
2    struct contact_s* C2;
```

- La commande

```
1    *C2=*C;
```

  On recopie champs à champs les données : **sans les dupliquer !!!**

- Ainsi les deux struct pointent vers les mêmes champs !!!!

Pour copier le contenu d'un struct dans une autre, on doit faire une affection plus "profonde" (deep copy)

Affectation par copie d'une variable

```
1   struct contact_s* C1 = creer_contact("Raoul Boisjoli", "06.43.30.20.12", 63);
2   struct contact_s* C3=  creer_contact("", "", 0);
3   *C3=*C1;
4   C1->departement=64;
5   C1->nom[0]='T';
6   affichage_contact(C1);
7   affichage_contact(C3);
```

Affiche : Taoul Boisjoli dep. 64 : 06.43.30.20.12
Taoul Boisjoli dep. 63 : 06.43.30.20.12

En effet, les deux struct du nom pointent sur la même chaîne de caractères !

Copie (duplication) de variables structurées

Solution

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

```
1 struct contact_s * copier_contact(struct contact_s *C) {
2   return  creer_contact(C->nom, C->numtel, C->departement);
3 }
```

Affectation par copie d'une variable

```
1  struct contact_s* C1 = creer_contact("Raoul Boisjoli", "06.43.30.20.12", 63);
2  struct contact_s* C4=copier_contact(C1);
3  C1->departement=64;
4  C1->nom[0]='T';
5  affichage_contact(C1);
6  affichage_contact(C4);
```

Affiche : Taoul Boisjoli dep. 64 : 06.43.30.20.12
Raoul Boisjoli dep. 63 : 06.43.30.20.12

Enfin, les deux pointeurs sur struct sont différents avec des données différentes !

Tableaux à plusieurs dimensions

Rappel sur les tableaux

- Un (nom de) tableau à une dimension est un pointeur (constant) ;
- par exemple, un tableau de flottants à une dimension est de type `float *`.

Tableau à deux dimensions (matrices)

- Un tableau à deux dimensions est un tableau à une dimension dont chaque clé est elle-même un tableau à une dimension ;
- c'est donc un tableau de pointeurs, ou encore un **pointeur vers un bloc de pointeurs** ;
- par exemple, un tableau de flottants à deux dimensions est de type `float **`.

Créer / détruire des tableaux à deux dimensions

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

void detruire_tableau(int ** tab, int h)
{
    int i;
    for(i=0; i<h; i++)
        free(tab[i]);
    free(tab);
}
```