

# Introduction à la Programmation

Benoit Donnet  
Année Académique 2021 - 2022



1

## Agenda

- Introduction
- Chapitre 1: Bloc, Variable, Instruction Simple
- Chapitre 2: Structures de Contrôle
- Chapitre 3: Méthodologie de Développement
- Chapitre 4: Introduction à la Complexité
- Chapitre 5: Structures de Données
- Chapitre 6: Modularité du Code
- Chapitre 7: Pointeurs
- Chapitre 8: Allocation Dynamique

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# Principe

- Jusqu'à présent, on a créé de manière statique
  - les tableaux
    - ✓ on connaissait, au moment de l'écriture/la compilation du programme, la taille du tableau
  - les structures de données
    - ✓ on était certain d'en avoir besoin dans le programme
- Ce n'est pas toujours possible dans le monde réel
- On est amené à traiter des données dont
  - ✓ la taille n'est pas connue au moment de l'écriture/compilation du programme
  - ✓ l'existence n'est pas certaine à l'écriture/compilation du programme

## Principe (2)

- Le C permet d'allouer dynamiquement de la mémoire
  - *création* de tableaux dont la taille est variable, en fonction des besoins
  - *création* d'enregistrements en fonction des besoins
  - de manière générale, *création* d'espaces mémoires en fonction des besoins
  - *libération* la mémoire après utilisation
- La mémoire est allouée dynamiquement sur le **tas** (*heap*)
  - cfr. Chap. 7

# Principe (3)

- C fournit les bibliothèques permettant d'allouer/libérer de la mémoire
  - `stdlib.h`

```
void *malloc(unsigned int n)
void *calloc(unsigned int nmemb, unsigned int n)
void *realloc(void *p, unsigned int n)

void free(void *p)
```

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
    - ✓ `malloc()`
    - ✓ `calloc()`
    - ✓ `realloc()`
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# malloc()

- Permet d'allouer un bloc de mémoire sur le tas
  - `void *malloc(unsigned int n)`
    - ✓ `n`, le nombre d'octets à allouer sur le tas
    - ✓ si succès, retourne un pointeur vers le bloc mémoire (de `n` octets) qui vient d'être alloué
    - ✓ si échec, retourne `NULL`

# malloc() (2)

- Comment connaître l'espace mémoire nécessaire pour un type primitif et/ou une structure complexe?
- Opérateur **sizeof** ()
  - doit être appliqué à un *type*
  - retourne le nombre d'octets nécessaire au stockage de l'instance du type
  - intérêt?
    - ✓ difficile de connaître la taille d'une structure, surtout si elle est complexe
- Exemple
  - **sizeof**(float)

# malloc() (3)

- Exemple
  - création d'un entier

```
#include <stdlib.h>

/*
 * @pre: /
 * @post: cree_entier vaut un pointeur valide vers un entier
 *         valant val. NULL en cas d'erreur
 */
int *cree_entier(int val){
    int *i = malloc(sizeof(int));
    if(i==NULL)
        return NULL;

    *i = val;

    return i;
} //fin cree_entier()
```

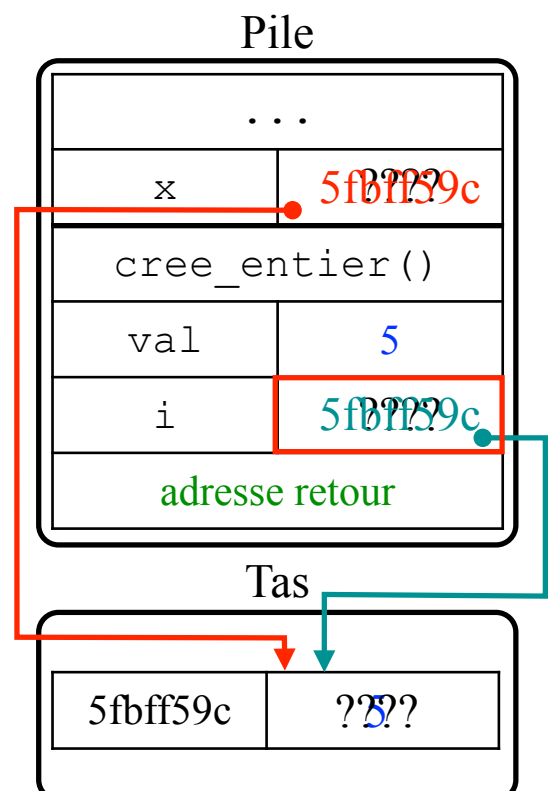
# malloc() (4)

```
int *cree_entier(int val){
    int *i = malloc(sizeof(int));
    if(i==NULL)
        return NULL;

    *i = val;

    return i;
} //fin cree_entier()
```

```
int *x = cree_entier(5);
```



# malloc() (5)

- Exemple

- création d'un tableau de  $n$  réels, avec  $n > 0$

```
#include <stdlib.h>
#include <assert.h>

/*
 * @pre: n>0
 * @post: alloue_tableau vaut un pointeur vers un tableau de
 *        réels. NULL sinon
 */
float *alloue_tableau(int n){
    assert(n>0);
    float *tab;

    tab = malloc(n*sizeof(float));
    if(tab==NULL)
        return NULL;

    return tab;
} //fin alloue_tableau()
```

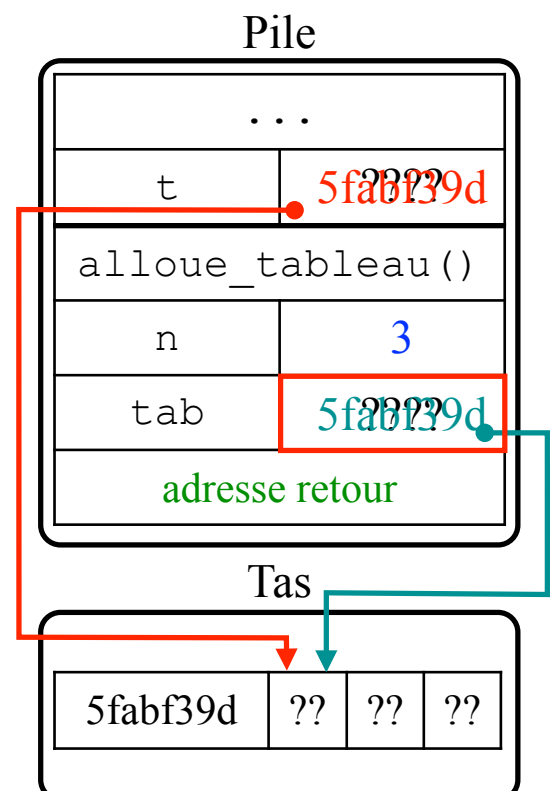
# malloc() (6)

```
float *alloue_tableau(int n){
    assert(n>0);
    float *tab;

    tab = malloc(n*sizeof(float));
    if(tab==NULL)
        return NULL;

    return tab;
} //fin alloue_tableau()
```

```
float *t = alloue_tableau(3);
```



# malloc() (7)

- Exemple
  - écrire une fonction qui recopie le tableau en paramètre

```
#include <stdlib.h>

/*
 * @pre: n > 0 et tab valide
 * @post: copie vaut une copie de tab.  NULL en cas d'erreur
 */
int *copie(int *tab, int n){
    int *tab2 = malloc(sizeof(int)*n);
    if(tab2==NULL)
        return NULL;
    int i;
    //Inv: tab2[0...i-1] = tab[0...i-1], 0 ≤ i ≤ n
    for(i=0; i<n; i++)
        tab2[i] = tab[i];

    return tab2;
} //fin copie()
```

# calloc()

- Permet d'allouer un bloc de mémoire sur le tas et remplit, en plus, la zone de zéros
  - void \*calloc(unsigned int nmem,  
unsigned int n)
    - ✓ nmem, nombre de cases à allouer
    - ✓ n, taille de chaque case mémoire



# calloc()

- Exemple

```
#include <stdlib.h>
#include <assert.h>

/*
 * @pre: nb_cases>0
 * @post: alloue_tableau vaut un pointeur vers un tableau
 *        d'entiers. NULL sinon
 */
int *alloue_tableau(int nb_cases){
    assert(nb_cases>0);
    int *tab;

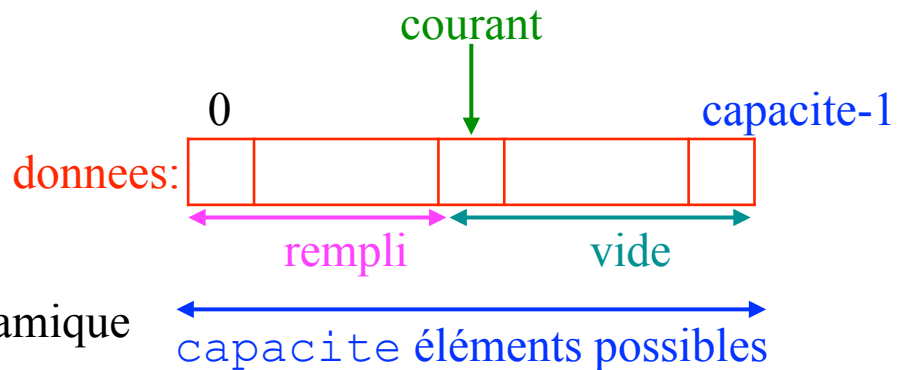
    tab = calloc(nb_cases, sizeof(int));
    if(tab==NULL)
        return NULL;

    return tab;
} //fin alloue_tableau()
```

# realloc()

- Permet de réallouer une zone mémoire à une nouvelle taille
  - void \*realloc(void \*p, unsigned int n)
    - ✓ p, zone mémoire allouée dynamiquement au préalable
    - ✓ n, nouvelle taille
- Les anciennes données sont conservées
  - tronquées si la taille est diminuée
- Possible copie des données sous-jacentes
- p doit pointer sur une zone valide!

# realloc() (2)



- Exemple
  - tableau dynamique

```
#include <stdlib.h>
#include <assert.h>
```

```
typedef struct{
    int *donnees;
    int capacite;
    int courant;
}tableau;
```

tableau  
taille du tableau  
indice courant

# realloc() (3)

```
/*
 * Ajoute valeur au tableau en l'élargissant si nécessaire
 * @pre: t!=NULL
 * @post: ajout vaut 1 si valeur a été ajouté dans t. -1 sinon.
 */
int ajout(tableau *t, int valeur){
    assert(t!=NULL && t->donnees!=NULL);

    if(t->courant == t->capacite){
        t->capacite *= 2;
        t->donnees = realloc(t->donnees, t->capacite*sizeof(int));

        if(t->donnees==NULL)
            return -1;
    }

    t->donnees[t->courant] = valeur;
    t->courant++;

    return 1;
} //fin ajout()
```

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
    - ✓ Principe
    - ✓ Exemple
    - ✓ Bonne Pratique
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# Principe

- Permet de libérer un bloc de mémoire précédemment alloué sur le tas
  - toute allocation de mémoire via `malloc()` doit être libérée
    - ✓ sinon, **fuite mémoire** (*memory leak*)
- `void free(void *p)`
  - libère le bloc pointé par `p`

# Exemple

- Création et libération d'un entier

```
#include <stdlib.h>

int *cree_entier(int val){
    int *i = malloc(sizeof(int));
    if(i==NULL) return NULL;

    *i = val;

    return i;
} //fin cree_entier()

int main(){
    int *x = cree_entier(5);
    //...
    free(x);
    //...
    return 0;
} //fin programme
```

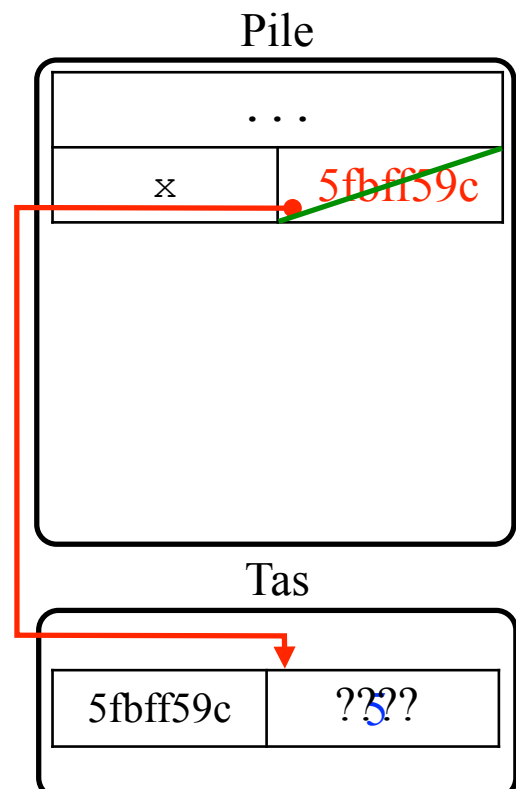
## Exemple (2)

```
int *cree_entier(int val){
    int *i = malloc(sizeof(int));
    if(i==NULL)
        return NULL;

    *i = val;

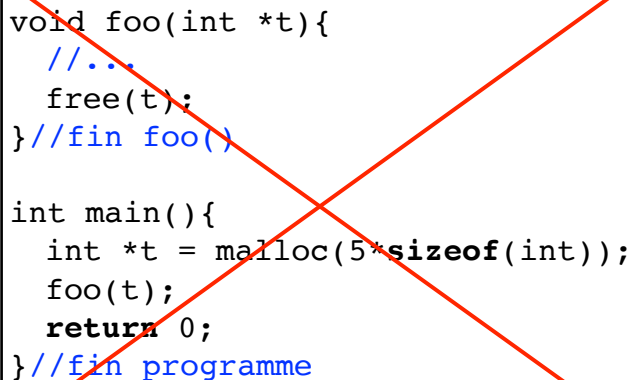
    return i;
} //fin cree_entier()
```

```
int *x = cree_entier(5);
//...
free(x);
x = NULL;
```



# Bonne Pratique

- Ne jamais lire un pointeur sur une zone libérée
- Ne jamais appliquer `free()` sur une zone non allouée au préalable
- `1 malloc() == 1 free()`
  - c'est celui qui alloue qui libère



```
void foo(int *t){
    //...
    free(t);
} //fin foo()

int main(){
    int *t = malloc(5*sizeof(int));
    foo(t);
    return 0;
} //fin programme
```

```
void foo(int *t){
    //...
} //fin foo()

int main(){
    int *t = malloc(5*sizeof(int));
    foo(t);
    free(t);
    return 0;
} //fin programme
```

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# Alloc. Enregistrement

- La bonne pratique veut qu'on définisse une fonction pour allouer un enregistrement, une autre pour le détruire
  - la fonction d'allocation est appelée **constructeur**
  - la fonction de libération est appelée **destructeur**
- Pour l'allocation
  - on passe en arguments les valeurs initiales des différents champs

## Alloc. Enregistrement (2)

```
#include <stdlib.h>
typedef struct{
    double reel;
    double image;
}Complexe;

Complexe *cree_complexe(double r, double i){
    Complexe *c = malloc(sizeof(Complexe));
    if(c==NULL)
        return NULL;
    c->reel = r;
    c->image = i;
    return c;
} //fin cree_complexe()

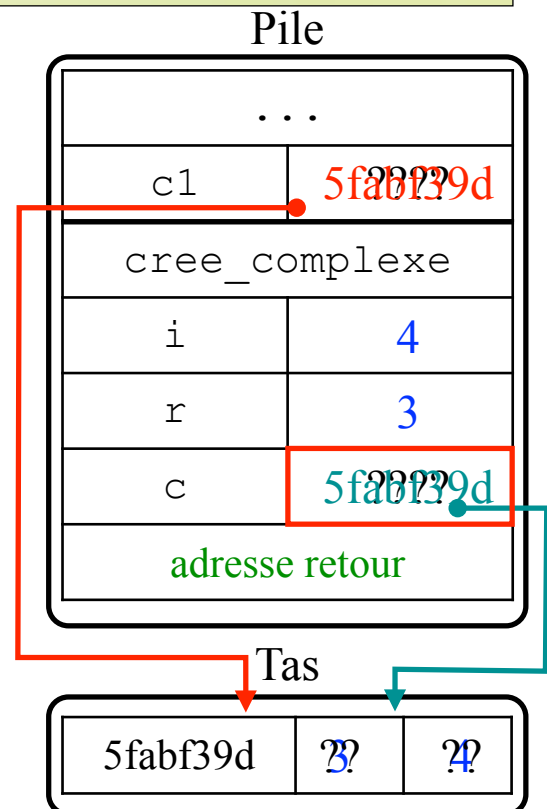
void detruit_complexe(Complexe *c){
    if(c==NULL)
        return;
    free(c);
} //fin detruit_complexe()
```

# Alloc. Enregistrement (3)

```

Complexe *cree_complexe(double r,
double i){
    Complexe *c =
    malloc(sizeof(Complexe));
    if(c==NULL)
        return NULL;
    c->reel = r;
    c->image = i;
    return c;
} //fin cree_complexe()
    
```

```
Complexe *c1 = cree_complexe(3, 4);
```



# Alloc. Enregistrement (4)

- Attention, la valeur retournée par **sizeof** ( ) pour un enregistrement est dépendante de l'alignement mémoire des différents champs
  - cfr. Chap. 5, Slides 134 → 140
- Potentiellement, le résultat de **sizeof()** peut être supérieur à la somme des besoins de chaque champ

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
    - ✓ Allocation
    - ✓ Libération
  - Application

# Allocation

- Tableau de tableaux
  - allouer le tableau principal
  - allouer chacun des sous-tableaux

```
int **cree_matrice(int n, int m){
    int i;

    int **t = malloc(n*sizeof(int *));
    if(t==NULL)
        return NULL;

    for(i=0; i<n; i++){
        t[i] = malloc(m*sizeof(int));
        if(t[i]==NULL)
            //il faut libérer tout ce qui a déjà été alloué
    }//fin for - i

    return t;
}//fin cree_matrice()
```



# Libération

- Libérer
  - d'abord les sous-tableaux
  - ensuite le tableau principal

```
void libere_matrice(int **t, int n, int m){
    int i;

    if(t==NULL)
        return;

    for(i=0; i<n; i++){
        if(t[i]!=NULL)
            free(t[i]);
    }//fin for - i

    free(t);
}//fin libere_matrice()
```

# Agenda

- Chapitre 8: Allocation Dynamique
  - Principe
  - Allocation de Mémoire
  - Libération de Mémoire
  - Allocation Enregistrements
  - Tableaux à 2 Dimensions
  - Application

# Application

- Manipulation de matrices à composantes réelles
- fichier `matrice.h`

```
typedef struct{
    double **el;
    unsigned int n, m;
}Matrice;

/*
 * @pre: n>0, m>0
 * @post: cree_matrice vaut un pointeur vers une matrice n*m
 *        t.q.  $\forall i \in [0, n-1], \forall j \in [0, m-1], t[i][j] = 0.$ 
 *        NULL en cas d'erreur.
 */
Matrice *cree_matrice(unsigned int n, unsigned int m);
```

## Application (2)

- Fichier `matrice.h` (suite)

```
/*
 * @pre: m valide.
 * @post: espace mémoire occupé par la matrice libéré.
 */
void libere_matrice(Matrice *m);

/*
 * @pre: a valide, b valide.
 * @post: somme_matrice vaut un pointeur vers la somme des 2
 *        matrices. NULL en cas d'erreur.
 */
Matrice *somme_matrice(Matrice *a, Matrice *b);

/*
 * @pre: a valide, b valide.
 * @post: produit_matrice vaut un pointeur vers le produit des 2
 *        matrices. NULL en cas d'erreur
 */
Matrice *produit_matrice(Matrice *a, Matrice *b);
```

# Application (3)

- Fichier `matrice.c`

```
#include <stdio.h>
#include <stdlib.h>

#include "matrice.h"

Matrice *cree_matrice(unsigned int n, unsigned int m){
    Matrice *r;
    unsigned int i, j;

    if(! (r=malloc(sizeof(Matrice)))) {
        return NULL;

    if(! (r->el=malloc(n*sizeof(double *))) {
        free(r);
        return NULL;
    }

    //à suivre en cas d'échec, il faut libérer ce qui a déjà été alloué
} //fin cree_matrice()
```

Création de l'enregistrement  
matrice

Création des n lignes du tableau,  
chaque ligne étant un pointeur  
sur double

# Application (4)

```
matrice *cree_matrice(unsigned int n, unsigned int m){
    //cfr. slide précédent

    for(i=0; i<n; i++){
        r->el[i] = malloc(m*sizeof(double));
        if(!r->el[i]){
            for(j=0; j<i; j++){
                free(r->el[j]);
            }
            free(r->el);
            free(r);
            return NULL;
        }

        for(j=0; j<m; j++){
            r->el[i][j] = 0.0;
        } //fin for - i

        r->n = n;
        r->m = m;
        return r;
    } //fin cree_matrice()
```

création des colonnes  
pour chaque ligne

en cas d'échec, il faut libérer  
tout ce qui a déjà été alloué

initialisation des colonnes  
pour la ligne courante

initialisation des derniers champs  
de l'enregistrement matrice

# Application (5)

- Fichier `matrice.c` (suite)

```
void libere_matrice(Matrice *r){
    unsigned i;
    for(i=0; i<r->n; i++)
        free(r->el[i]);

    free(r->el);
    free(r);
} //fin libere_matrice()

Matrice *somme_matrice(Matrice *a, Matrice *b){
    Matrice *s;
    unsigned int i, j;
    if(a->n!=b->n || a->m!=b->m)
        return NULL;

    //à suivre
} //fin somme_matrice()
```

libération des colonnes  
pour chaque ligne

libération de toutes les lignes  
libération de l'enregistrement

# Application (6)

```
Matrice *somme_matrice(Matrice *a, Matrice *b){
    //cfr. slide précédent
    s = cree_matrice(a->n, a->m);
    if(s==NULL)
        return NULL;

    for(i=0; i<a->n; i++)
        for(j=0; j<a->m; j++)
            s->el[i][j] = a->el[i][j] + b->el[i][j];
    return s;
} //fin somme_matrice()

Matrice *produit_matrice(Matrice *a, Matrice *b){
    Matrice *p;
    unsigned int i, j, k;

    if(a->n != b->m)
        return NULL;

    //à suivre
} //fin produit_matrice()
```

# Application (7)

```
Matrice *produit_matrice(Matrice *a, Matrice *b){
    //cfr. slide précédent

    p = cree_matrice(a->n, b->m);
    if(p == NULL)
        return NULL;

    for(i=0; i<a->n; i++)
        for(j=0; j<b->m; j++)
            for(k=0; k<a->m; k++)
                p->el[i][j] += a->el[i][k]*b->el[k][j];

    return p;
} //fin produit_matrice()
```

## Exercice

- Ecrire un programme qui charge, sous la forme d'un tableau de `float`, un fichier texte. La 1<sup>ère</sup> ligne du fichier donne le nombre d'élément du tableau, ensuite chaque ligne donne un élément du tableau
  - écrire la fonction qui charge le tableau à partir du fichier
  - écrire une fonction qui affiche le contenu du tableau
  - écrire le programme principal qui charge le fichier et affiche le tableau