# Programmation avancée en C :

## Allocation dynamique

Licence informatique 3e année

Université Gustave Eiffel



## Les pointeurs

- Pointeur = adresse mémoire + type (interprétation)
- `type* nom;` ⇒ `nom` pointe vers une zone mémoire interprétée comme `type`
- Le type peut être quelconque (C doit juste être capable d'évaluer son `sizeof`).
- Valeur spéciale `NULL` ∼ l'adresse 0. (`NULL` = `(void*)0`)
- Pointeur générique : `void* nom;`



## Arithmétique des pointeurs

- Un tableau ⇒ un pointeur
- L'addition et la soustraction en fonction de la taille du type

  `t+3` = pointeur vers l'élément d'indice 3 après `t`
- Si le pointeur `t` pointe sur la case `x` du tableau, on peut le décaler sur la case `x+1` en faisant `t++`.

```
1  int* copy(char* src, char* dst){
2    if(src == NULL || dst == NULL){
3      return -1;
4    }
5    while(*src != '\0'){
6      *dst = *src;
7      src++;
8      dst++;
9    }
10   *(++dst) = '\0';
11   return 0;
12 }
```



## Transtypage

- Pas de problème de conversion pour :

  `void*` ← `foo*`, `foo*` ← `void*`
- Problème potentiel pour `bar*` ← `foo*`
- On peut chercher à interpréter la mémoire de façon différente, mais il faut une conversion explicite :

```
foo* f = ...;
bar* b = (bar*)f;
```

- Lors d'un appel de fonction, le matching des types d'arguments doit être parfait ! Il faut caster le cas échéant pour éviter des warning.

## Exemple 1

- Ordre de rangement des octets en mémoire
  ⇒ voir un `int` comme un tableau de 4 octets

```
1  int endianness(void) {
2    unsigned int i=0x12345678;
3    unsigned char* t = (unsigned char*)(&i);
4    switch (t[0]){
5      /* 12 34 56 78 */
6      case 0x12: return BIG_ENDIAN;
7      /* 78 56 34 12 */
8      case 0x78: return LITTLE_ENDIAN;
9      /* 34 12 78 56 */
10     case 0x34: return BIG_ENDIAN_SWAP;
11     /* 56 78 12 34 */
12     default: return LITTLE_ENDIAN_SWAP;
13   }
14 }
```



## Exemple 2

- Voir les octets qui composent un `double`

```
1  void show_bytes(double d){
2    unsigned char* t = (unsigned char*)(&d);
3    int i;
4    for(i = 0; i < sizeof(double); i++){
5      printf("%X␣", t[i]);
6    }
7    printf("\n");
8  }
```

```
nborie@perceval:~> ./test 645.36452324
75 13 29 8B EA 2A 84 40
```



## Façonner la mémoire

- Pour utiliser les données (forcément binaires) dans la mémoire, il faut :
  - l'adresse des données,
  - la taille des paquets et leur interprétation,
  - le nombre de paquets à lire.

| Adresse seulement | |
|---|---|
| `void* t` | `void* c` |
| Adresse + interprétation | |
| `int* t` | `char* c` |
| Adresse + interprétation + nombre de paquets | |
| `int t[64]` | `char c[8]` |

La programmation générique en C consiste à manipuler des `void*` et leur donner l'interprétation souhaitée au moment où l'on veut exploiter les données pointées. (choisir un cast ⇔ choisir une interprétation)



## Allocation dynamique

- Principe : demander une zone mémoire au système
- Zone représentée par son adresse, pris sur le tas
- Zone persistante jusqu'à ce qu'elle soit libérée explicitement par le programme (≠ variables locales)
- Il faut une conversion pour donner l'interprétation à ce zone.

## Malloc

- `void* malloc(size_t size);` dans `<stdlib.h>`
- `size` = taille **en octets** de la zone réclamée
- Retourne la valeur spéciale `NULL` en cas d'échec
- Si réussi, retourne l'adresse d'une zone au contenu **indéfini**

  ⇒ Penser à initialiser la zone avant utilisation !



## Règles d'or de malloc

- **Toujours** tester le retour de malloc
- **Toujours** multiplier le nombre d'éléments par la taille (sizeof)
- **Toujours** mettre un cast pour indiquer le type (ceci est gcc-facultatif, mais plus lisible)
- Usage prototypique :

```
1 Cell* c = (Cell*)malloc(sizeof(Cell));
2 if(c == NULL){
3   /* ... */
4 }
```

- Pour un tableau, on multiplie par le nombre d'éléments.

```
1 int* array = (int*)malloc(size * sizeof(int));
2 if(array == NULL){
3     /* ... */
4 }
```



## Calloc

- `void *calloc(size_t nmemb, size_t size);`
- Alloue un zone de `nmemb` éléments de taille `size` et le remplit avec des zéros :

```
1 int create_array_zero(int size){
2   int* array;
3   array = (int*)calloc(size, sizeof(int));
4   if(array == NULL){
5     fprintf(stderr, "Not enough memory\n");
6     exit(1);
7   }
8   return array;
9 }
```

- Attention ! Paramètres **différents** de `malloc` !
- **Toujours** tester la valeur retournée !



## Realloc

- `void *realloc(void *ptr, size_t size);`
- Réalloue la zone pointée par `ptr` à la nouvelle taille `size`

  - anciennes données conservées

  - ou tronquées si la taille a diminuée
- Possibles copies de données sous-jacentes (avec le coût que cela produit...)
- `ptr` doit pointer sur une zone valide !
- **Toujours** tester la valeur retournée !

## Exemple de réallocation

```c
struct array{
  int* data;
  int capacity;
  int current;
}

/* Add a given value to a given array
   enlarging it if needed */
void add_int(struct array* a, int value){
  if(a->current == a->capacity){
    a->capacity *= 2;
    a->data = (int*)realloc(a->data, a->capacity*sizeof(int));
    if(a->data == NULL){
      fprintf(stderr, "Not enough memory!\n");
      exit(1);
    }
  }
  a->data[a->current] = value;
  a->current++;
}
```



## Libération de la mémoire

- `void free(void *ptr);`
- Libère la zone pointée par `ptr`
- `ptr` peut être à `NULL` (pas d'effet).
- Sinon, `ptr` doit pointer sur une zone valide obtenue avec `malloc`, `calloc` ou `realloc`.
- La zone ne doit pas déjà avoir été libérée.
- **NE JAMAIS** lire un pointeur sur une zone libérée
- Attention aux allocations cachées

  `char *strdup(const char *s);`



## Libération de la mémoire

- 1 `malloc` = 1 `free`
- Celle qui alloue est celle qui libère :

**Pas bien !**

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

int main(int argc, char* argv[]){
  int* t;
  t = (int*)malloc(N*sizeof(int));
  foo(t);
  return 0;
}
```

**Mieux !**

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

int main(int argc, char* argv[]){
  int* t;
  t = (int*)malloc(N*sizeof(int));
  foo(t);
  free(t);
  return 0;
}
```



## Allocations de structures

- Solution propre : faire une fonction d'allocation/initialisation et une fonction de libération

```c
typedef struct{
  double real, imaginary;
}Complex

Complex* new_complex(double r, double i){
  Complex* c = (Complex *)malloc(sizeof(Complex));
  if(c == NULL){
    fprintf(stderr, "Not enough memory!\n");
    exit(1);
  }
  c->real = r;
  c->imaginary = i;
  return c;
}

void free_complex(Complex* c){
  free(c);
}
```

## Utilisation de malloc

- L'allocation est coûteuse en temps et en espace.
- À utiliser avec discernement (pas quand la taille est connue par exemple...)

**Pas bien !**

```
1  int main(int argc, char* argv[]){
2  Complex* a = new_complex(2,3);
3  Complex* b = new_complex(7,4);
4  Complex* c = mult_complex(a,b);
5  /* ... */
6  free_complex(a);
7  free_complex(b);
8  free_complex(c);
9  return 0;
10 }
```

**Mieux !**

```
1  int main(int argc, char* argv[]){
2  Complex a = {2,3};
3  Complex b = {7,4};
4  Complex* c = mult_complex(&a, &b);
5  /* ... */
6  free_complex(c);
7  return 0;
8  }
```



## Tableaux dynamiques à 2 dimensions

- Tableau 2D = tableau de tableaux :
  - allouer le tableau principal (pointeurs vers sous-tableaux)
  - allouer chacun des sous-tableaux

```
1  int** init_array(int n, int m){
2    int i;
3    int** t = (int**)malloc(n * sizeof(int*));
4    if(t == NULL){
5      /* Error handling */
6    }
7    for(i = 0; i < n; i++){
8      t[i] = (int*)malloc(m * sizeof(int));
9      if(t[i] == NULL){
10       /* Error handling */
11     }
12   }
13   return t;
14 }
```

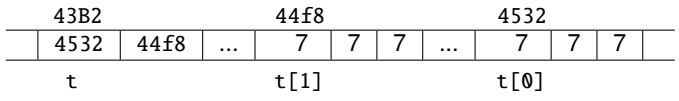
- On peut étendre à $n$ dimensions.



## Tableaux dynamiques à 2 dimensions

- Les éléments ne sont pas toujours contigus !
  ≠ Tableaux statiques
- Les sous-tableaux ne sont pas forcément dans l'ordre.

`int** t de 2x3 contenant la valeur 7`






## Tableaux à 2 dimensions

- Libération :
  - d'abord les sous-tableaux,
  - puis le tableau principal.

```
1  void free_array(int** t, int n){
2    if(t == NULL) return ;
3    int i;
4    for(i = 0; i < n; i++){
5      free(t[i]); /* free(NULL) is OK */
6    }
7    free(t);
8  }
```

## Listes chaînées

- Chaque cellule pointe vers la suivante, donc on peut avoir des listes arbitrairement longues.

```
1  typedef struct cell{
2    float value;
3    struct cell* next;
4  }Cell;
5
6  void print(Cell* list){
7    while(list != NULL){
8      printf("%f\n", list->value);
9      list = list->next;
10   }
11 }
```

- Représentation en mémoire de la liste : 3 7 15

| FB07 | | | FB42 | | | FBAE | |
|---|---|---|---|---|---|---|---|
| 3 | FBAE | ... | 15 | NULL | ... | 7 | FB42 |



## Complexités

- Attention aux complexités cachées de `malloc` et `free` (et de toutes les autres fonctions)
- Si on a une fonction linéaire sur les listes mais que les `malloc` en quadratique, on n'aura pas un temps d'exécution linéaire...



## Bien allouer

- Allocation n'est pas forcément synonyme de `malloc`, qui présente un surcoût mémoire.
- Ne pas utiliser `malloc` quand :
  - plein de petits objets (par exemple < 32 octets)
  - plein d'allocations, mais pas de désallocation
- Solution possible : gérer sa propre mémoire avec un tableau



## Mauvais malloc

```
1  #define SIZE_STRING 29
2  #define N_STRINGS 100000000
3
4  int main(int argc, char* argv[]){
5    int i;
6    for(i = 0; i < N_STRINGS; i++){
7      malloc(SIZE_STRING);
8    }
9    getchar();
10   return 0;
11 }
```

On voit sur le moniteur :

```
nborie@perceval:~> gcc -o test test4.c -Wall -ansi
nborie@perceval:~> ./test
top - 11:32:15 up 1 day, 23:33,  6 users,  load average: 0,12, 0,21, 0,35
Tasks: 201 total,   2 running, 199 sleeping,   0 stopped,   0 zombie
%Cpu(s): 1,1 us, 0,7 sy, 0,1 ni, 97,6 id, 0,4 wa, 0,0 hi, 0,0 si, 0,0 st

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
31039 nborie    20   0 4581m 4,5g  372 S   0,0 58,2   0:02.69 test
```

## Bonne allocation personnelle

```
1  #define SIZE_STRING 29
2  #define N_STRINGS 100000000
3  char memory[SIZE_STRING * N_STRINGS];
4
5  char* my_malloc(long int size){
6    static long int pos = 0;
7    if (pos + size >= SIZE_STRING * N_STRINGS) return NULL;
8    char* res = memory + pos; pos += size;
9    return res;
10 }
11
12 int main(int argc, char* argv[]){
13   int i; char* adr;
14   for (i = 0; i < N_STRINGS - 1; i++){
15     adr = my_malloc(SIZE_STRING); adr[0] = 'a'; adr[1] = 0;
16   }
17   getchar(); return 0;
18 }
```

```
  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM     TIME+  COMMAND
31710 nborie    20   0 2769m 2,7g  328 S   0,0 35,2   0:00.83 test
```



## Bien réallouer

- Si on doit utiliser `realloc`, il faut éviter de le faire trop souvent.
- Mauvais exemple :

```
1  void add_int(struct array* a, int value){
2    if(a->current == a->capacity){
3      a->capacity++;
4      a->data = (int*)realloc(a->data, a->capacity*sizeof(int));
5      if(a->data == NULL){
6        fprintf(stderr, "Not enough memory!\n");
7        exit(1);
8      }
9    }
10   a->data[a->current] = value;
11   a->current++;
12 }
```



## Bien réallouer

- Bonne conduite : doubler la taille lorsqu'elle devient insuffisante. On peut réajuster avec un dernier `realloc` si la mémoire est critique.

```
1  void add_int(struct array* a, int value){
2    if(a->current == a->capacity){
3      a->capacity *= 2;
4      a->data = (int*)realloc(a->data, a->capacity*sizeof(int));
5      if(a->data == NULL){
6        fprintf(stderr, "Not enough memory!\n");
7        exit(1);
8      }
9    }
10   a->data[a->current] = value;
11   a->current++;
12 }
```

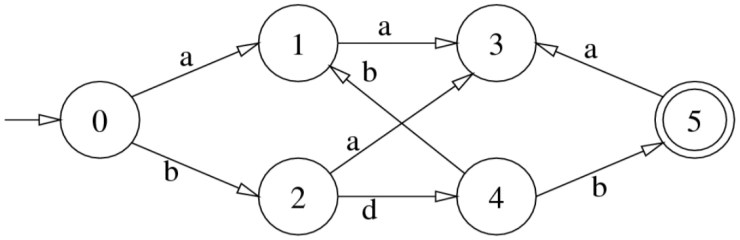


## Bien désallouer

- On ne doit jamais invoquer `free` plus d'une fois sur un même objet mémoire.
- D'où un problème : libération de mémoire manipulée avec plusieurs points d'entrées (plusieurs pointeurs différents).
- Solutions :
  - Ne pas le faire
  - Comptage de références
  - Table de pointeurs
  - Garbage collector

## Exemple symptomatique

- Comment manipuler un automate (ou un graphe orienté avec un nombre fixé d'arêtes sortants) ?






## Solution 1 : éviter le problème

- Chaque état est une structure.
- Automate = tableau de structures
- État = indice dans le tableau
- Une seule allocation/libération :
  - le tableau de structures
- Très bonne solution si on connais une borne du nombre d'états en avance
- En cas où il faut aussi supprimer des états, pour l'ajout il faut trouver une case vide. Cela se fait en $O(1)$ avec une structure auxiliaire.



## Solution 2 : comptage de référence

- Chaque état est une structure.
- On ajoute à cette structure un compteur.
- À chaque fois qu'on fait pointer une adresse sur un état, on augmente son compteur.
- À chaque fois qu'on déréférence un état, on décrémente son compteur et on le libère quand on atteint 0.



## Solution 2 : comptage de référence

- Lourd à mettre en oeuvre
- Risque d'erreur (oubli de mise à jour du compteur)
  - **Oubli d'incrémentation** ⇒ compteur sous évalué ⇒ libération prématurée ⇒ utilisation d'une zone libérée ⇒ **Segfault (dans le meilleur des cas)**
  - **Oubli de décrémentation** ⇒ compteur sur évalué ⇒ libération jamais atteinte ⇒ **fuite mémoire**
- Dangereux

## Solution 3 : table de pointeurs

- Chaque état est identifié par son indice
- Automate = tableau de pointeurs
- Pour accéder à l'état $n$, on passe par $tableau[n]$.
- Pour libérer l'état $n$, on libère $tableau[n]$ et on le met à `NULL`, s'il n'était pas déjà à `NULL`
- `realloc` si besoin
- À n'utiliser que si la solution 1 n'est pas possible (très rare cas...)



## Solution 4 : garbage collector

- Allocation explicite, mais pas de libération
- Tâche de fond ou périodique qui vérifie toute la mémoire
- Comment faire :

  - `https://www.hboehm.info/gc/`
  (A garbage collector for C and C++)

  - Changer de langage (Java, Python, ...)
- Objectif : nettoyer ce dont on n'a plus besoin

  - Définir et identifier ces choses (très lié au langage et classes de stockage)



## Manipulation de la mémoire

Le bon programmeur C doit savoir :

- Connaître les différentes opportunités mémoires du langage et leurs comportements (utilisation / localisation / durée de vie)
- Identifier le type de mémoire le plus adapté à son problème
- Allouer / désallouer proprement et efficacement tout type de mémoire
- Jongler avec les bits et leurs interprétations (jouer avec des `void*`)
- Propager l'objectif du langage de produire des exécutables efficaces



## Quelques conseils

- Ce cours est à lire et à relire calmement pour être digéré...
- Connaître les règles des échecs ne suffit pas pour savoir y jouer.
- Pour maîtriser un langage, connaître sa syntaxe et ses fonctions de bases ne suffit pas.
- C'est plus l'appréhension de ses paradigmes et la compréhension de son esprit qui fait le bon programmeur.
- S'il y a 1000 façon à résoudre un problème dans un langage, il n'en reste que 10 si on impose de respecter son esprit.