CM 12 - L'ALLOCATION DYNAMIQUE

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

RAPPELS

▷ Un pointeur = (adresse mémoire + type cible)

▷ Lors de la déclaration `type_cible* nom;` la variable `nom` contient l'adresse d'une zone mémoire contenant une valeur de type `type_cible`

▷ Le type cible peut être quelconque

▷ `NULL` représente une valeur spéciale symbolisant une adresse invalide

POINTEURS GÉNÉRIQUES

▷ Il existe une notion de pointeur générique en C noté `void*`

▷ Ce type de pointeur est compatible avec tous les autres pointeurs (i.e., à même de pointer vers n'importe quel type cible)

▷ ⚠Ce n'est pas un pointeur vers un objet de type `void` mais bien un type à part

▷ Il n'est pas possible de déréférencer un pointeur générique car sans type cible, il ne connaît pas la taille mémoire à récupérer

▷ On ne peut donc pas non plus appliquer d'arithmétique sur ce type de pointeur

POINTEURS GÉNÉRIQUES

▷ Pour utiliser un pointeur générique il faut le transtyper

▷ Par définition, il n'y a aucun problème de conversion depuis ou vers un pointeur générique

  ▷ `void*` ← `void*`, `foo*` ← `foo*`

  ▷ `void*` ← `foo*`, `foo*` ← `void*`

▷ Il y a en revanche un problème pour `bar*` ← `foo*`

▷ Si l'on souhaite interpréter la mémoire de façon différente, il faut une conversion explicite

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

EXEMPLE : ENDIANESS

▷ On peut constater et détecter l'ordre de rangement des octets en mémoire

▷ Il suffit d'interpréter un `int` comme un tableau d'octets

```
int endianness(){
    unsigned int i = 0x12345678;
    unsigned char* t = (unsigned char*)(&i);
    switch(t[0]){
        /*12 34 56 78*/
        case 0x12: return BIG_ENDIAN;
        /*78 56 34 12*/
        case 0x78: return LITTLE_ENDIAN;
        /*34 12 78 56*/
        case 0x34: return BIG_ENDIAN_SWAP;
        /*56 78 12 34*/
        default: return LITTLE_ENDIAN_SWAP;
    }
}
```

EXEMPLE : SHOWBYTES

▷ On peut visualiser les octets composant un `double`

```
void show_bytes(double d){
    unsigned char* t = (unsigned char*)(&d);
    int i;
    for(i=0;i<sizeof(double);i++){
        printf("%X",t[i]);
    }
    printf("\n");
}
```

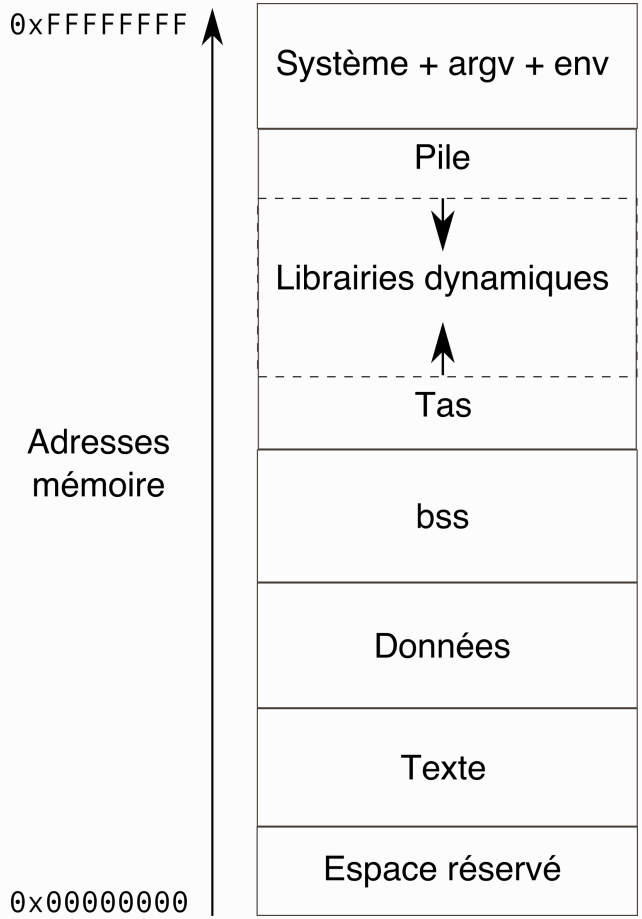
```
$>./a.out 375.57898541
FA 8C 34 86 43 79 77 40
```

LA ZONE MÉMOIRE TAS (heap)

▷ Le tas est un espace mémoire allouable dynamiquement par le programmeur

▷ Cette zone mémoire n'a pas de taille fixe et augmente/diminue en fonction des demandes du programmeur, qui peut réserver (allocation) ou supprimer (libération) des blocs mémoires

▷ Une variable stockée dans le tas est accessible partout dans le programme, par l'intermédiaire des pointeurs, jusqu'à sa libération



malloc

▷ L'obtention d'une adresse sur le tas s'effectue via l'appel de la fonction **`malloc`** de la librairie **`stdlib.h`**

  ▷ `void* malloc(size_t size);`

▷ Le paramètre **`size`** représente la taille en octets de la zone réclamée

▷ La fonction retourne l'adresse d'une zone au contenu indéfini de la taille demandée et la valeur spéciale `NULL` en cas d'échec

  ▷ ⚠Il faut donc penser à initialiser la zone allouée

▷ Notons que la fonction **`malloc`** renvoie un pointeur générique

Règles d'or du `malloc`

▷ Il faut toujours tester le retour de la fonction **`malloc`** pour détecter les cas d'erreurs d'allocations

▷ Il faut toujours fournir une taille qui soit portable en utilisant la fonction **`sizeof()`** que l'on multiplie par le nombre d'éléments du type concerné

▷ Il faut toujours convertir le pointeur générique explicitement pour indiquer le type cible (bien que facultatif)

```
foo* c = (foo*) malloc(2*sizeof(foo));
if(c==NULL){
    fprintf(stderr,"Not enough memory!\n");
    exit(EXIT_FAILURE);
}
...
```

ALLOCATION DE TABLEAUX SUR LE TAS

▷ Pour allouer un tableau de k éléments d'un type donné, il suffit d'allouer une zone mémoire de taille égale à la taille d'un élément de ce type multipliée par k

```
int* create_array(int size){
    int* array = (int*) malloc(size*sizeof(int));
    if(array==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    return array;
}
```

▷ Qu'est-ce qui est retournée par la fonction ?

calloc

- La librairie `stdlib.h` fournit une fonction alternative
  - `void* calloc(size_t count, size_t size);`
- Le paramètre `count` représente un nombre d'éléments et `size` la taille en octets de chaque élément
- La fonction retourne l'adresse d'une zone remplie de 0 de la taille demandée ou `NULL` en cas d'échec

```
int* create_array(int size){
    int* array = (int*) calloc(size,sizeof(int));
    if(array==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    return array;
}
```

realloc

▷ La librairie `stdlib.h` fournit également une fonction permettant de modifier la taille d'une zone mémoire allouée sur le tas

  ▷ `void* realloc(void* ptr, size_t size);`

▷ Le paramètre `ptr` doit représenter une adresse valide sur le tas acquise par l'appel d'une des fonctions d'allocations et `size` la nouvelle taille désirée en octets

▷ La fonction retourne l'adresse d'une zone de la nouvelle taille demandée ou `NULL` en cas d'échec

  ▷ Les anciennes données sont conservées

  ▷ Ou tronquées si la taille a diminuée

▷ ⚠ Il y a donc une possible copie des données

EXEMPLE DE RÉALLOCATION

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

/*Adds the given value to the given array,
 *enlarging it if needed*/
void add_int(struct array* a, int value){
  if((*a).current==(*a).capacity){
    (*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(EXIT_FAILURE);
    }
  }
  (*a).data[(*a).current] = value;
  ((*a).current)++;
}
```

LIBÉRATION DE LA MÉMOIRE

▷ La libération d'une zone mémoire sur le tas s'effectue via l'appel de la fonction `free` de la librairie `stdlib.h`

  ▷ `void free(void* ptr);`

▷ Le paramètre `ptr` doit représenter une adresse valide sur la tas acquise par l'appel d'une des fonctions d'allocations ou `NULL`

▷ La fonction libère la zone pointée par `ptr` (pas d'effet si `NULL`)

▷ ⚠En cas de valeur différente de `NULL` pour `ptr`, la zone mémoire ne doit pas déjà avoir été libérée

Règles d'or du `free`

- Il ne faut jamais essayer de déréférencer un pointeur sur une zone libérée (comportement indéfini - pas forcément une erreur)
- Il convient donc de mettre à `NULL` un pointeur après l'appel de la fonction **`free`** sur ce dernier
  - ⚠ Cela ne règle pas tous les problèmes possibles
- 1 **`malloc`** = 1 **`free`**
- Celui qui alloue est celui qui libère

RÈGLES D'OR DU free

```c
#define N 10
void foo(int* t){
    /*...*/
    free(t);
}
int main(void){
    int* t;
    t = (int*) malloc(N*sizeof(int));
    foo(t);
    /*...*/
    return EXIT_SUCESS;
}
```

⚠

```c
#define N 10
void foo(int* t){
    /*...*/
}
int main(void){
    int* t;
    t = (int*) malloc(N*sizeof(int));
    foo(t);
    free(t);
    t=NULL;
    /*...*/
    return EXIT_SUCESS;
}
```

ALLOCATION DE STRUCTURES

▷ Pour une structure, il est souvent préférable de définir une fonction d'allocation/initialisation et une fonction de libération

```
typedef struct{
  double real;
  double imaginary;
} Complex;
Complex* newComplex(double r, double i){
  Complex* c= (Complex*) malloc(sizeof(Complex));
  if(c==NULL){
    fprintf(stderr,"Not enough memory!\n");
    exit(EXIT_FAILURE);
  }
  (*c).real=r;
  (*c).imaginary=i;
  return c;
}
void freeComplex(Complex* c){
  free(c);
}
```

```
void foo(Complex* c){
  /*...*/
}

int main(void){
  Complex* c;
  c=newComplex(10.4, 12.0);
  foo(c);
  freeComplex(c);
  c=NULL;
  /*...*/
  return EXIT_SUCESS;
}
```

ALLOCATION DYNAMIQUE

▷ L'allocation dynamique est coûteuse en temps et en espace

▷ En théorie `malloc` obtient la mémoire depuis le système d'exploitation

▷ Le système manipule des pages (souvent de l'ordre de 4ko) et vous avez le droit de demander d'allouer moins ...

▷ C'est `malloc` qui gère de manière interne les pages allouées par le système pour savoir ce qui est libre ou alloué à un temps donné

▷ Ceci nécessite de stocker des informations supplémentaires dans la mémoire et peut prendre du temps à calculer

ALLOCATION DYNAMIQUE

```c
#include <stdlib.h>
int main(void){
    int i=0;
    for(i=0;i<100000000;i++){
        malloc(4);
    }
    while(1){
    }
}
```

▷ 100M allocations de 4o

```c
#include <stdlib.h>
int main(void){
    int i=0;
    malloc(400000000);
    while(1){
    }
}
```

▷ Allocation de 400 Mo

```
  PID USER  PR  NI    VIRT    RES  SHR S  %CPU %MEM     TIME+ COMMAND
34812 gblin 20   0 1564620 1,491g  800 R  99,7  1,2   1:52.82 mem
34813 gblin 20   0  392632    536  484 R  99,7  0,0   1:52.82 mem2
```

ALLOCATION DYNAMIQUE

▷ L'allocation dynamique est donc à utiliser avec discernement;
e.g. pas quand la taille d'un tableau est connue

```c
#include <stdlib.h>
int main(void){
    Complex* a=newComplex(2,3);
    Complex* b=newComplex(7,4);
    Complex* c=addComplex(a,b);
    /*...*/
    freeComplex(a);
    freeComplex(b);
    freeComplex(c);
    return EXIT_SUCCESS;
}
```

```c
#include <stdlib.h>
int main(void){
    Complex a={2,3};
    Complex b={7,4};
    Complex* c=addComplex(&a,&b);
    /*...*/
    freeComplex(c);
    return EXIT_SUCCESS;
}
```

Allocation dynamique d'un tableau à 2 dimensions

▷ Un tableau à 2 dimensions correspond à un tableau de tableaux

▷ Il faut allouer le tableau principal puis allouer chacun des sous-tableaux (on peut étendre ce principe à n dimensions)

```
int** initArray(int X, int Y){
    int** t = (int**) malloc(X*sizeof(int*));
    if(t==NULL){
        /*...*/
    }
    int i;
    for(i=0; i<X; i++){
        t[i] = (int*) malloc(Y*sizeof(int));
        if(t[i]==NULL){
            /*...*/
        }
    }
    return t;
}
```

ALLOCATION DYNAMIQUE D'UN TABLEAU À 2 DIMENSIONS

▷ A la différence des tableaux statiques, les éléments ne sont pas forcément tous contigus et les sous-tableaux ne sont pas forcément dans l'ordre

```
#include <stdlib.h>
#include <stdio.h>
int main(void){
    int** t = initArray(2,3);
    printf("t[0] - %p : %p - ", &t[0], t[0]);
    printf("t[1] - %p : %p\n", &t[1], t[1]);
    for(int i=0;i<6;i++){
       printf("t[%d][%d] - %p - ",i/3, i%3, &t[i/3][i%3]);
    }
    return EXIT_SUCCESS;
}
```

```
$> ./a.out
t[0] - 0x9a9a008 : 0x9a9a048 - t[1] - 0x9a9a00c : 0x9a9a018
t[0][0] - 0x9a9a048 - t[0][1] - 0x9a9a04c - t[0][2] - 0x9a9a050 -
t[1][0] - 0x9a9a018 - t[1][1] - 0x9a9a01c - t[1][2] - 0x9a9a020 -
```

ALLOCATION DYNAMIQUE D'UN TABLEAU À 2 DIMENSIONS

▷ Il est possible de ne faire qu'un seul malloc et gérer à la main la localisation des sous-tableaux (`t[i][j]` n'est plus utilisable)

```
int* initArray(int X, int Y){
    int* t = (int*) malloc(X*Y*sizeof(int));
    if(t==NULL){ /*...*/}
    return t;
}
int main(void){
    int* t = initArray(2,3);
    for(int i=0;i<6;i++){
       printf("t[%d][%d] - %p - ",i/3, i%3, &t[i]);
    }
    return EXIT_SUCCESS;
}
```

```
$> ./a.out
t[0][0] - 0x8456008 - t[0][1] - 0x845600c - t[0][2] - 0x8456010 -
t[1][0] - 0x8456014 - t[1][1] - 0x8456018 - t[1][2] - 0x845601c -
```

Libération d'un tableau à 2 dimensions

▷ Il faut libérer les sous-tableaux avant de pouvoir libérer le tableau principal (sinon on perd l'adresse des sous-tableaux qui deviennent non libérables)

```
void freeArray(int** t, int X){
    if(t==NULL){
        return;
    }
    int i;
    for(i=0;i<X;i++){
        if(t[i]!=NULL){
            free(t[i]);
        }
    }
    free(t);
}
```

LISTES CHAÎNÉES

▷ Grâce à l'allocation dynamique, on peut mettre en oeuvre des listes arbitrairement longues

```
typedef struct cell{
    int value;
    struct cell* next;
}Cell;
void print(Cell* list){
    while(list!=NULL){
        printf("%d\n",(*list).value);
        list=(*list).next;
    }
}
Cell* addInFront(Cell* list, int value){
    Cell* ncell = (Cell*) malloc(sizeof(Cell));
    if(ncell==NULL){ /*...*/}
    (*ncell).value=value;
    (*ncell).next=list;
    return ncell;
}
```

LISTES CHAÎNÉES

▷ Liste est représentée par l'adresse de son premier élément

```
int main(void){
    Cell* l = NULL;
    l = addInFront(addInFront(addInFront(NULL,15),7),3);
    while(l!=NULL){
        printf("%p:%d-%p ",l,(*l).value,(*l).next);
        l=(*l).next;
    }
    return EXIT_SUCCESS;
}
```

```
$> ./a.out
0x9724028:3-0x9724018 0x9724018:7-0x9724008 0x9724008:15-(nil)
```

BIEN ALLOUER

▷ L'allocation n'est pas forcément synonyme d'appel à la fonction `malloc`, qui a un coût mémoire

▷ Il ne faut pas utiliser `malloc` quand

- ▷ On a plein de petits objets ($<$ 32 octets)
- ▷ Plein d'allocations mais peu ou pas de désallocations

▷ Il est possible de gérer sa propre mémoire avec un tableau d'octets par exemple

MAUVAISE ALLOCATION

▷ Un ensemble de **malloc** s'accumulant - 88072 ko

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE_STRING 84
#define N_STRINGS 1000000
int main(void){
    int i;
    for(i=0;i<N_STRINGS;i++){
        malloc(SIZE_STRING);
    }
    getchar();
    return EXIT_SUCCESS;
}
```

```
PID   USER  PR  NI VIRT  RES    SHR S  %CPU %MEM     TIME+ COMMAND
43524 gblin  20 0  88072 86628  800 T  26,3  0,1   0:00.08 a.out
```

BONNE ALLOCATION PERSONNELLE

▷ Un seul `malloc` de 84032 ko (soit une économie de 4Mo !)

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE_STRING 84
#define MAX N_STRINGS*SIZE_STRING
char memory[MAX];
int pos=0;
char* myAlloc(int size){
    if(pos+size>=MAX){
        return NULL;
    }
    char* res=&(memory[pos]);
    pos=pos+size;
    return res;
}
```

```c
int main(void){
    int i;
    for(i=0;i<N_STRINGS;i++){
        myAlloc(SIZE_STRING);
    }
    getchar();
    return EXIT_SUCCESS;
}
```

```
PID   USER  PR  NI VIRT  RES    SHR S  %CPU %MEM     TIME+ COMMAND
45346 gblin 20  0  84032 536    484 R  25,7  0,0    0:07.80 a.out
```

BONNE ALLOCATION PERSONNELLE

▷ L'économie est d'autant plus grande que la taille des objets allouées est petite

▷ En définissant `SIZE_STRINGS` à `8` (`#define SIZE_STRING 8`)

```
PID   USER  PR  NI VIRT  RES    SHR S  %CPU %MEM     TIME+ COMMAND
7250 gblin  20  0  25524 16472  876 R 26,0  0,0   0:00.23 a.out
47074 gblin 20  0  9816  564    516 R 24,0  0,0   0:00.23 a.out
```

▷ Soit une économie de 15Mo !

BIEN RÉALLOUER

▷ De manière similaire, la réallocation (à l'aide de `realloc`) doit se faire de manière parcimonieuse

▷ Mauvais exemple

```
/*Adds the given value to the given array,
 *enlarging it if needed*/
void addInt(struct array* a, int value){
  if((*a).current==(*a).capacity){
    (*a).capacity=(*a).capacity+1; // OUPS
    (*a).data=(int*) realloc((*a).data,(*a).capacity*sizeof(int));
    if((*a).data==NULL){
      fprintf(stderr,"Not enough memory!\n");
      exit(EXIT_FAILURE);
    }
  }
  (*a).data[(*a).current]=value;
  ((*a).current)++;
}
```

BIEN RÉALLOUER

▷ La bonne consuite est de doubler la taille, quitte à la réajuster quand on a fini de remplir le tableau

```
/*Adds the given value to the given array,
 *enlarging it if needed*/
void addInt(struct array* a, int value){
  if((*a).current==(*a).capacity){
    (*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(EXIT_FAILURE);
    }
  }
  (*a).data[(*a).current]=value;
  ((*a).current)++;
}
```

BIEN DÉSALLOUER

▷ ⚠On ne doit jamais invoquer `free` plus d'une fois sur un même objet mémoire

▷ Cela peut poser un problème de libération si on a plusieurs pointeurs sur un même objet

▷ Les solutions sont

  ▷ Ne pas le faire

  ▷ Compter les références actives (ne libérer la mémoire que si ce compteur atteint 0)

  ▷ Utiliser un table de pointeurs (on transmet l'indice du pointeur comme référence)

  ▷ Un système de garbage collector (tâche de fond périodique qui vérifie la mémoire ; c.f. java)

BIEN DÉSALLOUER

▷ Tant que l'on a pas indiqué à `malloc` qu'une zone mémoire n'est pas réutilisable, il ne va pas sans servir pour de nouvelles allocations

▷ Si à un moment il n'existe plus de variable dans la pile (ou statique) qui pointe directement ou indirectement vers une zone acquise par allocation dynamique alors il y a une fuite mémoire (memory leak)

DOGGY BAG

TO TAKE AWAY ...

▷ Toute variable est soit dans la pile soit dans le segment de donnée

▷ En aucun cas une variable de votre programme peut avoir une adresse appartenant au tas

▷ L'utilisation de l'allocation dynamique impose l'utilisation de pointeurs permettant de gérer les adresses mémoires

▷ `malloc` n'a aucune idée du type de données stockées ; il est important d'interpréter ces zones correctement

Questions?