Cours de C

Allocation dynamique

Sébastien Paumier

Les pointeurs

- pointeur=adresse mémoire+type
- `type* nom;` ⇒ `nom` pointe sur une zone mémoire correspondant au `type` donné
- le type peut être quelconque
- valeur spéciale `NULL` équivalente à 0
- pointeur générique: `void* nom;`

Arithmétique des pointeurs

- `int* t` ⇒ on peut voir `t` comme un tableau d'`int`

  `*t` ⇔ `t[0]`

- l'addition et la soustraction se font en fonction de la taille des éléments du tableau

  t+3 = t+3*sizeof(type des éléments)

  `*(t+i)` ⇔ `t[i]`

Arithmétique des pointeurs

- si le pointeur **`t`** pointe sur la case **`x`** du tableau, on peut le décaler sur la case **`x+1`** en faisant **`t++`**

```
int copy(char* src,char* dst) {
    if (src==NULL || dst==NULL) {
        return 0;
    }
    while ((*dst=*src)!='\0') {
        src++;
        dst++;
    }
    return 1;
}
```

Transtypage

- pas de problème de conversion pour:

  `void* ← void*` `truc* ← truc*`

  `void* ← truc*` `truc* ← void*`

- problème pour `truc* ← biniou*`
- on peut chercher à voir la mémoire de façon différente, mais il faut une conversion explicite:

```
biniou* b=...;
truc* t=(truc*)b;
```

Exemple

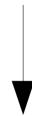
- ordre de rangement des octets en mémoire
  - ⇒ voir un `int` comme un tableau de 4 octets

```
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 2

- voir les octets qui composent 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
```

Allocation dynamique

- principe: demander une zone mémoire au système
- zone représentée par son adresse
- zone prise sur le tas
- zone persistante jusqu'à ce qu'elle soit libérée (≠ variables locales)

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 ou l'adresse d'une zone au contenu indéfini sinon

  ⇨ penser à initialiser la zone

Règles d'or de malloc

- toujours tester le retour de malloc
- toujours multiplier le nombre d'éléments par la taille
- toujours mettre un cast pour indiquer le type (facultatif, mais plus lisible)
- usage prototypique:

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

Allocation de tableaux

- il suffit de multiplier par le nombre d'éléments:

```
int* create_array(int size) {
   int* array;
   array=(int*)malloc(size*sizeof(int));
   if (array==NULL) {
      fprintf(stderr,"Not enough memory!\n");
      exit(1);
   }
   return array;
}
```

calloc

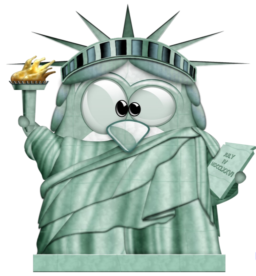
- `void* calloc(size_t nmemb,size_t size);`
- alloue et remplit la zone de zéros:

```
int* create_array(int size) {
   int* array;
   array=(int*)calloc(size,sizeof(int));
   if (array==NULL) {
      fprintf(stderr,"Not enough memory!\n");
      exit(1);
   }
   return array;
}
```

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
- possible copie de données sous-jacente
- `ptr` doit pointer sur une zone valide!

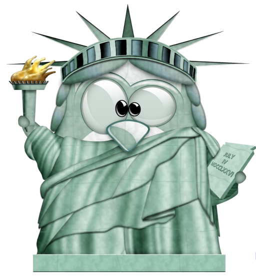
paumier@univ-mlv.fr



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(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)
- `ptr` doit pointer sur une zone valide obtenue avec `malloc`, `calloc` ou `realloc`
- la zone ne doit pas déjà avoir été libérée

Libération de la mémoire

- ne JAMAIS lire un pointeur sur une zone libérée ⚡
- attention aux allocations cachées:

```
char* strdup(const char* s);
```
(string.h)

Libération de la mémoire

- 1 malloc = 1 free
- celui qui alloue est celui qui libère:

```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;
}
```

```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;
}
```

Allocation de structures

- mieux vaut faire une fonction d'allocation et d'initialisation et une fonction de libération

```
typedef struct {
   double real;
   double 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 d'un tableau est connue par exemple)

```
int main(int argc,char* argv[]) {
Complex* a=new_Complex(2,3);
Complex* b=new_Complex(7,4);
Complex* c=mult_Complex(a,b);
/* ... */
free_Complex(a);
free_Complex(b);
free_Complex(c);
return 0;
}
```

```
int main(int argc,char* argv[]) {
Complex a={2,3};
Complex b={7,4};
Complex* c=mult_Complex(&a,&b);
/* ... */
free_Complex(c);
return 0;
}
```

Tableaux à 2 dimensions

- = tableau de tableaux:
  - allouer le tableau principal
  - allouer chacun des sous-tableaux

```
int** init_array(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;
}
```

- on peut étendre à *n* dimensions

Tableaux à 2 dimensions

- les éléments ne sont pas tous contigus!

  ≠ tableaux statiques

- les sous-tableaux ne sont pas forcément dans l'ordre

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

	43B2			44F8				4532			
	4532	44F8	...	7	7	7	...	7	7	7	

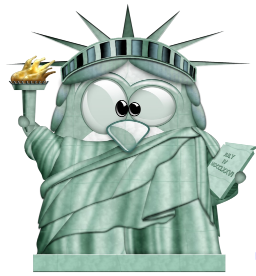
- on pourrait ne faire qu'un seul `malloc`

Tableaux à 2 dimensions

- libération:
  - d'abord les sous-tableaux
  - puis le tableau principal

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

- on peut étendre à *n* dimensions



Listes chaînées

- chaque cellule pointe sur la suivante
- grâce à l'allocation dynamique, on peut avoir des listes arbitrairement longues

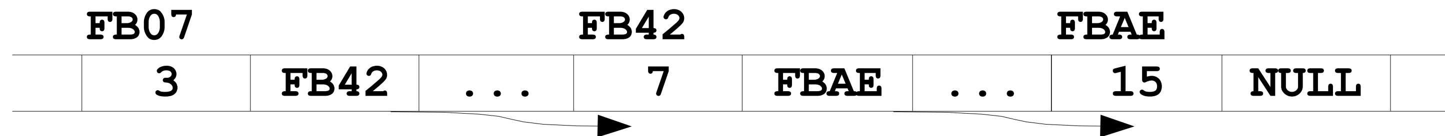
```
typedef struct cell {
   float value;
   struct cell* next;
} Cell;


void print(Cell* list) {
   while (list!=NULL) {
      printf("%f\n",list->value);
      list=list->next;
   }
}
```

double définition pour pouvoir:
1) avoir un type récursif
2) avoir un nom de type sans **`struct`**

Listes chaînées

- représentation en mémoire de la liste: 3 7 15



- liste=adresse de son premier élément (ici FB07)

Complexités

- attention aux complexités cachées de `malloc` et `free` (et de toutes les autres fonctions) ⚡
- si on a une fonction linéaire sur des listes mais que les `malloc` sont en n^2, on n'aura pas un temps d'exécution linéaire

Qui est n ?

Bien allouer

- allocation n'est pas forcément synonyme de `malloc`, qui a un coût mémoire
- ne pas utiliser `malloc` quand:
  - plein de petits objets (<32 octets)
  - plein d'allocations, mais pas de désallocation
- exemple de solution: gérer sa propre mémoire avec un tableau d'octets

Mauvais malloc

plein de malloc qui se cumulent
=87572Ko

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

#define SIZE_STRING 84
#define N_STRINGS 1000000

int main(int argc,char* argv[]) {
   int i;
   for (i=0;i<N_STRINGS;i++) {
      malloc(SIZE_STRING);
   }
   getchar();
   return 0;
}
```

```
$>./a.out&
$>top
  PID USER        VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19929 paumier    87572  84m  260 T  0.0  9.6  0:00.20  a.out
```

Bonne allocation personnelle

utilisation d'un tableau
=83540Ko

économie
=87572-83540
=4032Ko
≈4Mo

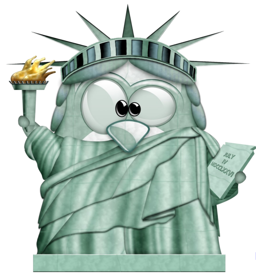
```
$>./a.out&
$>top
  PID USER          VIRT  RES  SHR
19952 paumier      83540  300  252
```

```
#define SIZE_STRING 84
#define N_STRINGS 1000000
#define MAX N_STRINGS*SIZE_STRING

char memory[MAX];
int pos=0;

char* my_alloc(int size) {
   if (pos+size>=MAX) return NULL;
   char* res=&(memory[pos]);
   pos=pos+size;
   return res;
}

int main(int argc,char* argv[]) {
   int i;
   for (i=0;i<N_STRINGS;i++) {
      my_alloc(SIZE_STRING);
   }
   printf("%d\n",pos);
   getchar();
   return 0;
}
```

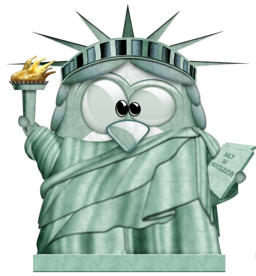


Bonne allocation personnelle

- même comparaison avec des objets de 8 octets au lieu de 84 (par exemple, une structure avec 2 champs entiers)

```
$>top
  PID USER        VIRT   RES   SHR S %CPU %MEM    TIME+  COMMAND
21944 paumier    17216  15m   260 T  0.0  1.8   0:00.08 a.out
21999 paumier     9324  300   252 T  0.0  0.0   0:00.01 a.out
```

- économie=17218-9324=7894Ko≈7,7Mo



Bien réallouer

- si on doit utiliser **realloc**, il faut éviter de le faire trop souvent
- <u>mauvais exemple:</u>

```
/* 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+1;
   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)++;
}
```

Bien réallouer

- bonne conduite: doubler la taille, quitte à la réajuster quand on a fini de remplir le tableau

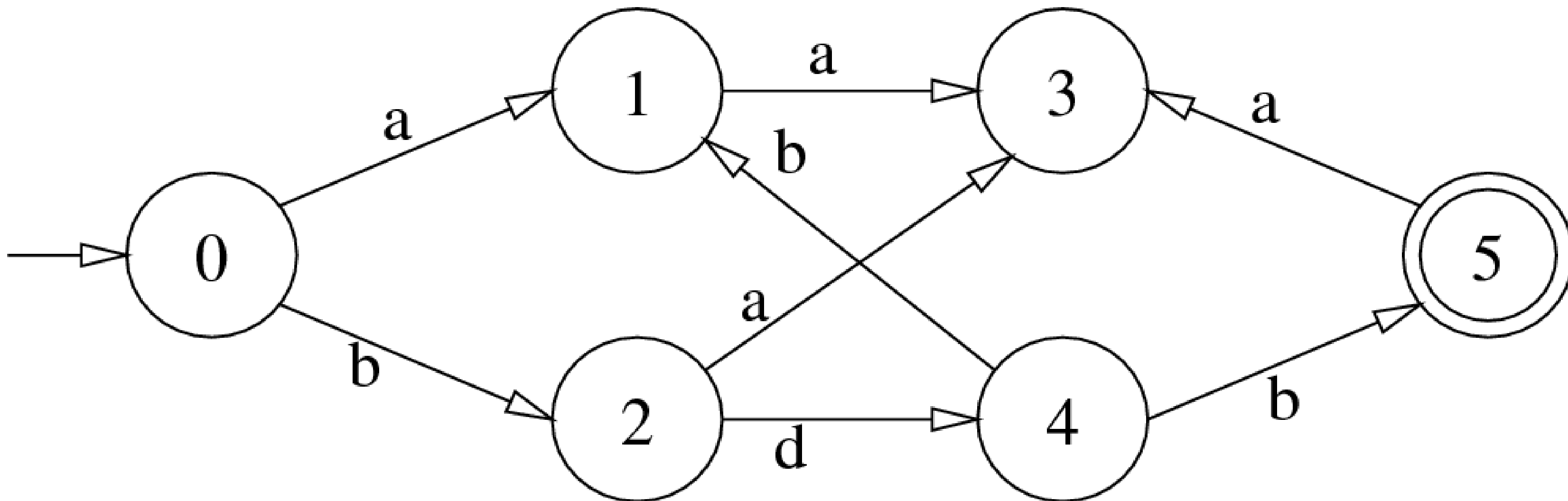
```
/* Adds the given value to the given array,
 * enlarging it if needed */
void add_int(struct array* a,int value) {
if (a->current==a->capacity) {
   a->capacity=a->capacity*2;
   a->data=(int*)realloc(a->data,a->capacity*sizeof(int));
   if (a->data==NULL) {
      fprintf(stderr,"Not enough memory!\n");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

Bien désallouer

- on ne doit jamais invoquer `free` plus d'une fois sur un même objet mémoire ⚡
- d'où: problème de libération si on a plusieurs pointeurs sur un même objet
- solutions:
  - ne pas le faire
  - comptage de référence
  - table de pointeurs
  - garbage collector

Exemple problématique

- comment représenter un automate ?



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
- <u>très bonne solution</u>

Solution 2: comptage de réf.

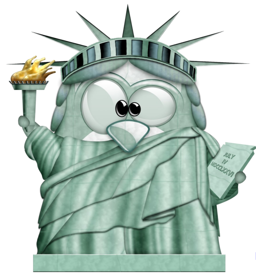
- chaque état est une structure
- ajouter à cette structure un compteur
- à chaque fois qu'on fait pointer une adresse sur un état, on augmente son compteur
- à chaque fois qu'on déréférence un état, on décrémente son compteur et on le libère quand on atteint 0

Solution 2: comptage de réf.

- lourd à mettre en œuvre
- risque d'erreur (oubli de mise à jour du compteur)
- à éviter

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`**
- à n'utiliser que si la solution 1 n'est pas possible (c'est rare)



Solution 4: garbage collector

- allocation explicite, mais pas de libération
- tâche de fond ou périodique qui vérifie toute la mémoire
- comment faire:
  - http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html
  - faire du Java :)