Programmation C

Allocation dynamique

ESIPE Info 1ère année

Université de Marne-la-Vallée

Les pointeurs

- pointeur = adresse mémoire + type
- `type* nom;` $\Rightarrow$ `nom` pointe vers une zone mémoire correspondant au `type` donné
- le type peut être quelconque (C doit juste être capable d'évaluer son `sizeof`)
- valeur spéciale `NULL` équivalente à 0. (`NULL` est juste 0 casté en `void*`, c'est l'adresse 0 en hexadécimale)
- pointeur générique : `void* nom;`

Arithmétique des pointeurs

- `int* t` $\Rightarrow$ on peut voir `t` comme un tableau d'`int`

  `*t` $\Leftrightarrow$ `&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 de t))`

  `*(t+i)` $\Leftrightarrow$ `t[i]`

- Le compilateur ne connaissant pas la taille d'un `void*`, il est nécessaire de caster avant de faire de l'arithmétique.

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++`

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

Transtypage

- pas de problème de conversion pour :

  `void*` ← `void*` `foo*` ← `foo*`

  `void*` ← `foo*` `foo*` ← `void*`

- problème pour `bar*` ← `foo*`

- On peut chercher à voir 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

- ordre de rangement des octets en mémoire

  $\Rightarrow$ 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

- Des données (forcément binaires) mémoires sont complètement utilisables lorsque l'on connaît :
  - 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 $\Leftrightarrow$ choisir une interprétation)

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 explicitement par le programme (≠ 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

  $\Rightarrow$ penser à initialiser la zone avant utilisation !

Règles d'or de malloc

- toujours tester le retour de malloc
- toujours multiplier le nombre d'éléments pas la taille
- 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  }
```

Allocation de tableaux

- il suffit de multiplier par le nombre d'éléments

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

Calloc

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

```
1  int create_array(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  }
```

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 !

exemple de réallocation

```
 1  struct array{
 2    int* data;
 3    int capacity;
 4    int current;
 5  }
 6
 7  /* Add a given value to a given array
 8     enlarging it if needed */
 9  void add_int(struct array* a, int value){
10    if (a->current == a->capacity){
11      a->capacity=a->capacity*2;
12      a->data=(int*)realloc(a->data, a->capacity*sizeof(int));
13      if (a->data == NULL){
14        fprintf(stderr, "Not enough memory!\n");
15        exit(1);
16      }
17    }
18    a->data[a->current] = value;
19    a->current++;
20  }
```

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);
```

Libération de la mémoire

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

Pas bien !

```
 1  void foo(int* t){
 2    /* ... */
 3    free(t);
 4  }
 5
 6  int main(int argc, char* argv[]){
 7    int* t;
 8    t=(int*)malloc(N*sizeof(int));
 9    foo(t);
10    return 0;
11  }
```

Mieux !

```
 1  void foo(int* t){
 2    /* ... */
 3  }
 4
 5  int main(int argc, char* argv[]){
 6    int* t;
 7    t=(int*)malloc(N*sizeof(int));
 8    foo(t);
 9    free(t);
10    return 0;
11  }
```

Allocations de structures

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

```
1  typedef struct{
2    double real, imaginary;
3  }Complex
4
5  Complex* new_complex(double r, double i){
6    Complex* c=(Complex *)malloc(sizeof(Complex));
7    if (c == NULL){
8      fprintf(stderr, "Not enough memory!\n");
9      exit(1);
10   }
11   c->real=r;
12   c->imaginary=i;
13   return c;
14 }
15
16 void free_complex(Complex* c){
17   free(c);
18 }
```

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...)

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 à 2 dimensions

- = tableau de tableaux :

  - allouer le tableau principal (qui est rempli de pointeurs vers les sous tableaux)

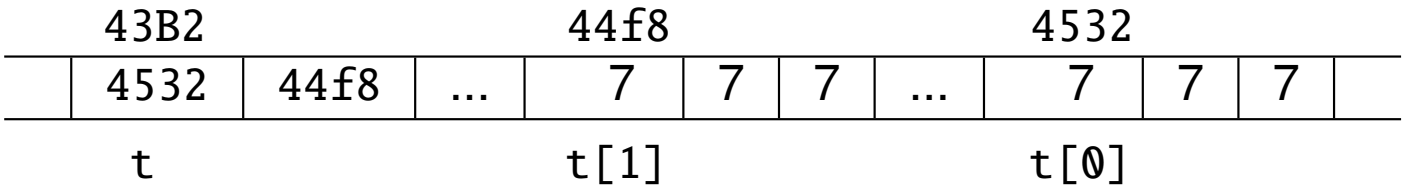
  - allouer chacun des sous-tableaux

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

- On peut étendre à *n* dimensions

Tableaux à 2 dimensions

- les éléments ne sont pas toujours contigus !
  $\neq$ tableaux statiques
- Les sous-tableaux ne sont pas forcément dans l'ordre
  `int** t de 2x3 contenant la valeur 7`



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

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        if (t[i]!=NULL)
6           free(t[i]);
7     }
8     free(t);
9  }
```

- on peut encore étendre à n dimensions

Listes chaînées

- chaque cellule pointe sur la suivante
- grâce à l'allocation dynamique, 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 }
```

- Pour un type récursif (qui s'utilise lui même sous la forme d'un pointeur), on ne peut pas faire de structure anonyme.

Listes chaînées

- représentation en mémoire de la liste : 3 7 15

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

- 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 les listes mais que les `malloc` sont en n^2, on n'aura pas un temps d'exécution linéaire

  encore faut-il donner un sens à ce n...

Bien allouer

- allocation n'est pas forcément synonyme de `malloc`, qui présente un surcoût 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

```
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,
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,

  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
 8461 nborie    20   0 1651m 537m  35m S   2,8  6,8  45:58.47 firefox
```

Bonne allocation personnelle

```
 1  #define SIZE_STRING 29
 2  #define N_STRINGS 100000000
 3  char memory[2900000000];
 4  long int pos=0;
 5
 6  char* my_malloc(long int size){
 7    if (pos+size>=2900000000) return NULL;
 8    char* res=&(memory[pos]); 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';
16    }
17    getchar();
18    return 0;
19  }
```

```
  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=a->capacity+1;
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.

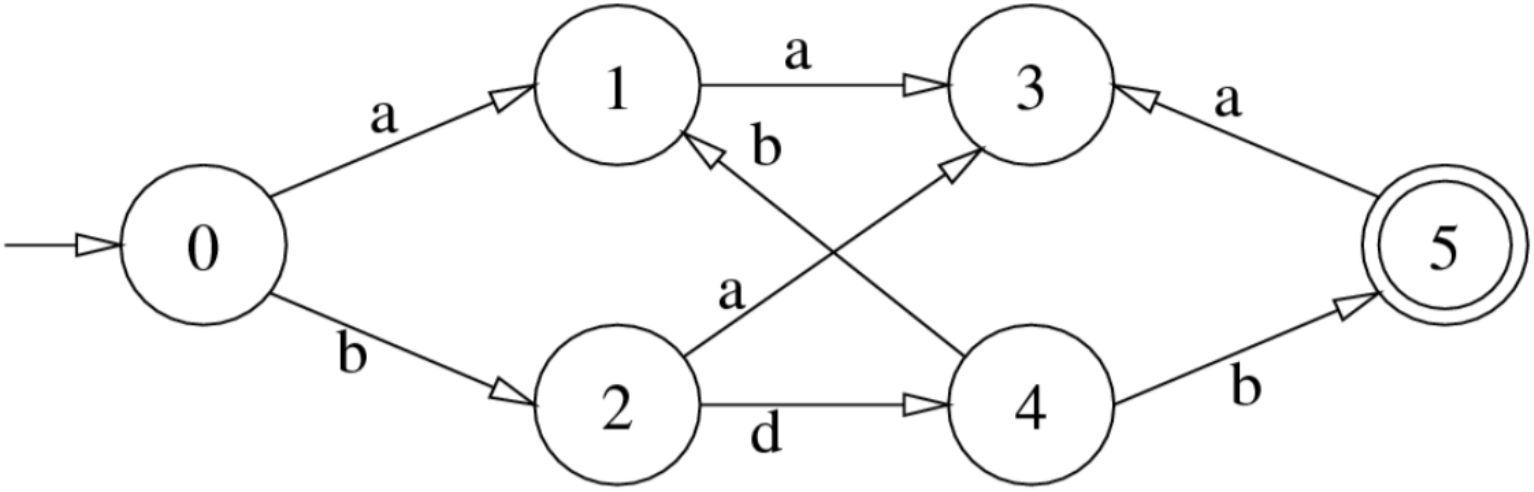
```
1  void add_int(struct array* a, int value){
2    if (a->current == a->capacity){
3      a->capacity=a->capacity*2;
4      a->data=(int*)realloc(a->data, a->capacity*sizeof(int));
5      if (a->data == NULL){
6        fprintf(stderr, "Not enough memory!\n");
7        exit(1);
8      }
9    }
10 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



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

Solution 2 : comptage de réf.

- 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 réf.

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

  - **oubli d'incrémentation** $\Rightarrow$ compteur sous évalué $\Rightarrow$ libération prématurée $\Rightarrow$ utilisation d'une zone libérée $\Rightarrow$ **Segfault (dans le meilleur des cas)**

  - **oubli de décrémentation** $\Rightarrow$ compteur sur évalué $\Rightarrow$ libération jamais atteinte $\Rightarrow$ **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`
- 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 :

  - http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (A garbage collector for C and C++)

  - changer de langage (Java, Python, ...)
- objectif : nettoyer ce dont on 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