IN3R11-2 – C

Cours 3

Damien MASSON
d.masson@esiee.fr

http://esiee.fr/~massond/Teaching/IN3R11-2/

24 novembre 2011

Le retour des 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;`
- l'opérateur & donne l'adresse d'une variable
- &*t == t puisque l'adresse de la variable pointée par t... est t

```c
#include <stdio.h>
int main(int argc, char* argv[]){
        int i;
        int * t = &i;
        printf("%d\n",&*t==t);/* 1 */
}
```

Passage par adresse

```c
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main(int argc, char ** argv)
{
  int a = 5,b = 10;
  swapt(a,b);
  printf("a = %d, b = %d",a,b);
  /* a = 5, b = 10 */
}
```

```c
void swap(int * a, int * b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char ** argv)
{
  int a = 5,b = 10;
  swapt(&a,&b);
  printf("a = %d, b = %d",a,b);
  /* a = 10, b = 5 */
}
```

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

```
double t[10];
printf("%d\n",t+2==(void*)t+2*sizeof(double)); /* 1 */
printf("%d\n",*(t+8)==t[8]); /* 1 */
```

- l'opérateur ++ fonctionne sur les pointeurs :

```
int tab[]={1,2};
int * t = tab;
printf("%d\n",t[0]); /* 1 */
t++;
printf("%d\n",t[0]); /* 2 */
```

Structures et pointeurs

- une structure (resp une union ou une enum) peut avoir des champs de type pointeurs
- accès au champ par `nom_struct.nom_champ`
- accès à la valeur pointée par `*(nom_struct.nom_champ)`
- à ne pas confondre avec `(*nom_struct).nom_champ`
- qui s'écrit aussi `nom_struct->nom_champ`
- qui permet d'accéder au champs d'une structure dont on a l'adresse

```c
#include <stdio.h>
typedef struct intArray{
    int* data;
    int capacity;
    int current;
} IntArray;
void printIntArray(IntArray * array){
  int i;
  for(i=0;i<(*array).current;i++){
    printf("%d␣",*((*array).data)+i); /* !!!! */
    printf("%d␣",*((array->data)+i)); /* !!   */
    printf("%d␣",array->data[i]);     /* :)   */
  }
  printf("\b\n");
}
int main(int args, char* argv[]){
        int tab[5]={1,2,3,0,0};
        IntArray array = {tab,5,3};
        printIntArray(&array);
        return 0;
}
```

Note : le "plus" est prioritaire sur l'étoile, mais mieux vaut parentheser quand meme

Transtypage

- pas de problème de conversion pour :
  - void* vers void*
  - truc* vers truc*
  - truc* vers void *
  - void* vers truc*
- problème pour truc1* vers truc2*
- il faut une conversion explicite (cast) :

```
truc1* b=...;
truc2* t=(truc2*)b;
```

Exemple d'utilisation

Écrire une fonction qui retourne le codage des entiers sur la machine ? On interprète un int comme un tableau de char, et on test le premier élément.

```
enum{BIG_ENDIAN,LITTLE_ENDIAN,BIG_ENDIAN_SWAP,
     LITTLE_ENDIAN_SWAP};
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;
    }
}
```

Allocation dynamique

- principe : demander une zone mémoire au système (peut donc échouer !)
- le système renvoit l'adresse de la zone, donc un pointeur
- cette zone est prise sur le tas
- et est persistante (réservée) jusqu'à ce qu'elle soit libérée explicitement (contrairement aux variables locales allouées sur la pile)

malloc

```
void* malloc(size_t size);
```

- définie dans stdlib.h
- size = taille en octets de la zone réclamée
- retourne :
  - la valeur spéciale NULL en cas d'échec
  - l'adresse d'une zone au contenu indéfini sinon
- le type de reour est `void*` : pas besoin donc d'écrire le cast explicitement
- mais en pratique on l'écrit, évite les erreurs bêtes !

Règles d'or du malloc

- toujours tester le retour de malloc (puisque la fonction peut échouer !)
- toujours multiplier le nombre d'éléments par la taille
- toujours mettre un cast pour indiquer le type (lisibilité)
- usage prototypique (allocation d'un tableau de 10 entiers) :

```
int* t=(int*)malloc(10*sizeof(int));
if (t==NULL) {
  fprintf(stderr,"Not enough memory!\n");
  exit(1);
}
```

calloc

```
void* calloc(size_t nmemb,size_t size);
```

- `nmemb` : le nombre d'élément
- `size` : la taille d'un élément
- équivalent à `malloc(nmemb*size);` mais initialise en plus chaque élément à 0

```
int* create_array(int size) {
    int* array=(int*)calloc(size,sizeof(int));
    if (array==NULL) {
        fprintf(stderr,"Not enough memory!\n");
        exit(1);
    }
    return array;
}
int main(int argc, char* argv[]){
   int* tab = create_array(10);
   printf("%d\n",tab[5]); /* 0 */
   ...
}
```

realloc

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

- réalloue la zone pointée par `ptr` à la nouvelle taille `size`
- les anciennes données sont conservées (ou tronquées si la taille a diminuée)
- possible copie de données si la nouvelle zone mémoire est allouée ailleur !
- `ptr` doit pointer sur une zone valide (allouée précédemment)

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`
  - qui n'a pas déjà été libérée
- faire attention à ne jamais lire un pointeur sur une zone libérée
- tant que la mémoire n'est pas réutilisée, le programme semble fonctionner
- `segmentation fault` bien plus tard (dépend de la mémoire, des autres processus etc), bug très difficile à localiser !
- attention aux allocations cachées par l'appel à des librairies

Règles à respecter...

...pour éviter les bugs indetectables.

- 1 malloc/calloc = 1 free
- c'est la partie du programme qui est responsable de l'allocation qui fait la libération
- autrement dit, on ne libère pas la mémoire que l'on a pas nous même allouée

```c
#define N 10
void foo(int* t) {
    /* ... */
    free(t); /* !!! */
}
int main(void) {
   int* t;
   t=(int*)malloc(N*sizeof(int)
        );
   if(t==NULL)return 1;
   foo(t);
   return 0;
}
```

```c
#define N 10
void foo(int* t) {
    /* ... */
}
int main(void){
   int* t;
   t=(int*)malloc(N*sizeof(int)
        );
   if(t==NULL)return 1;
   foo(t);
   free(t);
   return 0;
}
```

Tableaux dynamiques à 2 dimensions

C'est un tableau de tableaux

- on déclare un pointeur de pointeur : ex. `int ** t;`
- **t est un entier, *t est un int*, t est un int**
- allouer d'abord le tableau principal (attention au type des éléments !)
- puis parcourir ce tableau et allouer chaque sous-tableaux
- le raisonnement peut s'étendre à n dimensions (pas de limite)
- différent des tableaux déclarés $t[][]$: éléments non contigus !

Exemple

```
int** init_array(int X,int Y) {

  int i,j;
  int** t;

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

  for (i=0;i<X;i++) {
    t[i]=(int*)malloc(Y*sizeof(int));
    if (t[i]==NULL){
      for(j=0;j<i;j++)
        free(t[j]);
      free(t); /* ne pas oublier ! */
      return NULL;
    }
  }

  return t;
}
```

Structures dynamiques

Quand on passe une structure à une fonction, ou qu'une fonction renvoit une structure, problèmes :

- peut être couteux
- visibilité des modifications ?

Donc en général, on passe la structure par adresse, et on renvoit une adresse.

Règles

On fait une fonction d'allocation/initialisation et une fonction de libération pour la structure :

```
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);
}
```

On n'utilise pas malloc/free dans les autres fonctions !

Règles

```c
typedef struct { /* ... */} Other;
typedef struct {
    Other * other; /* et pas Other other; */
    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;
  c->other = new_Other(...); /* et pas c->other=malloc() */
  return c;
}
void free_Complex(Complex* c) {
  free_Other(c->other); /* et pas free(c->other); */
  free(c);
}
```

Listes chaînées

chaque cellule contient une donnée et l'adresse de la donnée 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;
    }
}
```

Listes Chaînées VS Tableaux

- tous les éléments ne sont plus au même endroit de la mémoire
- parcours plus coûteux, mais de même complexité
- insertions/suppressions en temps constant
- une même cellule peut servir dans plusieurs listes (attention aux modifications)
- pour le reste, voir un cours d'algo