Cours de C - IR1
2007-2008

Types structurés et allocation dynamique

Sébastien Paumier

Les structures

- objets regroupant plusieurs données appelées "champs"
- à définir hors d'une fonction
- définition:

```
struct nom {
    type_champ1 nom_champ1;
    type_champ2 nom_champ2;
    ...
};
```

Les structures

- déclaration d'une variable:

```
struct nom_type nom_var;
```

- accès aux champs: **nom_var.nom_champ**

```
struct complex {
    double real;
    double imaginary;
};

int main(int argc,char* argv[]) {
    struct complex c;
    c.real=1.2;
    c.imaginary=6.3;
    printf("%f+%f*i\n",c.real,c.imaginary);
    return 0;
}
```

Choix des champs

- on peut mettre tout ce dont le compilateur connaît la taille
- on ne peut donc pas mettre la structure elle-même:

```
struct list {
   int value;
   struct list next;
};
```



```
$>gcc -Wall -ansi struct.c
struct.c:7: champ « next » a un type incomplet
```

Structures récursives

- mais on peut mettre un pointeur sur la structure, car la taille des pointeurs est connue:

```
struct list {
   int value;
   struct list* next;
};
```

Imbrication de structures

- un champ peut être une structure:

```
struct id {
    char firstname[MAX];
    char lastname[MAX];
};

struct people {
    struct id id;
    int age;
};

int main(int argc,char* argv[]) {
    struct people p;
    strcpy(p.id.firstname,"Master");
    strcpy(p.id.lastname,"Yoda");
    p.age=943;
    /* ... */
    return 0;
}
```

pas d'ambiguïté car le type ne s'appelle pas **id** mais **struct id**

Imbrication de structures

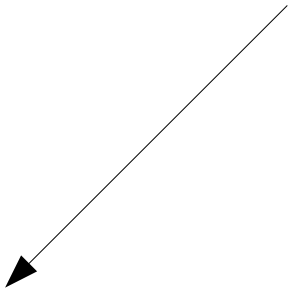
- 2 structures peuvent s'appeler l'une l'autre, à condition d'utiliser des pointeurs:

```
struct state {
    struct transition* t;
    char final;
};

struct transition {
    char letter;
    struct state* dest;
    struct transition* next;
};
```

Contraintes sur les structures

- champs organisés dans l'ordre de leur déclarations:

```
struct foo {
    int a;
    char b;
    float c;
};

int main(int argc,char* argv[]) {
    struct foo f;
    printf("%p < %p < %p\n",
            &(f.a),&(f.b),&(f.c));
    return 0;
}
```



```
$>./a.out
0022FF58 < 0022FF5C < 0022FF60
```

Alignement

- adresse d'une structure = adresse de son premier champ
- pour les autres champs, le compilateur fait de l'alignement pour avoir, selon l'implémentation:
  - des adresses multiples de la taille des données
  - des adresses multiples de la taille des pointeurs
  - ...

Alignement

- taille variable suivant l'ordre des champs:

```
struct to {
    char a;
    int b;
    char c;
    char d;
};
```

→ `sizeof(to)=12`

```
struct ta {
    int a;
    char b;
    char c;
    char d;
};
```

→ `sizeof(ta)=8`

- ne jamais essayer de deviner l'adresse d'un champ ou la taille d'une structure:
  - noms des champs
  - **`sizeof`**

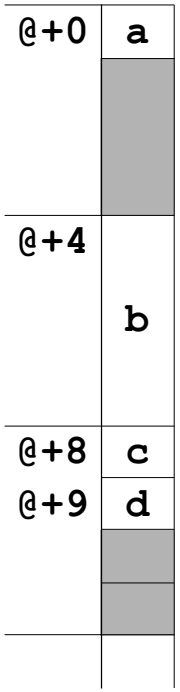
Alignement

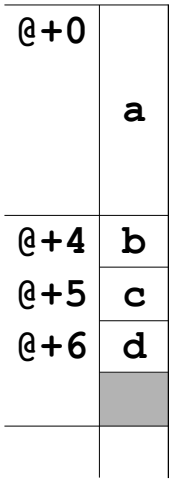
```
struct to {
    char a;
    int b;
    char c;
    char d;
};
```

→ sizeof(to)=12

```
struct ta {
    int a;
    char b;
    char c;
    char d;
};
```

→ sizeof(ta)=8





Initialisation

- **struct to t={val$_0$,val$_1$,...,val$_{n-1}$};**

```
struct to t={'a',145,'y','u'};
```

- seulement lors de la déclaration

```
int main(int argc,char* argv[]) {
    struct to t;
    t={'a',145,'y','u'};
    printf("%c %d %c %c\n",t.a,t.b,t.c,t.d);
    return 0;
}
```

```
$>gcc -Wall -ansi struct.c
struct.c: Dans la fonction « main »:
struct.c:33: erreur d'analyse syntaxique avant le jeton « { »
```

Opérations

- l'affectation fonctionne

```
int main(int argc,char* argv[]) {
    struct to z={'a',145,'y','u'};
    struct to t=z;
    printf("%c %d %c %c\n",t.a,t.b,t.c,t.d);
    return 0;
}
```

- attention: si un champ est un pointeur, on ne copie que l'adresse
- pas d'opérateur de comparaison

  ⇨ à écrire soi-même

Structures en paramètres

- les structures sont passées par valeur!
  - non modifiables
  - peuvent occuper beaucoup de place sur la pile (tableaux)
  - temps de recopie
- toujours les passer par adresse 

Structures en paramètres

```
struct array {
   int t[100000];
   int size;
};

void f(struct array a) {
   int i;
   for (i=0;i<a.size;i++) {
      printf("%d\n",a.t[i]);
   }
}
```

```
struct array {
   int t[100000];
   int size;
};

void f(struct array* a) {
   int i;
   for (i=0;i<(*a).size;i++) {
      printf("%d\n",(*a).t[i]);
   }
}
```

attention au parenthésage: `(*a).t[i]` ≠ `*a.t[i]`

Structures en paramètres

- passage d'adresse =
  - gain de temps
  - gain d'espace sur la pile
  - on peut modifier la structure
- notation simplifiée: **`(*a).size`** = **`a->size`**

```
void f(struct array* a) {
   int i;
   for (i=0;i<(*a).size;i++) {
      printf("%d\n",(*a).t[i]);
   }
}
```

```
void f(struct array* a) {
   int i;
   for (i=0;i<a->size;i++) {
      printf("%d\n",a->t[i]);
   }
}
```

Retourner une structure

- on peut, mais c'est une opération de copie sur la pile
- mêmes problèmes que pour le passage par valeur
- à éviter absolument 
- il faudra utiliser de l'allocation dynamique

Les unions

```
union toto {
    type1 nom1;
    type2 nom2;
    ...
    typeN nomN;
};
```

→ zone mémoire que l'on peut voir soit comme un `type1`, soit comme un `type2`, etc.

- s'utilisent comme les structures

```
union toto {
    char a;
    float b;
};

void foo(union toto* t) {
    t->a='z';
}
```

Les unions

- taille = taille du plus grand champ
- au programmeur de savoir quel champ doit être utilisé

```
union toto {
    char a;
    char s[16];
};

int main(int argc,char* argv[]) {
    union toto t;
    strcpy(t.s,"coucou");
    t.a='$';
    printf("%s\n",t.s);
    return 0;
}
```

→

```
$>./a.out
$oucou
```

Les unions

- utiles quand on doit manipuler des informations exclusives les unes des autres

```
union student {
    char login[16]
    int id;
};
```

- peuvent être utilisées anonymement dans une structure

```
struct student {
    char name[256];
    union {
        char login[16];
        int id;
    };
};
```

Unions complexes

- on peut mettre des structures (anonymes) dans les unions

```
union color {
    /* RGB representation */
    struct {
        unsigned char red,blue,green;
    };
    /* 2 colors: 0=black 1=white */
    char BandW;
};
```

on peut utiliser soit **red**, **blue** et **green**, soit **BandW**

Quel(s) champ(s) utiliser ?

- encapsulation dans une structure avec un champ d'information

```
struct color {
   /* 0=RGB 1=black & white */
   char type;
   union {
      /* RGB representation */
      struct {
         unsigned char red,blue,green;
      };
      /* 2 colors: 0=black 1=white */
      char BandW;
   };
};
```

Les énumérations

- **enum nom {id_0,id_1,...,id_{n-1}};**
- si on a une variable de type **enum nom**, elle pourra prendre les valeurs **id_i**

```
enum gender {male,female};

void init(enum gender *g,char c) {
   *g=(c=='m')?male:female;
}
```

Les énumérations

- valeurs de type **`int`**
- par défaut, commencent à 0 et vont de 1 en 1
- on peut les modifier

```
enum color {
    blue=45,
    green, /* 46 */
    red,   /* 47 */
    yellow=87,
    black  /* 88 */
};
```

Les énumérations

- on peut avoir plusieurs fois la même valeur

```
enum color {
   blue=45, BLUE=blue, Blue=blue,
   green/* =46 */, GREEN=green, Green=green
};

int main(int argc,char* argv[]) {
   printf("%d %d %d %d %d %d\n",
          blue,BLUE,Blue,green,GREEN,Green);
   return 0;
}
```

```
$>./a.out
45 45 45 46 46 46
```

Contrôles des valeurs

- contrairement aux espoirs du programmeur, pas de contrôle!

```
enum gender {male='m',female='f'};

enum color {blue,red,green};

int main(int argc,char* argv[]) {
   enum gender g='z';
   enum color c=g;
   /* ... */
   return 0;
}
```

Contrôles des valeurs

- seul contrôle: avec **-Wall**, gcc indique s'il manque des cas dans un **switch** (quand il n'y a pas de **default**)

```
enum color {blue,red,green,yellow};

void foo(enum color c) {
    switch (c) {
        case blue: /* ... */ break;
        case red: /* ... */ break;
    }
}
```

```
$>gcc -Wall -ansi enum.c
enum.c: Dans la fonction « foo »:
enum.c:24: AVERTISSEMENT: valeur d'énumération « green » n'est pas traitée dans le switch
enum.c:24: AVERTISSEMENT: valeur d'énumération « yellow » n'est pas traitée dans le switch
```

Déclaration de constantes

- si on veut juste déclarer des constantes, on peut utiliser une énumération anonyme

```
enum {Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday};

char* names[]={"Monday","Tuesday","Wednesday","Thursday","Friday",
               "Saturday","Sunday"};

void print_day(int day) {
   printf("%s\n",names[day]);
}

int main(int argc,char* argv[]) {
   print_day(Saturday);
   return 0;
}
```

Combinaison union/enum

- quand on utilise une union, c'est plus propre de décrire les alternatives avec une énumération:

```
enum cell_type {EMPTY,BONUS,MALUS,PLAYER,MONSTER};

struct cell {
    enum cell_type type;
    union {
        Bonus bonus;
        Malus malus;
        Player player;
        Monster monster;
    };
};
```

typedef

- **`typedef type nom;`**
- permet de donner un nom à un type, simple ou composé
- pratique pour éviter de recopier les mots-clés **`struct`**, **`union`** et **`enum`**

```
typedef signed char sbyte;
typedef unsigned char ubyte;

typedef struct cell Cell;
typedef enum color Color;
```

typedef

- pas de nouveaux types
- seulement des synonymes interchangeables

```
struct array {
    int t[N];
    int size;
};

typedef struct array Array;

int main(int argc,char* argv[]) {
    Array a;
    struct array b=a;
    /* ... */
    return 0;
}
```

typedef

- pour les types structurés, deux modes de définition:

```
enum cell_type {EMPTY,BONUS,MALUS,
                PLAYER,MONSTER};

typedef enum cell_type CellType;

struct cell {
   CellType type;
   union {
      Bonus bonus;
      Malus malus;
      Player player;
      Monster monster;
   };
};

typedef struct cell Cell;
```

```
typedef enum {EMPTY,BONUS,MALUS,
        PLAYER,MONSTER} CellType;

typedef struct {
   CellType type;
   union {
      Bonus bonus;
      Malus malus;
      Player player;
      Monster monster;
   };
} Cell;
```

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!

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:

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

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

```
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

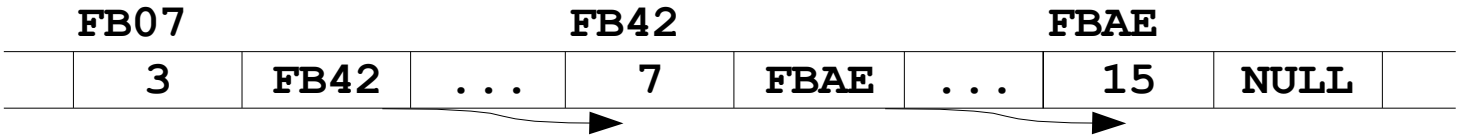
- chaque cellule pointe sur la suivante
- grâce à l'allocation dynamique, on peut avoir des listes arbitrairement longues

```
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