CM 8 - TYPES STRUCTURÉS ET TYPEDEF

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

TYPES STRUCTURÉS

LES STRUCTURES

▷ Les structures sont des objets regroupant plusieurs données appelées ”champs”

▷ Elles sont à définir hors de toute fonction

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

▷ ⚠Le `;` est nécessaire

LES STRUCTURES

▷ La déclaration d'une variable de type structure s'effectue ainsi

```
struct nom_type nom_var;
```

▷ L'accès à un champ d'une structure s'effectue comme suit:

```
nom_var.nom_champ
```

```
struct complex {
    double real;
    double imaginary;
};
int main(void) {
    struct complex c;
    c.real=1.2;
    c.imaginary=6.3;
    printf("%.1f+%.1f*i\n",c.real,c.imaginary);
    return EXIT_SUCCESS;
}

$>./a.out
1.2+6.3*i
```

RESTRICTIONS SUR LES CHAMPS

▷ On peut mettre comme champ d'une structure 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 -std=c99 struct.c
struct.c:7: champ "next" a un type incomplet
```

STRUCTURES RÉCURSIVES

▷ La taille d'un pointeur est commune quelque soit le type cible (mais dépendante de l'architecture du système)

▷ On peut donc créer une structure récursive à l'aide d'un pointeur sur la structure

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

IMBRICATION DE STRUCTURES

▷ Un champ peut être une structure

▷ `char* strcpy(char* dest, const char* src)` de la librairie `<string.h>` copie la chaîne `src` vers `dest` (⚠ aux débordements)

```
struct id {
    char firstname[MAX];
    char lastname[MAX];
};
struct people {
    struct id identifier;
    int age;
};
int main(void) {
    struct people p;
    strcpy(p.identifier.firstname,"Master");
    strcpy(p.identifier.lastname,"Yoda");
    p.age=943;
    return EXIT_SUCCESS;
}
```

IMBRICATION DE STRUCTURES

▷ Deux structures peuvent se référencer l'une l'autre, à condition d'utiliser des pointeurs

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

struct transition {
    char letter;
    struct state * p_dest;
    struct transition * p_next;
};
```

CONTRAINTES SUR LES STRUCTURES

▷ Les champs sont organisés dans l'ordre de leur déclaration

```
struct foo {
    int a;
    char b;
    float c;
};
int main(void) {
    struct foo f;
    printf("%p < %p < %p\n", &(f.a),&(f.b),&(f.c));
    return EXIT_SUCCESS;
}
```

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

ALIGNEMENT

- L'adresse du premier champ d'une structure est la même que celle de la structure elle même
- 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

▷ ⚠La taille d'une structure est variable suivant l'ordre des champs

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

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

`sizeof(to)=12`

`sizeof(ta)=8`

▷ Il ne faut jamais essayer de deviner l'adresse d'un champ ou la taille d'une structure mais s'en remettre

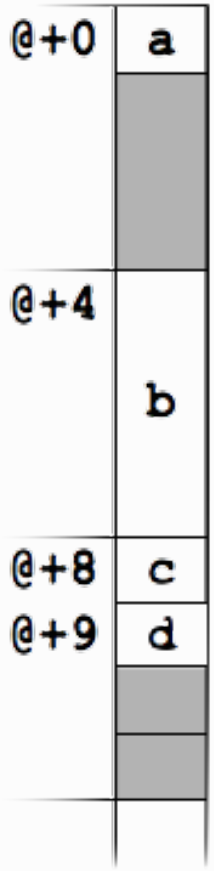
  ▷ Aux noms des champs

  ▷ Et à l'opérateur `sizeof`

ALIGNEMENT

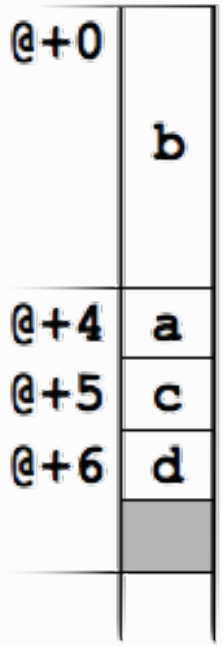
```
struct to {
    char a;
    int b;
    char c;
    char d;
};
```

sizeof(to)=12




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

sizeof(ta)=8



Initialisation d'une structure

▷ L'initialisation d'une variable de type structure peut s'effectuer ainsi `struct to t={val0,val1,...,valn};` mais uniquement lors de sa déclaration

```
int main(void) {
    struct to t={'a',145,'y','u'};
    printf("%c %d %c %c\n",t.a,t.b,t.c,t.d);
    return EXIT_SUCCESS;
}
```

```
$>./a.out
a 145 y u
```

INITIALISATION D'UNE STRUCTURE

▷ L'initialisation d'une variable de type structure peut s'effectuer ainsi `struct to t={val0,val1,...,valn};` mais uniquement lors de sa déclaration

```
int main(void) {
    struct to t;
    t={'a',145,'y','u'};
    printf("%c %d %c %c\n",t.a,t.b,t.c,t.d);
    return EXIT_SUCCESS;
}
```

```
$>gcc -Wall -std=c99 struct.c
struct.c: Dans la fonction "main":
struct.c:33: erreur d'analyse syntaxique avant le
jeton "{"
```

Opérations sur une structure

▷ L'affectation d'une structure fonctionne et produit la recopie des valeurs des champs

```
int main(void) {
    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 EXIT_SUCCESS;
}
```

▷ ⚠Si un champ est un pointeur, on ne copie que l'adresse

▷ Il n'y a, en revanche, pas d'opérateur de comparaison

  ▷ Il faut écrire une fonction de comparaison soi-même

STRUCTURES EN PARAMÈTRES

- ▷ Ce sont des copies des structures qui sont passées en paramètre
  - ▷ Elles peuvent occuper beaucoup de place en mémoire
  - ▷ ⚠La recopie est consommatrice de temps
- ▷ → Il faut transmettre leur adresse sous forme de pointeur

```
struct array {
    int t[100000];
    int size;
};
void f(struct array* p_a) {
    int i;
    for (i=0;i<(*p_a).size;i=i+1) {
        printf("%d\n",(*p_a).t[i]);
    }
}
```

STRUCTURES EN PARAMÈTRES VIA POINTEUR

- L'utilisation de pointeur vers les structures permet
  - Un gain de temps
  - Un gain d'espace mémoire
  - La modification de la structure pointée
- `(*p_a).size` et `p_a->size` sont des notations équivalentes

```
void f(struct array* p_a) {
    int i;
    for (i=0;i<(*p_a).size;i=i+1) {
        printf("%d\n",(*p_a).t[i]);
    }
}
```

```
void f(struct array* p_a) {
    int i;
    for (i=0;i<p_a->size;i=i+1) {
        printf("%d\n",p_a->t[i]);
    }
}
```

RETOURNER UNE STRUCTURE

▷ Retourner une structure est possible mais non recommandé

▷ Cela pose les mêmes problèmes que le passage en argument d'une structure

▷ Il ne faut l'utiliser que si on ne peut pas faire autrement

LES UNIONS

▷ Les unions se déclarent et s'utilisent de manière similaire aux structures

▷ Mais elles correspondent à une zone mémoire que l'on peut voir comme n'importe lequel de ses champs (un unique espace mémoire est partagé par les champs)

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

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

void foo(union toto* p_t) {
    (*p_t).a='z';
}
```

LES UNIONS

▷ La taille d'une **union** correspond à la taille de son plus grand champ

▷ ⚠C'est au programmeur de savoir quel champ utiliser

```
union toto {
    char a;
    char s[16];
};
int main(void) {
    union toto t;
    strcpy(t.s,"coucou");
    t.a='$';
    printf("%s\n",t.s);
    return EXIT_SUCCESS;
}
```

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

UTILITÉ DES UNIONS

▷ Les unions sont utiles quand on doit manipuler des informations exclusives les unes des autres

▷ Elles peuvent être utilisées de manière anonyme dans une structure

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

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

UNIONS COMPLEXES

▷ Inversement, on peut utiliser des structures de manière anonyme dans des unions

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

▷ On peut alors utiliser soit les champs `red`, `blue` et `green`, soit le champ `BandW`

UNIONS COMPLEXES

▷ Pour préciser quel(s) champ(s) utiliser, on peut l'encapsuler 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

▷ Les énumérations permettent de regrouper des valeurs constantes entières nommées via la déclaration suivante

```
enum nom {id0, id1, ..., idn};
```

▷ Une variable de type **enum nom** pourra prendre une des valeurs constantes correspondantes

```
enum gender {male,female};
void init(enum gender * p_g, char c) {
    if(c=='m'){
        *p_g=male;
    }else{
        *p_g=female;
    }
}
```

LES ÉNUMÉRATIONS

▷ Ce sont des valeurs entières commençant par défaut à 0 et allant de 1 en 1

▷ Mais dont on peut modifier/spécifier la valeur

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

LES ÉNUMÉRATIONS

▷ On peut avoir plusieurs fois la même valeur dans une énumération

```
enum color {
    blue=45, BLUE=blue, Blue=blue,
    green /* 46 */, GREEN=green, Green=green
};
int main(void) {
    printf("%d %d %d %d %d %d\n",
        blue, BLUE, Blue, green, GREEN, Green);
    return EXIT_SUCCESS;
}
```

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

CONTRÔLE DES VALEURS

▷ ⚠Contrairement aux espoirs du programmeur, il n'y a pas de contrôle des valeurs !

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

enum color {blue,red,green};

int main(void) {
    enum gender g='z';
    enum color c=g;
    return EXIT_SUCCESS;
}
```

DÉCLARATION DE CONSTANTES

▷ Si l'on souhaite juste déclarer un ensemble de 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(void) {
    print_day(Saturday);
    return EXIT_SUCCESS;
}
```

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

▷ L'instruction `typedef` permet de donner un nom à un type simple ou composé via la déclaration suivante

```
typedef type nom;
```

▷ Cela peut permettre d'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

▷ Cela n'induit pas la création de nouveaux types mais offre uniquement des synonymes interchangeables

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

typedef struct array Array;

int main(void) {
    Array a;
    struct array b=a;
    return EXIT_SUCCESS;
}
```

TYPEDEF

▷ On peut définir de deux façons les types structurés

```
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;
```

DOGGY BAG

TO TAKE AWAY ...

▷ Les structures sont des objets regroupant plusieurs données appelées ”champs”

▷ On peut mettre comme champ de structure tout ce dont le compilateur connaît la taille

▷ ⚠Il ne faut jamais essayer de deviner l’adresse d’un champ ou la taille d’une structure

▷ Une union correspond à une zone mémoire que l’on peut voir comme n’importe lequel de ses champs

▷ Les énumérations permettent de regrouper des valeurs constantes

Questions?