



Cours de C

Types structurés

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:

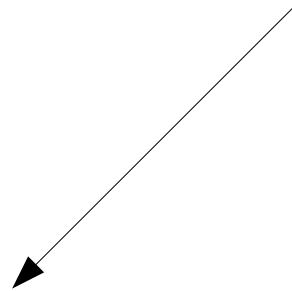
```
struct state {  
    struct transition* t;  
    char final;  
};  
  
struct transition {  
    char letter;  
    struct state* dest;  
    struct transition* next;  
};
```



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

@+0	a
@+4	b
@+8	c
@+9	d

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

→ sizeof(ta)=8

@+0	a
@+4	b
@+5	c
@+6	d



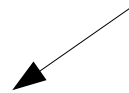
Initialisation

- `struct t t={val0,val1,...,valn-1};`

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

- seulement lors de la déclaration

```
int main(int argc, char* argv[]) {  
    struct t 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: $(*a).t[i] \neq *a.t[i]$



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 n'utiliser que très soigneusement ⚡
- le plus souvent, 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
\$coucou



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 {id0, id1, ..., idn-1};`
- si on a une variable de type `enum nom`, elle pourra prendre les valeurs `idi`

```
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 listes

- application: coder une liste chaînée de noms de couleurs
- utilisons une structure pour représenter une cellule de la liste:

```
#define COLOR_NAME_MAX 128

typedef struct {
    char name[COLOR_NAME_MAX];
    int next;
} ColorList;
```



Les listes

- la mémoire est simulée par un tableau de telles structures:

```
/* The cell array that represents memory */  
#define SIZE 1000  
extern ColorList memory[SIZE];
```

- une cellule est caractérisée par son indice dans ce tableau
- il faut une convention pour désigner l'absence de cellule en fin de liste:

```
/* The equivalent of NULL */  
#define NOTHING -1
```



Les listes

- pour gérer la mémoire, il faut savoir quelles cellules sont disponibles
- astuce: utilisons une valeur spéciale pour le champ **next**

```
/* A convention to indicate when a cell is not used */  
#define EMPTY_CELL -2
```



Les listes

- il faut savoir allouer et désallouer:

```
/**
 * Returns the index of the first free cell, or 'NOTHING' if there
 * is none.
 */
int get_free_cell() {
    int i;
    for (i=0;i<SIZE;i++) {
        if (memory[i].next==EMPTY_CELL) return i;
    }
    return NOTHING;
}

/**
 * Marks the given memory cell as being free.
 */
void free_cell(int index) {
    memory[index].next=EMPTY_CELL;
}
```



Les listes

- on peut maintenant ajouter une couleur en tête de liste:

```
/**
 * Adds a cell representing the given color at the beginning of the given
 * list. Returns 1 in case of success, 0 if there is no more memory.
 */
int add_color(int *list, char* color) {
    int index=get_free_cell();
    if (index==NOTHING) {
        return 0;
    }
    strcpy(memory[index].name,color);
    memory[index].next>(*list);
    (*list)=index;
    return 1;
}
```



Les listes

- on veut pouvoir afficher la liste:

```
/**
 * Prints the given list.
 */
void print_list(int list) {
    printf("list=");
    while (list!=NOTHING) {
        printf("%s ",memory[list].name);
        list=memory[list].next;
    }
    printf("\n");
}
```



Les listes

- enfin, il faut pouvoir libérer une liste entière:

```
/**
 * Frees the whole given list.
 */
void free_list(int list) {
    int tmp;
    while (list!=NOTHING) {
        tmp=memory[list].next;
        free_cell(list);
        list=tmp;
    }
}
```



Les listes

- on peut maintenant tester notre bibliothèque:

```
#include <stdio.h>
#include "color_list.h"

int main(int argc, char* argv[]) {
    init_memory();
    int list=NOTHING;
    add_color(&list, "blue");
    add_color(&list, "black");
    add_color(&list, "red");
    print_list(list);
    free_list(list);
    return 0;
}
```

→ \$> ./a.out
list=red black blue



Les listes

- bug si on essaie d'afficher la liste après l'avoir libérée:

```
#include <stdio.h>
#include "color_list.h"

int main(int argc, char* argv[]) {
    init_memory();
    int list=NOTHING;
    add_color(&list, "blue");
    add_color(&list, "black");
    add_color(&list, "red");
    print_list(list);
    free_list(list);
    print_list(list);
    return 0;
}
```

```
$> ./a.out
list=red black blue
list=blue blue blue
blue blue blue blue
blue blue blue blue
blue blue blue
blue...
```



Les listes

- normal: la condition d'arrêt de `print_list` est `list==NOTHING`, or on a mis la valeur `EMPTY_CELL`
- solution: mettre `list` à `NOTHING` après l'appel à `free_list`

```
int main(int argc, char* argv[]) {  
    ...  
    free_list(list);  
    list=NOTHING;  
    print_list(list);  
    return 0;  
}
```



Les listes

- la fausse bonne idée: faire en sorte que `free_list` modifie son paramètre:

```
/**
 * Frees the whole given list and sets '*list' to 'NOTHING'.
 */
void free_list2(int *list) {
    int tmp;
    while (*list!=NOTHING) {
        tmp=memory[*list].next;
        free_cell(*list);
        *list=tmp;
    }
}
```



Les listes

- pourquoi n'est-ce pas une bonne idée ?
- parce qu'une même cellule peut être référencée plusieurs fois, et qu'on ne va mettre à jour qu'une seule référence:

```
int main(int argc, char* argv[]) {  
    ...  
    print_list(list);  
    int list2=list;  
    free_list2(&list);  
    print_list(list);  
    print_list(list2);  
    return 0;  
}
```

```
$> ./a.out  
list=red black blue  
list=  
→ list=blue blue blue  
blue blue blue blue  
blue blue blue blue  
blue...
```