Programmation C

Types structurés

ESIFE Info 1^{ère} année

Université de Marne-la-Vallée

Les structures

- ▶ objets regroupant plusieurs données appelées "champs"
- ▶ à définir hors d'une fonction (dans un `.h` si la structure a vocation à être diffusée ailleurs, dans un `.c` s'il on souhaite la confiner)
- ▶ définition:

```
1 struct nom{  
2     type_champ1  nom_champ1 ;  
3     type_champ2  nom_champ2 ;  
4     ...  
5 };
```

Les structures

- ▶ déclaration d'une variable :

```
struct nom_type nom_var;
```

- ▶ accès aux champs : `nom_var.nom_champ`

```
1 struct complex{
2     double real;
3     double imaginary;
4 };
5
6 int main(int argc, char* argv[]){
7     struct complex c;
8     c.real = 1.2;
9     c.imaginary = 6.3;
10    printf("%f+%f*i\n", c.real, c.imaginary);
11    return 0;
12 }
```

Choix des champs

- ▶ on peut mettre tout ce dont le compilateur connaît la taille
- ▶ on ne peut donc pas mettre la structure elle-même
- ▶ mais on peut mettre un pointeur sur la structure , car la taille des pointeurs est connue :

```
1 struct list{  
2     int value;  
3     struct list* next;  
4 };
```

Imbrication de structures

- ▶ un champ peut être une structure :

```
1  struct id{
2      char firstname [MAX];
3      char lastname [MAX];
4  };
5
6  struct people{
7      struct id id;
8      int age;
9  };
10
11 int main(int argc char* argv[]){
12     struct people p;
13     strcpy(p.id.firstname, "Master");
14     strcpy(p.id.lastname, "Yoda");
15     p.age = 943;
16     /* ... */
17     return 0;
18 }
```

Imbrication de structures

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

```
1 struct etat{
2     char* nom;
3     int nb_transition;
4     struct transition* tab;
5 };
6
7 struct transition{
8     char letter;
9     struct etat* source;
10    struct etat* destination;
11 };
```

Contraintes sur les structures

- ▶ champs organisés dans l'ordre de leurs déclarations:

```
1  struct foo{
2      char c;
3      int i;
4      float f;
5      double d;
6  };
7
8  int main(int argc, char* argv[]){
9      struct foo f;
10     printf("%p<_%p<_%p<_%p\n",
11            &(f.c), &(f.i), &(f.f), &(f.d));
12     return 0;
13 }
```

donne sur une exécution (machine 64 bits)

```
$> ./test
```

```
0x7fffc6c41a40 < 0x7fffc6c41a44 < 0x7fffc6c41a48 < 0x7fffc6c41a50
```

Alignement mémoire

- ▶ 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
 - ... (d'autres choses choisies raisonnablement par les développeurs de gcc)

Alignement mémoire

- ▶ taille variable suivant l'ordre des champs :

```
1  struct s1{
2    char c1;
3    int i;
4    char c2;
5  };
6
7  struct s2{
8    char c1;
9    char c2;
10   int i;
11  };
```

`sizeof(struct s1) = 12` et `sizeof(struct s2) = 8`

- ▶ Ne jouez pas aux petits malins! On ne devine pas la taille des structures ou l'adresse des différents champs.
 - on utilise le nom des champs
 - on utilise `sizeof`

Alignement mémoire

```
1 struct s1{  
2   char c1;  
3   int i;  
4   char c2;  
5 };
```

@+0	c1
@+4	i
@+8	c2

```
1 struct s2{  
2   char c1;  
3   char c2;  
4   int i;  
5 };
```

@+0	c1
@+1	c2
@+4	i

Initialisation

- ▶ `struct foo var={val1, val2, ... , valn}`
- ▶ seulement lors de la déclaration de la variable `var`

```
1  struct foo{
2    int i;
3    char c;
4  };
5
6  int main(int argc, char* argv[]){
7    struct foo t;
8    t={12, 'a'};
9    return 0;
10 }
```

donne

test3.c:8:5: erreur: expected expression before '{' token

Opérations

- ▶ l'affectation fonctionne

```
1 int main(int argc, char* argv[]){  
2     struct foo t={12, 'a'};  
3     struct foo z=t; /* recopiage ok! */  
4     return 0;  
5 }
```

- ▶ attention : si un champ est un pointeur, on ne copie que l'adresse (et donc si on libère le premier avec `free`, bonjour les dégâts)
- ▶ pas d'opération 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 copie
- ▶ conclusion : toujours passer les structures par adresse (qu'on les modifie ou pas)

Structures en paramètres

```
1  struct array{
2      int t[100000];
3      int size;
4  };
5
6  void f(struct array* a){
7      int i;
8      for (i=0 ; i<(*a).size ; i++)
9          printf("%d\n", (*a).t[i]);
10 }
```

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 :

`(*foo).bar` $\Leftrightarrow$ `foo->bar`

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 champs de bits

- ▶ on peut économiser de la mémoire dans une structure en spécifiant le nombre de bits que l'on souhaite utiliser pour stocker certains entiers
- ▶ Pas d'overkill ! Les machines actuelles ont beaucoup de mémoire R.A.M. !
- ▶ toujours forcer la signature du type entier associé (signed int ou unsigned int mais pas int)
- ▶ Les champs N'ONT PLUS D'ADRESSES !!!!! Tous les octets de la R.A.M. ont une adresse, mais pas les bits! On utilise alors les champs mais pas d'opérateur d'adressage, attention aux arrondis.

Les champs de bits

```
1  struct fiche1 {
2      char nom[STRMAX];
3      char prenom[STRMAX];
4      unsigned int sexe;
5      unsigned int marie;
6      unsigned int nbEnfants;
7  };
8
9  struct fiche2 {
10     char nom[STRMAX];
11     char prenom[STRMAX];
12     unsigned int sexe:1; /* champ de 1 bit */
13     unsigned int marie:1; /* champ de 1 bit */
14     unsigned int nbEnfants:5; /* champ de 5 bit */
15 };
16
17 int main(int argc, char* argv[]){
18     printf("fiche1 : %lu\nfiche2 : %lu\n",
19           sizeof(struct fiche1), sizeof(struct fiche2));
20     return 0;
21 }
```

```
nborie@perceval:~> ./test
fiche1 : 36
fiche2 : 24
```

Les unions

```
1 union foo{
2     type1 nom1;
3     type2 nom2;
4     ...
5     typeN nomN;
6 };
```

- ▶ zone mémoire que l'on peut voir soit comme un **type1**, soit comme un **type2**, soit ... etc.
- ▶ s'utilisent comme des structures

```
1 union foo{
2     char a;
3     float b;
4 };
5
6 void foo(union foo* t){
7     t->a = 'z';
8 }
```

Les unions

- ▶ taille = taille du plus grand champ
- ▶ c'est au programmeur de savoir quel champ doit être utilisé

```
1 union foo{
2     char a;
3     char s[16];
4 };
5
6 int main(int argc, char* argv[]){
7     union foo t;
8     strcpy(t.s, "coucou");
9     t.a = '$';
10    printf("%s\n", t.s);
11    return 0;
12 }
```

donne

```
nborie@perceval:~> ./test
$coucou
```

Les unions

- ▶ utiles quand on veut manipuler des informations exclusives les unes des autres

```
1 union etudiant{  
2     char login[16];  
3     int id;  
4 };
```

- ▶ peuvent être utilisées anonymement dans les structures

```
1 struct etudiant{  
2     char name[256];  
3     union{  
4         char login[16];  
5         int id;  
6     };  
7 };
```

Unions complexes

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

```
1 union color{
2     struct{
3         unsigned char red , blue , green ;
4     };
5     char name[6];
6 };
```

on peut alors utiliser soit `red`, `blue` et `green`, soit le champ `name`

Quel(s) champ(s) utiliser ?

- ▶ encapsulation dans une structure avec un champ d'information

```
1 union color{
2     char type; /* 0 = RGB, 1 = by name */
3     struct{
4         unsigned char red , blue , green ;
5     };
6     char name[6];
7 };
```

Les énumérations

- ▶ `enum nom {id0, id2, ... , idn-1}`
- ▶ si on a une variable de type `enum nom`, elle pourra prendre les valeurs `idi`

```
1 enum gender {male, female};  
2  
3 void init(enum gender *g, char c){  
4     *g=(c=='m')?male:female;  
5 }
```

Les énumérations

- ▶ valeur de type `int`
- ▶ par défaut, commencent à 0 et vont de 1 en 1
- ▶ on peut les modifier (avec une bonne raison !)

```
1 enum color{
2     blue=45,
3     green, /* 46 */
4     red, /* 47 */
5     yellow=87,
6     black /* 88 */
7 };
```

Les énumérations

- ▶ on peut avoir plusieurs fois la même valeur

```
1 enum color{
2     blue=45, BLUE=blue, Blue=blue, /* Tout a 45 */
3     green, GREEN=green, Green=green /* Tout a 46 */
4 };
5
6 int main(int argc, char* argv[]){
7     printf("%d %d %d %d %d %d\n",
8           blue, BLUE, Blue, green, GREEN, Green);
9     return 0;
10 }
```

donne

```
nborie@perceval:~> ./test
45 45 45 46 46 46
```

Contrôles des valeurs

- ▶ contrairement aux espoirs du programmeur, pas de contrôle sur les valeurs !

```
1 enum gender {male='m', female='f'};
2
3 enum color {red, green, blue};
4
5 int main(int argc, char* argv[]){
6     enum gender g;
7     enum color c;
8     g = 3;
9     c = 132;
10    printf("m|f : %d\n", (g==male || g==female));
11    printf("r|g|b : %d\n", (c==red || c==green || c==blue));
12    return 0;
13 }
```

donne

```
nborie@perceval:~> ./test
```

```
m|f : 0
```

```
r|g|b : 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`)

```
1 enum color {blue, red, green, yellow};
2
3 void foo(enum color c){
4     switch(c) {
5         case blue: /* ... */ break;
6         case red: /* ... */ break;
7     }
8 }
```

donne

```
test8.c: In function 'foo':
```

```
test8.c:4:3: attention : enumeration value 'green' not handled
in switch [-Wswitch]
```

```
test8.c:4:3: attention : enumeration value 'yellow' not handled
in switch [-Wswitch]
```

Déclaration de constantes

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

```
1 enum {Monday, Tuesday, Wednesday, Thursday, Friday,
2       Saturday, Sunday};
3
4 char* names[] = {"Monday", "Tuesday", "Wednesday",
5                 "Thursday", "Friday", "Saturday",
6                 "Sunday"};
7
8 void print_day(int day){
9     printf("%s\n", names[day]);
10 }
11
12 int main(int argc, char* argv[]){
13     print_day(Saturday);
14     return 0;
15 }
```

Combinaison union/enum

- ▶ Une solution propre consiste à utiliser une énumération pour les alternatives d'une union

```
1 enum cell_type {EMPTY, BONUS, MALUS, PLAYER, MONSTER};
2
3 struct cell{
4     enum cell_type type;
5     union{
6         Bonus bonus;
7         Malus malus;
8         Player player;
9         Monster monster;
10    };
11 };
```

Typedef

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

```
1 typedef signed char Sbyte;  
2 typedef unsigned char Ubyte;  
3  
4 typedef struct cell Cell;  
5 typedef enum color Color;
```

Typedef

- ▶ pas de nouveaux types
- ▶ seulement des synonymes interchangeables

```
1 struct array{
2     int t[N];
3     int size;
4 };
5
6 typedef struct array Array;
7
8 int main(int argc, char* argv[]){
9     Array a;
10    struct array b=a;
11    /* ... */
12    return 0;
13 }
```

Typedef

- ▶ pour les types structurés, deux modes de définition :

```
1 enum cell_type {EMPTY, BONUS,  
2                 MALUS, PLAYER,  
3                 MONSTER};  
4  
5 typedef enum cell_type CellType;  
6  
7 struct cell{  
8     enum cell_type type;  
9     union{  
10        Bonus bonus;  
11        Malus malus;  
12        Player player;  
13        Monster monster;  
14    };  
15 };  
16  
17 typedef struct cell Cell;
```

```
1 typedef enum {EMPTY, BONUS,  
2             MALUS, PLAYER,  
3             MONSTER} CellType;  
4  
5 typedef struct {  
6     CellType type;  
7     union{  
8         Bonus bonus;  
9         Malus malus;  
10        Player player;  
11        Monster monster;  
12    };  
13 } Cell;
```

définition et déclaration commutent dans le code (comme pour les fonctions)

Les listes (non chaînées)

- ▶ exemple : coder une liste chaînée de couleurs
- ▶ utilisons une structure pour définir une cellule de la liste :

```
1 #define COLOR_NAME_MAX 128
2
3 typedef struct {
4     char name[COLOR_NAME_MAX];
5     int next;
6 } ColorList;
```

Les listes

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

```
1  /* memory represente un tableau de cellule */
2  #define SIZE 1000
3  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 :

```
1  /* equivalent a NULL */
2  #define NOTHING -1
```

Les listes

- ▶ pour gérer la mémoire, il faut savoir quelles cellules sont disponibles
- ▶ astuce : utilisons une valeur spéciale du champs `next`

```
1  /* convention pour indiquer les cellules vides */  
2  #define EMPTY_CELL -2
```

Les listes

- ▶ Il faut savoir allouer et désallouer (ce n'est pas du malloc mais on a quand même besoin de savoir l'indiquer) :

```
1  /* Retourne l'indice de la premiere cellule libre dans  
2  la memoire ou -1 s'il n'en existe plus de libre. */  
3  int get_free_cell() {  
4      int i;  
5      for (i=0 ; i<SIZE ; i++){  
6          if (memory[i].next==EMPTY_CELL) return i ;  
7      }  
8      return NOTHING;  
9  }  
10  
11 /* Marque la cellule indexe par 'index' comme etant libre */  
12 void free_cell(int index){  
13     memory[index].next=EMPTY_CELL;  
14 }
```

Les listes

- ▶ On peut maintenant ajouter une couleur en tête de liste :

```
1  /* Rajoute une cellule contenant 'color' a la liste donnee
2     'list '. Retourne 1 en cas de succes et 0 s'il n'y a plus
3     de place disponible. */
4  int add_color(int *list , char* color){
5     int index=get_free_cell();
6     if (index==NOTHING){
7         return 0;
8     }
9     strcpy(memory[index].name, color);
10    memory[index].next>(*list);
11    (*list)=index;
12    return 1;
13 }
```

Les listes

- ▶ on veut pouvoir afficher une liste

```
1  /* Affiche la liste 'list' donnee en argument */
2  void print_list(int list){
3      printf("list=");
4      while (list!=NOTHING){
5          printf("%s_", memory[list].name);
6          list=memory[list].next;
7      }
8      printf("\n");
9  }
```

Les listes

- ▶ enfin, on veut pouvoir libérer une liste entière :

```
1  /* libere complètement la liste donnée en argument */
2  void free_list(int list){
3      int tmp;
4      while (list!=NOTHING){
5          tmp=memory[ list ].next;
6          free_cell(list);
7          list=tmp;
8      }
9  }
```

Les listes

- ▶ on peut maintenant tester notre bibliothèque :

```
1 #include <stdio.h>
2 #include "color_list.h"
3
4 int main(int argc, char* argv[]){
5     init_memory(); /* marque toutes les cellules libres */
6     int list=NOTHING; /* liste vide */
7     add_color(&list, "blue");
8     add_color(&list, "black");
9     add_color(&list, "red");
10    print_list(list);
11    free_list(list);
12    return 0;
13 }
```

donne

```
nborie@perceval:~> ./test
list=red black blue
```

Les listes

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

```
1 #include <stdio.h>
2 #include "color_list.h"
3
4 int main(int argc, char* argv[]){
5     init_memory(); /* marque toutes les cellules libres */
6     int list=NOTHING; /* liste vide */
7     add_color(&list, "blue");
8     add_color(&list, "black");
9     add_color(&list, "red");
10    print_list(list);
11    free_list(list);
12    print_list(list);
13    return 0;
14 }
```

donne

```
nborie@perceval:~> ./test
```

```
list=red black blue
```

```
list=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` partout
- ▶ solution : mettre `list` à `NOTHING` après l'appel à `free_list`

```
1 int main(int argc, char* argv[]){
2     ...
3     free_list(list);
4     list=NOTHING;
5     print_list(list);
6     return 0;
7 }
```

Les listes

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

```
1  /* libere la 'liste' donnee et assigne a '*list'
2     la valeur NOTHING */
3  void free_list2(int *liste){
4     int tmp;
5     while (*list!=NOTHING){
6         tmp=memory[*list].next;
7         free_cell(*list);
8         *list=tmp;
9     }
10 }
```

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 :

```
1  int main(int argc, char* argv[]){
2      ...
3      print_list(list);
4      int list2=list; /* une jolie copie */
5      free_list2(&list); /* liberation + modification */
6      print_list(list); /* OK */
7      print_list(list2); /* boucle infinie */
8      return 0;
9  }
```