Programmation avancée en C :

Types composés, construction du langage C

Licence informatique 3e année

Université Gustave Eiffel

Les structures

- Types contenant plusieurs données appelées “champs”
- Définis hors d’une fonction
- Dans un `.h` si la structure doit être utilisée ailleurs, dans un `.c` si on souhaite la confiner pour la modularité
- Schéma de définition :

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

- Déclaration d’une variable :
  `struct nom_type nom_var;`
- Accès aux champs : `nom_var.nom_champ`

Choix des champs

- On ne peut mettre que ce dont le compilateur connaît la taille.
- Donc on ne peut pas mettre la structure elle-même.
- Mais on peut mettre un pointeur sur n’import quelle structure.
- On peut aussi mettre une structure déjà définie.

```
struct person{
  int age;
  char* name;
};

struct person_list{
  struct person p;
  struct person_list* next;
};
```


Structures dans la mémoire, et son alignement

- Champs organisés dans l’ordre de leurs déclarations.
- Pour chaque champs, le compilateur fait de l’alignement pour avoir, selon l’implémentation :

  - des adresses multiples de la taille du type du champs

  - ... (d’autres choses choisies raisonnablement par les développeurs de `gcc`)
- Adresse d’une structure = adresse de son premier champ

```
struct s1{
  char c1; /* @ + 0 */
  int i; /* @ + 4 */
  char c2; /* @ + 8 */
};
```

```
struct s2{
  char c1; /* @ + 0 */
  char c2; /* @ + 1 */
  int i; /* @ + 4 */
};
```

Alignement mémoire

```
1  struct s1{
2    char c1; /* @+0 */
3    int i; /* @+4 */
4    char c2; /* @+8 */
5  };
```

```
1  struct s2{
2    char c1; /* @+0 */
3    char c2; /* @+1 */
4    int i; /* @+4 */
5  };
```

- Taille variable suivant l'ordre des champs : `sizeof(struct s1) = 12` et `sizeof(struct s2) = 8`
- Les petits avant les grands pour économiser la taille
- Ne jouez pas aux petits malins ! On ne devine pas la taille des strutures ou l'adresse des différents champs.
  - On utilise `sizeof`.
  - On utilise le nom des champs.

Initialisation et opérations

- `struct foo var` = $\{val_1, val_2, \ldots, val_n\}$
- Seulement lors de la déclaration de `var`, erreur sinon !
- Affectation avec =

```
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 : pour un champ de pointeur, on ne copie que l'adresse.
  Donc si on libère le premier avec `free`, bonjour les dégâts...
- Pas d'opération de comparaison ⇒ à écrire soi-même...

Les champs de bits

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

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


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 ! Il est possible que vous l'utilisez jamais...
- Toujours forcer la signature du type entier associé (`signed int` ou `unsigned int` mais pas `int`)
- Les champs n'ont plus d'adresse !!! Tous les octets de la mémoire ont une adresse, mais pas les bits ! On utilise alors les champs mais pas d'opérateur d'adressage, attention aux arrondis.

Les unions

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

- Zone mémoire que l'on peut voir soit comme un `type1`, soit comme un `type2`, soit ... etc.
- Interprétation multiple de l'espace mémoire à la même adresse
- Utilisés comme des structures
- Taille = taille du plus grand champ

Les unions

- C'est au programmeur de savoir quel champ doit être utilisé.

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

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

donne

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


Les unions

- Utiles pour manipuler des informations exclusives les unes des autres

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

- Peuvent être utilisées anonymement dans les structures

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


Unions complexes

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

```
union color{
  struct{
    unsigned char red, blue, green;
  };
  char name[6];
};
```

- On peut utiliser soit `red`, `blue` et `green`, soit le champ `name`

Les énumérations

- `enum nom` $\{id_0, id_2, \ldots, id_{n-1}\}$
- Pour une variable de type `enum nom`, elle pourra prendre les valeurs id_i.

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

- Valeurs `int` consécutives, commençant à 0 par défaut.
- On peut les modifier (avec une bonne raison !).

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


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

- 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
```


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
- Pas de nouveaux types, juste des synonymes
- 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

- Pour les types structurés, deux modes de définition :

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

struct cell{
  enum cell_type 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;
```

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

if... else...

- `if (condition) instruction1 else instruction2` ou `if (condition) instruction`
- En cas d'ambiguïté, le `else` s'attache au `if` le plus proche.
- Bien utiliser l'indentation et les accolades pour la lisibilité et pour éviter des erreurs.
- Opérateurs de comparaisons des entiers et des réels : `a < b`, `a <= b`, `a > b`, `a >= b`, `a != b`, `a == b` (mais pas =)

  Pas pour les chaînes (il faut des fonctions dans `string.h`)
- Opérateurs logiques : `&&` (et), `||` (ou), `!` (non)
- Convention : `0` = faux et $\neq 0$ = vrai

Quelques astuces

Les conditions sont des entiers. Donc des écritures simples :

`if (value!=0) ...` = `if (value) ...`

`if (value==0) ...` = `if (!value) ...`

Sachant que les codes `ascii` des lettres sont bien rangées ...

```
1 int is_letter(char c){
2   return ((c >= 'a' && c <= 'z') || (c >= 'A' && c<= 'Z'));
3 }
```

Cette fonction renvoie `0` (faux) ou `1` (vrai), utilisable dans un if.

Évaluation paresseuse

- Opérateurs `&&` et `||` paresseux, évalués de gauche à droite
- S'arrêtent dès que possible
  - `(a=0 && foo(b))` ⇒ `foo(b)` n'est jamais appelée
  - `(a=1 || foo(b))` ⇒ `foo(b)` n'est jamais appelée
- Éviter les effets de bord dans une condition (sauf besoin).
- L'ordre des conditions pourrait être important !
- On met alors d'abord celles faciles à calculer, puis celles plus lourd (typiquement des fonctions).
- Aussi vérification de conditions avant l'appel d'une fonction dans la condition de `if`

Priorités

- Du plus au moins prioritaire :

  `()`

  `--` `++` `!` `-` unaire

  `*` `/` `%`

  `+` `-`

  `>` `>=` `<` `<=`

  `==` `!=`

  `&&` `||`

  `=`

- Dans le doute, on met des parenthèses !!!
- Petite astuce de lisibilité : espace autour des opérateurs, sauf ceux en priorité relative

  `delta = b*b - 4*a*c;`

Conseil de style

- Opérateur ternaire : `? :`

  `a = (condition) ? expr1 : expr2;`

  ⇔ `if (condition) a = expr1; else a = expr2;`

  Pratique pour des petits manips, sauf si au prix de lisibilité !
- Un bon code ne contient pas de bloc vide !
- Tests inutiles : pas des tests dont on est sûr du résultat !

```
1 if (n>0){
2   /* .. */
3 }
4 if (n<0){
5   /* .. */
6 }
7 if (n==0){
8   /* .. */
9 }
```

```
1 if (n>0){
2   /* .. */
3 }
4 else if (n<0){
5   /* .. */
6 }
7 else{
8   /* .. */
9 }
```

Boucle `for`

- `for (init; cond; increment) corps`
  - faire `init`
  - évaluer `cond`, tant qu'elle est vraie, faire : `corps`, `increment`
- Condition vide = vraie, donc `for (init; ; increment)` = boucle infinie
- Double initialisation avec l'opérateur virgule

```
int i,n;
for (i=0, n=get_max(); i<n; i++){
  printf("i = %d\n", i);
}
```

- Boucle vide est parfois pratique. Mettre un bloc vide pour lisibilité.

```
1 int strlen(char* s, int n){
2   int i;
3   for (i = 0; (i < n) && (s[i] != 0); i++) {}
4   return i;
5 }
```


Attention à la condition

- La condition est évaluée à chaque tour !
- Attention aux complexités cachées

```
1 void spell(char* s){
2   int i;
3   for (i = 0; i < strlen(s); i++){
4     printf("s[%d] = %c", i, s[i]);
5   }
6 }
```

est quadratique alors qu'on peut faire du linéaire

```
1 void spell(char* s){
2   int i, n;
3   for (i = 0, n = strlen(s); i < n; i++){
4     printf("s[%d] = %c", i, s[i]);
5   }
6 }
```


Boucle `while`

- `while (condition) instruction`
- `while` VS `for` : question de lisibilité

```
1 int scan_and_sum(){
2   int sum=0, n;
3   for (;;){
4     if (scanf("%d",&n) != 1) return sum;
5     sum = sum + n;
6   }
7 }
```

```
1 int scan_and_sum(){
2   int sum=0, n;
3   while (scanf("%d",&n) == 1){
4     sum = sum + n;
5   }
6   return sum;
7 }
```


Boucle `do...while...`

- Ça existe... mais à éviter
- `do instruction while (condition);`
- "Utile" quand on doit passer au moins une fois dans la boucle
- Peut éviter de devoir initialiser des variables...

```
1 int yes_or_no(){
2   char c;
3   do{
4     printf("y/n ? ");
5     scanf("%c", &c);
6   }while(c != 'y' && c != 'n');
7   return (c == 'y') ? YES : NO;
8 }
```

switch

- Fait des saut selon la valeur **entières** !

```
switch (expression) {
    case const1: instruction
    case const2: instruction
    ...
    default: instruction
}
```

- Attention ! Ce sont des sautes, donc `break;` pour les cas aux traitements exclusifs !
- Sans `break`, l'exécution continue.
- Ne pas oublier le `default`, qui permet de traiter les erreurs de valeurs (affichage `stderr` par exemple).

switch

- Plus élégant que plein de `if ... else ...`
- Possibilité d'optimisation par le compilateur

```
1  int get_base(char choice){
2    int base;
3    switch(choice){
4      case 'b': base=2; break;
5      case 'o': base=8; break;
6      case 'd': base=10; break;
7      case 'h': base=16; break;
8      default : fprintf(stderr, "Option inconnue\n");
9    }
10   return base;
11 }
```


break

- Peut servir à sortir du bloc courante (boucle, switch, ...) :
  - si le calcul est fini au milieu d'une boucle,
  - dans certains cas d'erreurs,
  - dans un `switch` avec des cas à traitements exclusifs.

```
int product_foo(int tab[]){
  int i, prod = 1;
  for(i = 0; i < N; i++){
    prod *= foo(tab[i]); /* avec foo(int) couteuse */
    if(prod == 0) break;
  }
  return prod;
}
```

- À utiliser avec parcimonie et bon escient
- À éviter en cas de boucles imbriquées
- À éviter si on peut quitter la fonction (préférer `return`)
- Utile lorsque la sortie de boucle diminue la complexité

continue

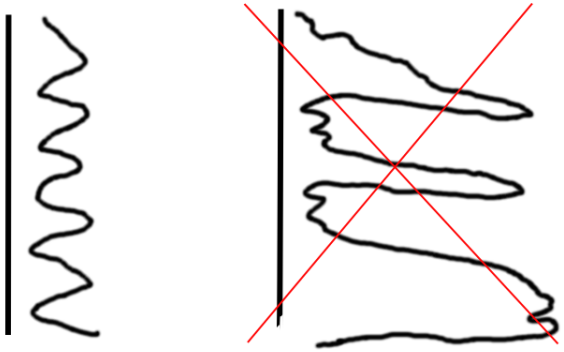
- Sert à sauter un tour de boucle
- Éviter un niveau d'indentation ⇒ lisibilité (surtout si le `else` est gros)

```
1  void print_free_seats(){
2    int i, j;
3    for(i = 1; i < NB_ROWS; i++){
4      /* No seats number 13
5      because of stupid superstitious people */
6      if(i == 13) continue;
7      for(j = 0; j < N_SEATS; j++){
8        if(seat[i][j]) printf("%2d%c ",i,j+'A');
9        else printf("    ");
10     }
11     printf("\n");
12   }
13 }
```

Code équilibré

On préféra toujours un `for` (bien lisible) à un `while` (moins lisible) ou encore un `do ... while` (pénible).

Un dessin vaut mieux qu'un long discours, ainsi, un code équilibré a la silhouette suivante :



printf

- Fonction d'affichage (`stdio.h`) : `printf("format", ...);`
- Il faut autant de variables que de `%...` .
- Variables indiquées avec :

  `%d` : `int` `%f` : `double, float`
  `%c` : `char` (caractère) `%s` : `char*` (chaîne)
  `%%` pour le caractère `'%'`
- Ne pas afficher une chaîne directement avec `printf(str);` !
- Voir `man 3 printf` pour plus d'options
- L'affichage est bufferisé.
  - L'affichage se déclenche quand il y a un `'\n'`.
  - L'affichage se déclenche quand le buffer est plein.

Messages d'erreurs

- `printf` est bufferisée.

```
1 int main(int argc, char* argv[]){
2   char* nom = "Jean";
3   printf("Ou se trouve l'erreur de segmentation");
4   nom[34] = 'y';
5   return 0;
6 }
```

  Il n'affiche rien, une erreur de segmentation l'arrête avant.
- La sortie d'erreur : `fprintf(stderr, "format", ...);`
- Même fonctionnement que `printf` mais non bufferisée !
- **Exemples :**

  `perror` pour les erreurs systèmes classiques, voir le `man` !

  Pour `fopen`, `if (file == NULL) ...`

```
fprintf(stderr, "Echec de l'ouverture du fichier %s \n",
file);
```

  Pour `malloc`, `if ((C = (Cellule*)malloc()) == NULL) ...`

scanf

- Fonction de saisie au clavier (entrée standard, en `stdio.h`)
- ressemble à `printf`, mais :
  - que des variables, pas de constantes ;
  - `&` devant les variables, sauf pour les chaînes ;
  - la chaîne de format n'est pas affichée ;
  - espaces (tab, retourne à la ligne) ignorés ;
  - saisie bufferisée par ligne ;
  - les entrées sont délimitées par les espaces ;
  - renvoie le nombre de variables saisies correctement.

scanf - exemple

```
1  int main(int argc, char* argv[]){
2    int a;
3    char s1[32];
4    char s2[32];
5    scanf("%d␣%s␣%s", &a, s1, s2);
6    printf("%d␣%s␣%s", a, s1, s2);
7    return 0;
8  }
```

donne avec les entrées suivantes :

```
$>./a.out
5  hello 32   abc
5 hello 32
$>
```


scanf

- Sans vérification de type !

```
1  int main(int argc, char* argv[]){
2    int a, b, c, n;
3    n = scanf("%d␣%d␣%d", &a, &b, &c);
4    printf("%d␣%d␣%d␣%d\n", n, a, b, c, d);
5  }
```

donne pour les entrées suivantes :

```
$>./a.out
4 8 hello
2 4 8 4198592
$>
```


scanf

- Lecture de caractères : attention à ce qu'il reste dans le buffer !!!

```
1  #define YES 1
2  #define NO 0
3
4  int yes_or_no(){
5    char c;
6    do{
7      printf("y/n␣?␣");
8      scanf("%c", &c);
9    } while (c != 'y' && c != 'n');
10   return (c == 'y') ? YES : NO;
11 }
```

donne avec les entrées suivantes :

```
nborie@perceval:~> ./test
y/n ? u
y/n ? y/n ? y
nborie@perceval:~>
```