8- LANGAGE C

LES STRUCTURES

Loïc Cuvillon
l.cuvillon@unistra.fr

Sommaire chapitre 8

La structure en C

- rassemblement de plusieurs variables (de types différents ou non) sous un seul nom
- pour faciliter l'organisation et la manipulation de données complexes

- définition: `struct` nom_du_type_de_structure
  { définition } ;

Attention: un point virgule après chaque déclaration.

- Exemples :

```
struct coureur  {
    char    nom[30] ;
    char    prenom[30] ;
    int     dossard ;
    float   temps ;
                };
```

```
struct point
{
        int x;
        int y;
};
/*coordonnées d'un point*/
```

Les Structures

- `struct` nom_du_type_de_structure définit un nouveau type

- Les variables dans une structure sont appelées *champs* ou *membres*

- Déclaration de variables de type structure (idem. autre variables):

```
struct point   ptA;
struct coureur    Bolt, Lewis;
```

- Initialisation (syntaxe uniquement valable à la declaration !) :

```
struct point   ptA = {10,6};
struct coureur   Bolt = {"Bolt","Usain",1,9.58};
```

Les Structures

- Autres exemples :

```
struct triangle
{
    struct point ptA;
    struct point ptB;
    struct point ptC;
};
```

```
struct chaine_elts
{
  int data;

  struct chaine_elts *next;
  /*pointeur sur une struct*/

};
```

/*les membres peuvent être des structures*/

/*un membre peut être un pointeur sur structure identique (autoréférence) */

Les Structures

- Accès aux champs/membres de la structure par l'opérateur ' . ' :

```
ptA.x = 3 ;
ptA.y = -2;
bolt.temps = 9.54 ;
strcpy(loic.name,"cuvillon");
```

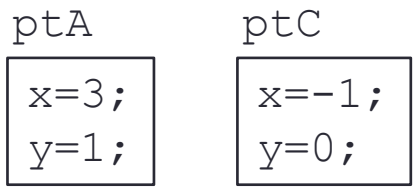
- Si les structures sont imbriquées :

```
struct triangle triangle1;
triangle1.ptA.x = 0;
triangle1.ptA.y = -2;
```

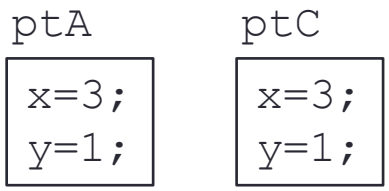
Assignation de structure

- On peut assigner des variables de même type de structure entre elle:

```
struct point ptC={-1,0}, ptA={3,1};
```




```
ptC = ptA;
```



assignation membre à membre :
chaque membre de `ptA` copié dans le membre correspondant de `ptC`.

Assignation de structure

- On peut donc passer des structures par valeur à une fonction et retourner une structure via `return`.

- On préfère cependant utiliser un passage de pointeurs (adresses), en particulier pour de larges structures

---

Rappel : L'assignation est valide entre 2 variables de même type :

```
int a,b;                    a = b;
int *p_a, *p_b;             p_a = p_b;
struct point ptC, ptA;      ptC = ptA;
```

mais l'assignation est invalide entre 2 tableaux de variables en C:

```
int tabA[3], tabB[3];       tabA = tabB ;
```

car `tabA` et `tabB` sont les pointeurs (adresses) constants sur le 1er élément du tableau, donc sont non modifiables.

Les Structures et les fonctions

- passage par pointeur d'une structure (l'approche la plus usuelle) :

```
void initialise_point (struct point *p_point )
{
    (*p_point).x = 0;
    (*p_point).y = 0;
}

int main()
{
struct point ptA;
initialise_point(&ptA);
}
```

parenthèses nécessaires car
'*' est de plus faible priorité que '.'

Les Structures et les fonctions

- Raccourci d'écriture : l'opérateur '->' permet d'accéder au membre/champ si on a un pointeur sur la structure

```
struct point ptA;
struct point *p_ptA = &ptA;
```

```
p_ptA->x = 3   ⟷   (*p_ptA).x = 3;
```

Les Structures et les fonctions

- passage par pointeur d'une structure (l'approche la plus usuelle) :

(et la notation la plus usuelle):

```
void initialise_point (struct point *p_point )
{
    p_point->x = 0;
    p_point->y = 0;
}

int main()
{
struct point ptA;
initialise_point(&ptA);
}
```

Les tableaux de structures

- Déclaration d'un tableau de structures :

```
struct coureur  participants[16];
stuct point p[3];
```

- Initialisation d'un tableau de structure lors de la déclaration :

```
struct point  p[3]={0,2,3,-1,5,2};
struct point  p[3]={{0,2},{3,-1},{5,2}};
```

- Accès au membre des structures du tableau :

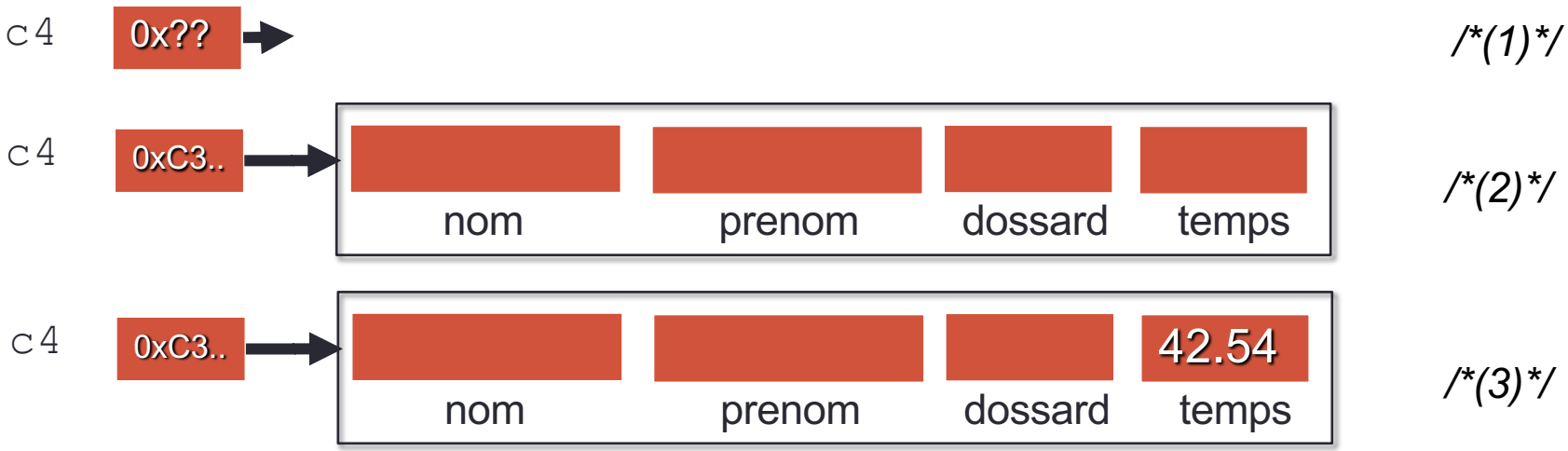
```
p[1].x = 3;          ou          (pt+1)->x = 3;
```

Les tableaux de structures

- Allocation dynamique (d'une structure ou d'un tableau de structure):

```
struct coureur *c4 ;          /* alloue un pointeur mais pas de mémoire
                                 pour la structure pointée (1)*/
c4=(struct coureur*) malloc(1*sizeof(struct coureur));
                                                    /*(2)*/

c4->temps = 42.54 ;
c4[0].temps=42.54;                                  /*ou (3)*/
```



Sommaire chapitre 8

typedef

- instruction pour définir un synonyme pour un type de données

```
typedef   type   nom_type_synonyme
```

- Exemple :

```
typedef  int  longueur;
        /* rend longueur synonyme de int */
```

ainsi

```
{…
longueur a, b=3;
a=b+4;
}
```

typedef

- Souvent utilisé pour alléger la syntaxe avec les structures :

```
typedef struct {char   nom[30] ;
                float  temps ;
               } Coureur;
```

On utiliser alors le nouveau type `Coureur` pour la structure :

```
Coureur bolt;
Coureur *p = (Coureur*) malloc (sizeof(Coureur));


/*sans typedef, on aurait: */
 struct coureur bolt;
 struct coureur *p = (struct coureur*) malloc(sizeof(stru…
```

typedef et structures

- Où mettre la définition des structures (avec ou sans typedef) :
  - un seul fichier source → en tête de fichier, après les #include
  - plusieurs fichiers sources → dans un fichier entête (header) qui sera inclus dans tous les sources avec #include "my_structs.h"

my_structs.h:

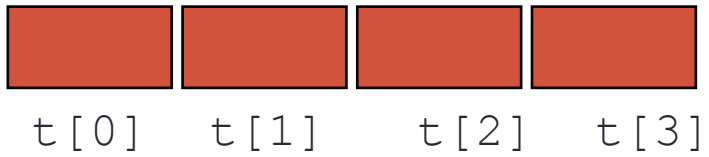
```
typedef struct {   char   nom[30] ;
                   float  temps ;
                } Coureur;
```

Sommaire chapitre 8

Liste

Liste : Suite ordonnée d'un nombre variable d'objets de même type.

Représentation naturelle → liste contiguë (tableau)



- En particulier, le tableau de structure :

```
typedef struct {char name[20]; float time;} coureur;
coureur liste[4];
```

Note: On peut ou pas choisir d'indiquer la fin courante de la liste avec un nom vide par exemple (" " ≅ '\0').

Exemple de Liste

```
typedef struct { char name[20];
                 float time; } coureur;
                    /* définition d'un type pour la struct */


int main ()
{   coureur liste[10];                          /*allocation statique*/

    strcpy(liste[0].name,"bolt");               /*initialisation liste*/
    liste[0].time = 9.58;
    strcpy(liste[1].name,"lemaitre");
    liste[1].time = 9.92;
    sprintf(liste[2].name,"");                  /*dernier élément actuel:
                                                  chaîne vide*/

    affiche_liste(liste);
    return 0;
}
```

Exemple de listes (via allocation dynamique)

```
int main ()
{
  coureur *p_liste;                          /*allocation dynamique*/
   p_liste = (coureur*) malloc(10*sizeof(coureur));

   strcpy(p_liste[0].name,"bolt");              /*initialisation*/
   p_liste[0].time = 9.58;
   sprintf(p_liste[1].name,"lemaitre");
   p_liste[1].time = 9.90;
   sprintf(p_liste[2].name,"");      /* ou  (p_liste+2)->name */


   affiche_liste(p_liste);
   return 0;
}
```

Exemple de manipulation de liste

- On affiche les éléments de la liste tant que le dernier élément n'est pas trouvé :

```
void affiche_liste( coureur* p_liste )
{
    while ( strcmp(p_liste->name,"") )
              /*tant que name différent de la chaine vide*/
    {
        printf("%s %f\n", p_liste->name, p_liste->time);
        p_liste++;
         /*incrémente le pointeur local d'un élément
         donc pointe sur l'élément suivant ! */
    }
}
```

Exemple de manipulation de liste

- Ou encore :

```
void affiche_liste( coureur* p_liste )
{
    int i=0;

    while ( strcmp(p_liste[i].name,"") )
              /*tant que différent de la chaine vide*/
    {
        printf("%s %f\n", p_liste[i].name, p_liste[i].time);
        i++;
    }
}
```

Notes

- `structA = structB;` → assignation; ok

- Mais ~~`struct A == struct B`~~ → test égalité relationnel, not ok

Test d'égalité non défini pour une structure car ambigüe, en particulier si il y a des pointeurs (égalité de l'adresse ou du contenu pointé ?)

→ il faut implémenter sa propre fonction de test d'égalité de 2 structures.