Les structures

julien.brajard@upmc.fr revus par maria.potop-butucaru@lip6.fr

Polytech'UPMC

Octobre 2018

Plan du cours

1 Structures de données

1. Structures de données
   - Introduction
   - Listes et arbres
   - Autres variables "structurées"

Définition

Une structure est un type de données composé de plusieurs eléments de type quelconque appelés champs ou membres.

Les structures permettent de regrouper des informations de types distincts mais ayant un lien sémentique fort pour le programmeur.

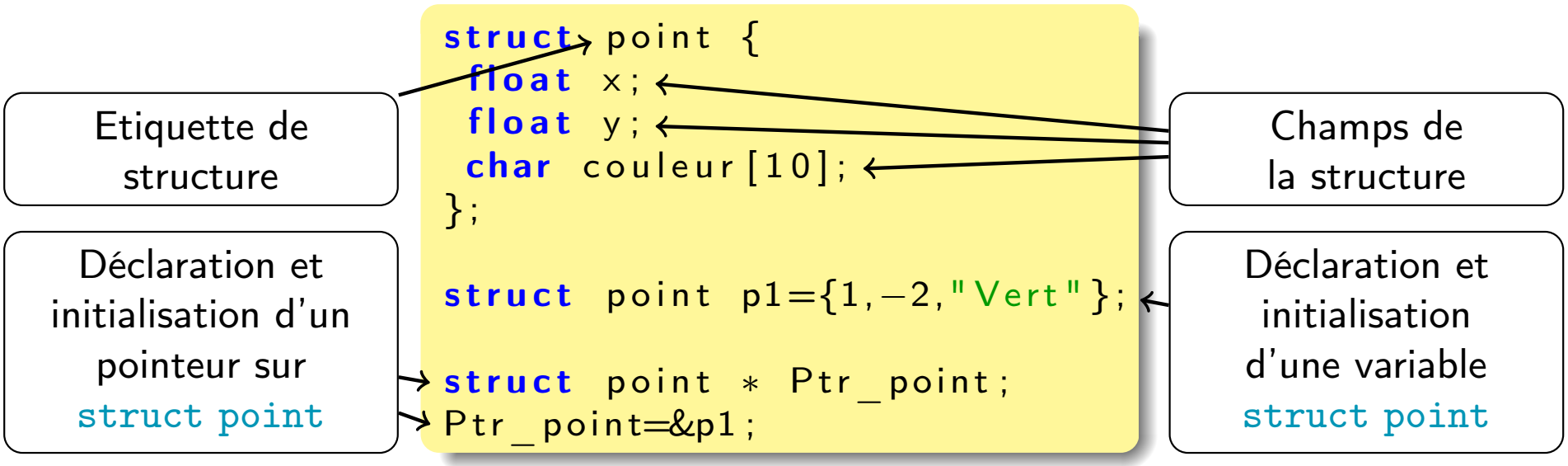
- Nom, Prénom, Date, lieu de naissance, adresse $\rightarrow$ **Identité**
- Jour, Mois, Année $\rightarrow$ **Date**
- Abscisse, Ordonnée $\rightarrow$ **Point**

Notes

- Les structures complètent la notion de tableau car il devient possible de regrouper des éléments de types différents.
- Les fonctions en C peuvent renvoyer une structure.

Déclaration et initialisation

Un modèle de structure se défini de la façon suivante :



- L'étiquette de structure permet de nommer le modèle
- L'initialisation est analogue à celle des tableaux.

Accès aux membres de la structure

Pour accéder aux membres de la structure, on utilise l'opérateur . (point)

Syntaxe : *Nomdevariable.membre*

Exemple

```
struct point {
 float x;
 float y;
 char couleur[10];
};
int main() {
 struct point origine;
 origine.x=0;
 origine.y=0;
 strcpy(origine.couleur,"noir");
 printf("abscisse : %f\n",pt.x);
 printf("ordonnée : %f\n",pt.y);
 printf ("couleur : %s\n",pt.couleur);
 return(0);
}
```

Remarques

- Les noms des champs sont locaux à la structure

```c
#include <stdio.h>
struct point {
 float x;
 float y;
 char couleur[10];
};
int main() {
 struct point M={1.1,0,"rouge"};
 float x=5.1;
 printf("x=%f\n",x);
 printf("M.x=%f\n",M.x);
 return(0);
}
```

Test d'exécution

x=...
M.x=...

- L'usage veut que les modèles de structure soient placés entre les directives préprocesseur (#) et les prototypes des fonctions.

Opération sur les structures

- Récupération d'adresse par &
- Accès au membres par . (point)
- Affectation globale pour des variables d'un même modèle.

```
struct point M={1.1,0,"rouge");
struct point N;
N=M;
}
```

Petit truc : cela permet de faire des copies de tableaux sans boucle. Ici `M.couleur` est un tableau de caractères et il est copié dans le tableau `N.couleur`.

Attention : Cette astuce ne fonctionne que pour les tableaux statiques.

Structures et fonctions

- Possibilité de passer une strutcture en paramètre d'une fonction

```
void affichePoint (struct point pt) {
    printf("abscisse : %f\n",pt.x);
    printf("ordonnée : %f\n",pt.y);
    printf ("couleur : %s\n",pt.couleur);
}
```

- Possibilité de retourner une structure

```
struct point construirePoint (float x, float y, char couleur[]) {
 struct point pt;
 pt.x=x;
 pt.y=y;
 strcpy(pt.couleur,couleur);
 return pt;
}
```

Comparaison de structure

Il n'existe pas d'opérateur de comparaison global.
Il faut comparer champ par champ

```
int comparepoint (struct point P1, struct point P2) {
 /* renvoie 1 si les points sont égaux 0 sinon*/
 int comp=0;
 if (((P1.x==P2.x) && (P1.y==P2.y)) &&
  !strcmp(P1.couleur,P2.couleur)) {
  comp=1;
 }
 return (comp);
}
```

Pointeurs sur une structure

- Possibilité de passer une structure par adresse

```
void symetrie (struct point *pt) {
 (*pt).x= -(*pt).x;
 (*pt).y= -(*pt).
}
```

- Pour alléger l'écriture, utilisation d'un symbole spécial : ->

```
void symetrie (struct point *pt) {
 pt->x= -pt->x;
 pt->y= -pt->y;
}
```

remarques

- Il est souvent préférable de passer les structures par adresse.
- Les pointeurs sur des structures sont très utilisés.

Tableaux de structures

- Les tableaux de structures sont très utilisés.

```
struct point Segm[2]={{0,0,"rouge"},{1,2.3,"vert"}};
float dx,dy ;
dx=Segm[1].x - Segm[0].x;
dy=Segm[1].y - Segm[0].y;
```

- On préfère généralement utiliser des tableaux de pointeurs de structure.

```
struct point M1={0,0,"rouge"};
struct point M2={1,2.3,"vert"};
struct point *Segm[2]={&M1,&M2};
```

Allocation dynamique de mémoire

La taille d'une structure est donnée par l'opérateur `sizeof()`

```
struct point M1={0,0,"rouge"};
struct point M2={1,2.3,"vert"};
struct point *Segm;
Segm=(struct point *)malloc(2*sizeof(struct Point));
Segm[0]=M1;
Segm[1]=M2;
```

Attention : La taille d'une structure est différente de la somme des tailles des champs qui la compose.

Types synonymes : typedef

La fonctionnalité typedef permet de définir des types synonymes.

```
typedef struct point {
int x;
int y;
char couleur[10];
} Point;

typedef int * PtrEntier
```

```
int main() {
  Point P={1,2,"vert"};
  PtrEntier pn;
```

équivalent à :

```
int main() {
struct point P={1,2,"vert"};
int * pn;
```

- Le type synonyme peut être utilisé dans toutes les expressions (en particulier les conversions () et sizeof).
- Il rend les noms des types plus courts et intuitifs.
- L'instruction typedef est placée aux mêmes endroits du code que les modèles de structures et commencent par une majuscule (convention).

Que peut-on déclarer comme champ de structure ?

- Les types classiques (`char`, `int`, `float`, ...) ;
- Les tableaux statiques ;
- Les pointeurs ;
- D'autres structures ;
- Des pointeurs sur une structure (y compris elle-même). Ce sont alors des structures autoréférentielles ou récursives.

1. Structures de données
   - Introduction
   - Listes et arbres
   - Autres variables "structurées"

Quelques définitions

Structure récursive

Structure de données dans laquelle un membre est un pointeur vers une variable de la même structure.

- Données éparpillées en mémoire reliées par des pointeurs.

Liste chaînée

Un membre pointe vers la variable suivante de la liste.

Liste doublement chaînée

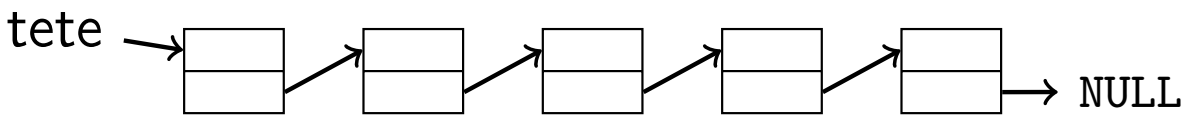
Un membre pointe vers la variable suivante et un autre vers la variable précédente.

arbre binaire

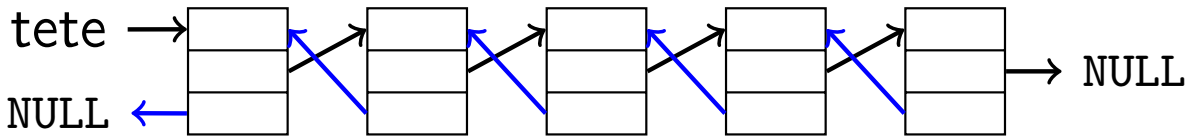
Deux pointeurs pointent vers deux membres suivants (les fils).

Illustration

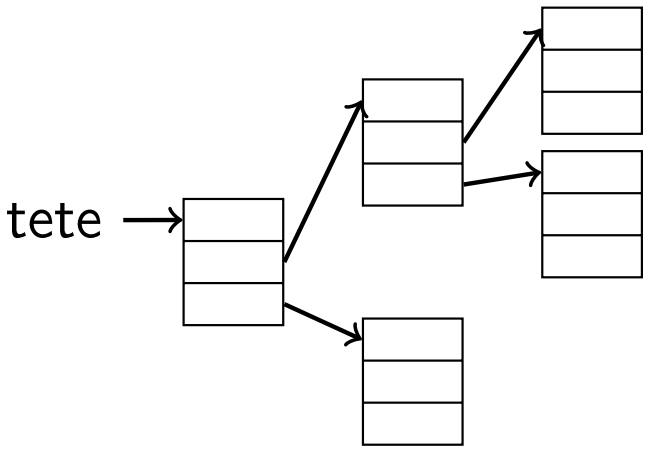
La liste chaînée :



La liste doublement chaînée :



L'arbre binaire :



Modèles

Liste chaînée

```
struct ListeSimple{
 type_val val;
 struct ListeSimple *suivant;
};
```

Liste doublement chaînée

```
struct ListeDouble{
 type_val val;
 struct ListeDouble *suivant;
 struct ListeDouble *precedent;
};
```

Liste doublement chaînée

```
struct Arbre{
 type_val val;
 struct Arbre *fils_droit;
 struct Arbre *fils_gauche;
};
```

Utilisation

On écrit des fonctions spécifiques pour manipuler les listes :

- Création d'un maillon
- Ajout d'un maillon
- Supression d'un maillon
- Traitement (affichage, recherche, concaténation

Création d'un maillon

Exemple de modèle de liste

```
struct liste{
 int val;
 struct liste *suivant;
};
```

Fonction creation

```
struct liste * creation() {
 struct liste * nouveau ;
 nouveau = (struct liste *) malloc (sizeof(struct liste));
 nouveau -> val = 0 ; //par exemple
 nouveau -> suivant = NULL ;
 return(nouveau) ;
```

Supression

Supprime l'élément dont la valeur est égale à `val`.

Fonction creation

```
struct liste *supprime(int val, struct liste *tete) {
 struct liste *maillon = tete ;
 struct liste *prec = tete ;
 while (maillon != NULL && maillon -> valeur != val) {
  prec = maillon ;
  maillon = maillon -> suivant ;
 }

 if (maillon != NULL) {
  if (maillon==tete)
   tete = prec -> suivant;
  else
   prec -> suivant = maillon -> suivant;
  free(maillon);
 }
}
```

Autre structure récursive : l'arbre

Vocabulaire

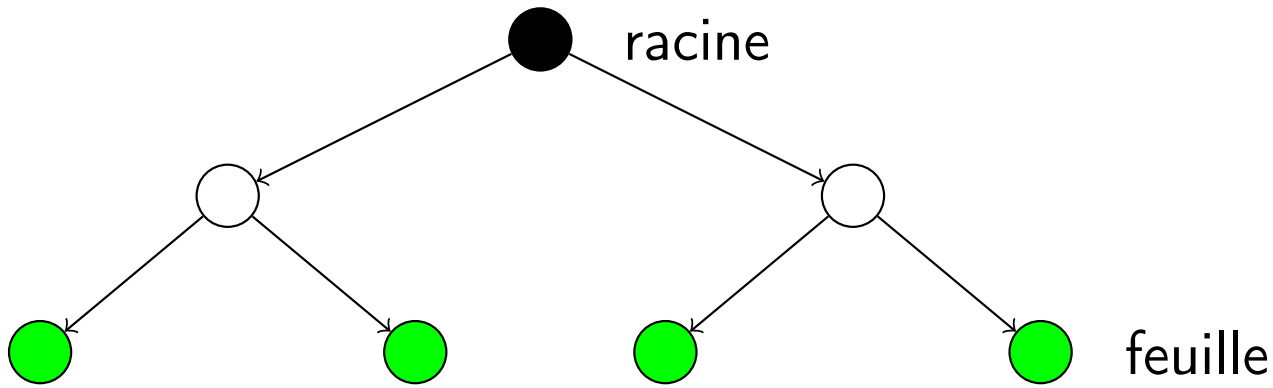
Noeud de l'arbre : Elément de base de l'arbre qui contient l'information ;

Fils d'un noeud N : Noeud pointé par le noeud N ;

Père d'un noeud F : Noeud qui pointe sur F ;

Racine de l'arbre : Noeud (unique) qui n'a pas de père ;

Feuille de l'arbre : Noeud qui n'a aucun fils (pointeurs = `NULL`).



Deux types d'arbres

Arbre binaire

Arbre dont chaque noeud a au plus deux fils (qu'on désigne fils gauche et fils droit)

Arbre n-aire

Arbre dont chaque noeud a un nombre indéterminé de fils. Les fils ont donc une structure de liste chaînée.

Modèle de structure d'un arbre n-aire

```
struct arbre {
  int valeur ;
  struct arbre *fils ;
  struct arbre *frere;
}
```

1. Structures de données
   - Introduction
   - Listes et arbres
   - Autres variables "structurées"

Les énumérations

Les énumérations permettent de définir des constantes.
Elles accroient la lisibilité des programmes.

Déclaration de constante :

```
enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

En fait, les constantes sont des entiers (dans l'exemple : `LUNDI==0`, `MARDI==1`, etc.)

Exemple d'utilisation

```
int j=LUNDI ;
if (j==LUNDI) printf ("%d, c'est le jour du cours d
'info\n",j);
```

Test d'exécution

```
0, c'est le jour du cours d'info
```

Une variable de type enum

Il est possible de définir un type enum.

```
enum jour {LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMANCHE};
```

Exemple d'utilisation

```
enum jour j1, j2 ;
j1 = LUNDI ;
j2 = MARDI ;
```

Les unions

Les unions permettent de définir une variable qui a un type "variable" parmi plusieurs.

La syntaxe est très proche de celle utilisée pour les structures :

```
union MonUnion {
 int entier;
 double reel;
 char chaine[100];
}
```

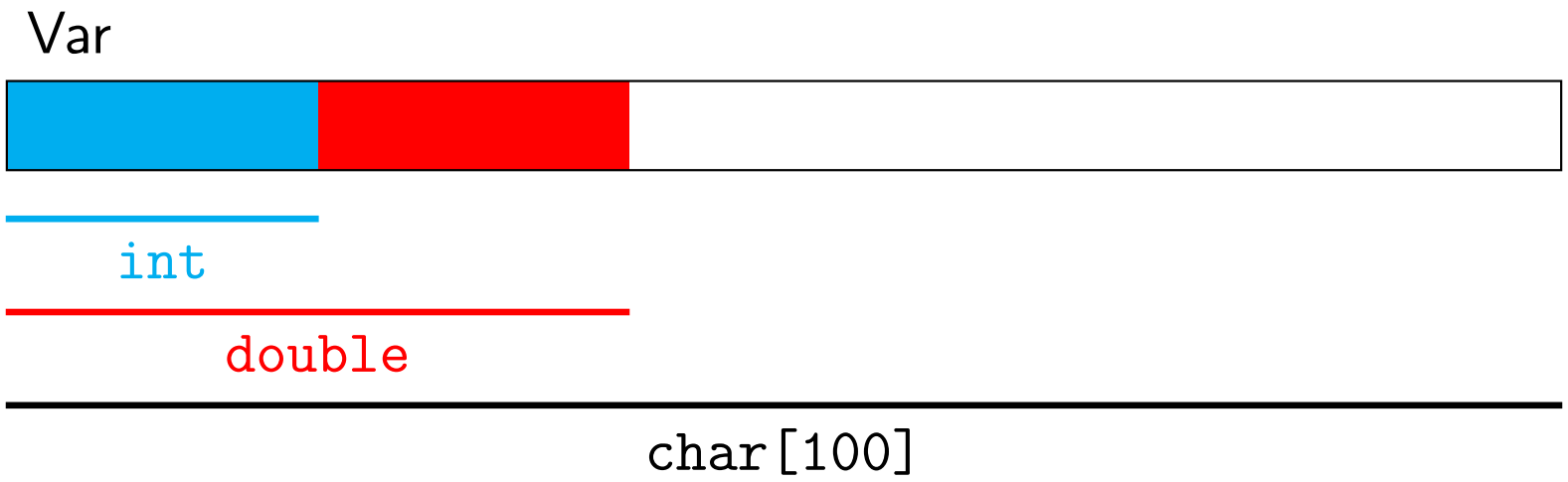
Utilisation (comme pour les structures) :

```
union MonUnion Var ;
Var.entier = 2 ;
```

La taille de la variable est égale à la taille du type le plus grand (dans l'exemple c'est `chaine` = 100 octets).

Les unions

Tous les champs de la variable de type `union` partage le même espace mémoire




```
union MonUnion Var;
Var.entier = 200;
printf("Val int = %d\n",Var.entier);
Var.reel = 1200.05;
printf("Val double = %lf\n",Var.reel);
printf("Val int = %d\n",Var.entier);
```

Test d'exécution

```
Val int = 200
Val double = 1200.05
Val int = 858993459
```

Adresse de la variable

```
union MonUnion variable;
printf("Adresse de l'union = %p\n",&variable);
printf("Adresse de la partie entière = %p\n",&variable.entier);
printf("Adresse de la partie réelle = %p\n",&variable.reel);
```

Test d'exécution

```
Adresse de l'union = 0023FF70
Adresse de la partie entière = 0023FF70
Adresse de la partie réelle = 0023FF70
```

Modèle de structure

```
enum type_t {ENTIER,CARACT};
struct touche {
  enum type_t type;
  union {
    int ent;
    char car;
  }
};
```

Dans le programme :

```
struct touche t;
int n,res;
res=scanf("%d",&n);
if (res==0) {
  scanf("%c",&t.car);
  t.type=CARACT;
}
else {
  t.ent=n;
  t.type=ENTIER;
}
```