Les structures

thibaut.lust@lip6.fr

Polytech Sorbonne

2020

`https://moodle-sciences.upmc.fr` (cours Informatique Générale EPU-R5-IGE)

Cours basé sur les diapositives créées par Julien Brajard

Plan du cours

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

Linus Torvalds (1969-)
créateur de Linux

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 sémantique fort pour le programmeur.

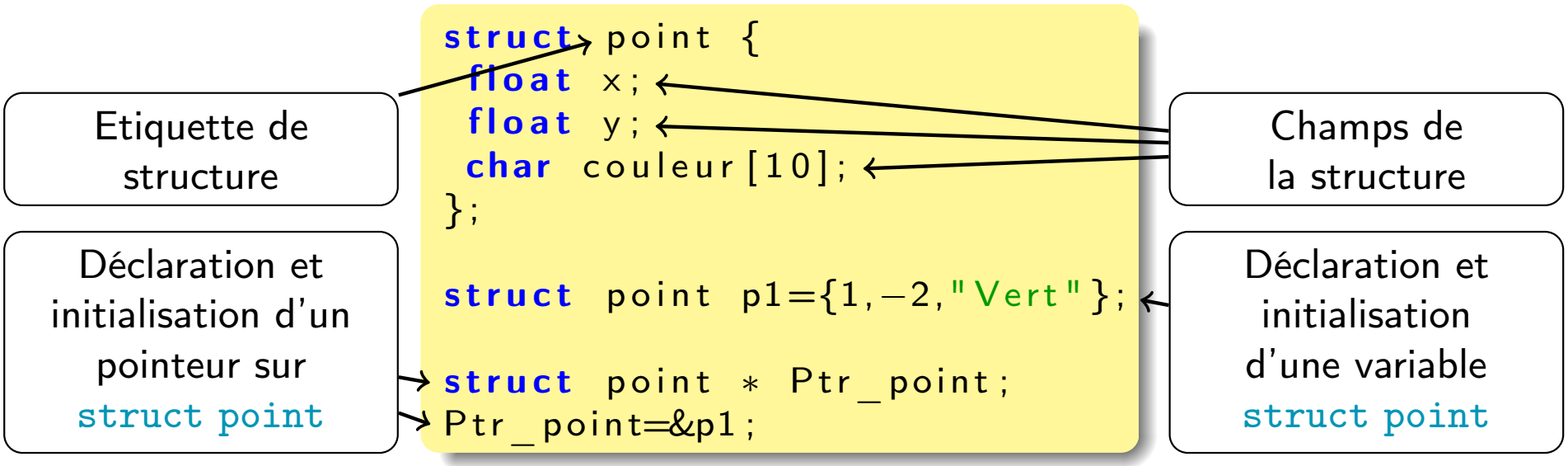
- Nom, Prénom, Date, Lieu de naissance, Adresse $\rightarrow$ **Identité**
- Jour, Mois, Année $\rightarrow$ **Date**
- Abscisse, Ordonnée $\rightarrow$ **Point**

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 définit 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",origine.x);
 printf("ordonnée : %f\n",origine.y);
 printf ("couleur : %s\n",origine.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 = 5.1
M.x = 1.1

- 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 aux 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 structure 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;
}
```

the convention is to use 0 to indicate "false" and any non-zero value to indicate "true"

Pointeurs sur une structure

- Possibilité de passer une structure par adresse

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

- 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.

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

typedef int * PtrEntier;
```

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

équivalent à :

```c
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
   - Autres variables "structurées"
   - Listes et arbres

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];
}
```

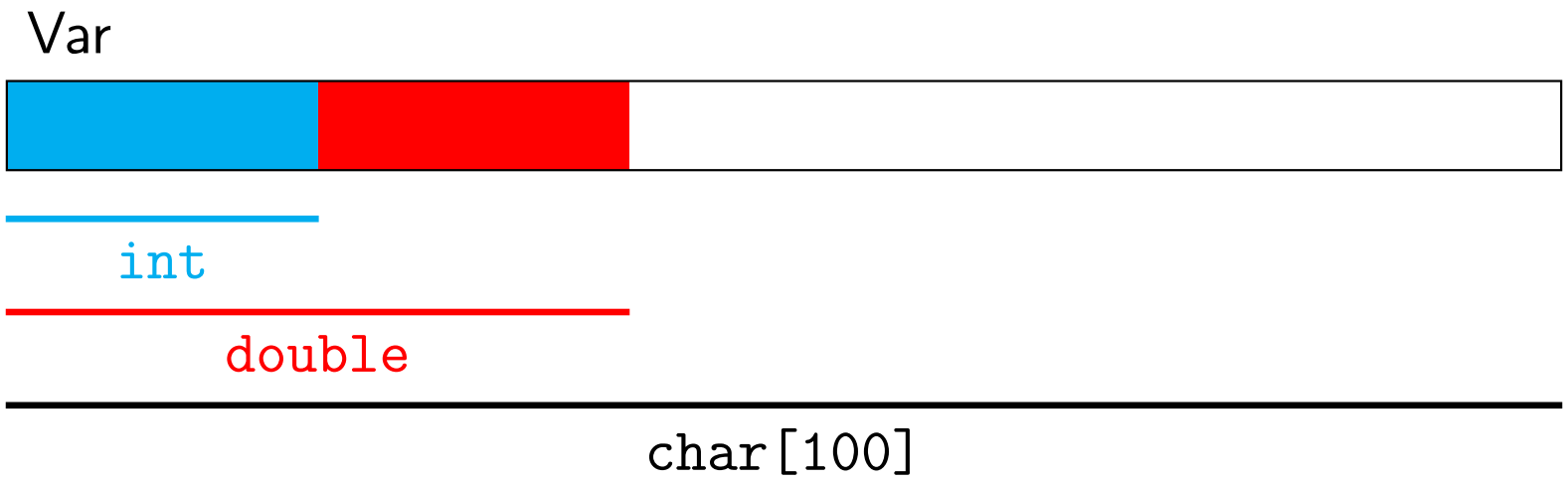
Utilisation (comme pour les structures) :

```
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

```c
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;
}
```

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

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

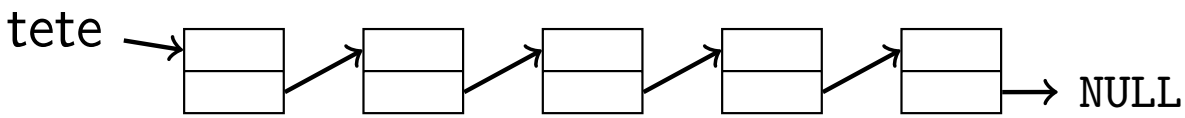
Un membre pointe vers la variable suivante et un autre vers la variable précédente.

Arbre binaire

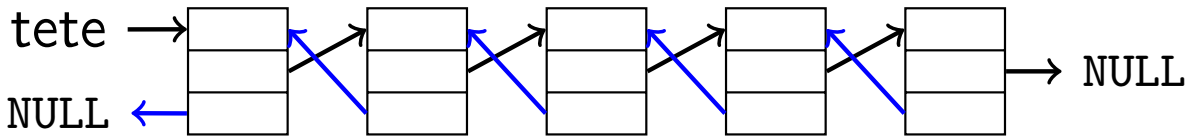
Deux pointeurs pointent vers deux membres suivants (les fils).

Illustration

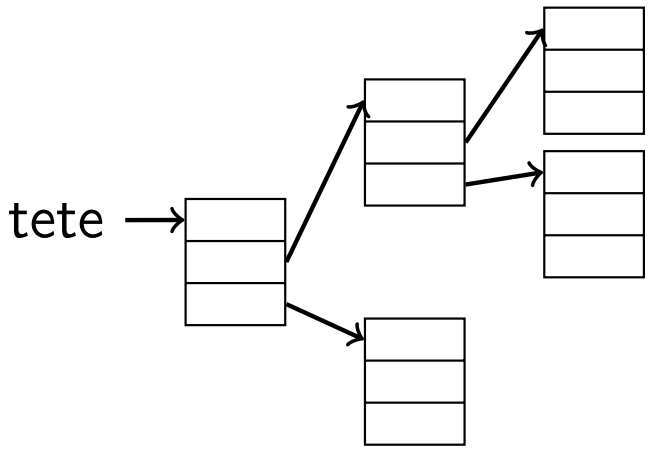
La liste chaînée :



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;
};
```

Arbre binaire

```
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
- Suppression d'un maillon
- Traitement (affichage, recherche, concaténation, etc.)

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);
}
```

Insertion d''un élement au début

Fonction insertion au début

```
struct liste* insertionDebut(struct liste* L, int val){
  struct liste* nouveau;
  nouveau = (struct liste*) malloc (sizeof(struct liste));
  nouveau -> val = val;
  nouveau -> suivant = L;
  return nouveau;
}
```

Dans le programme :

```
struct liste* L1 = creation();
for (int i=0;i<10;i++){
  L1=insertionDebut(L1,i+1);
}
```

Affichage des éléments de la liste

Fonction d'affichage

```
void affichageListe(struct liste* L){
  struct liste* courant = L;
  printf("[ ");
  while(courant!=NULL){
    printf("%d ",courant->val);
    courant = courant -> suivant;
  }
  printf("]\n");
}
```

Dans le programme :

```
printf("Affichage des éléments de la liste : \n");
affichageListe(L1);
```

Suppression du premier élément de la liste

Fonction de suppression du premier élément

```
void supprimeDebut(struct liste** L){
  struct liste* temp;
  if ((*L)!=NULL){
    temp=(*L)->suivant;
    free(*L); /*désallocation de la case supprimée*/
    *L=temp;
  }
}
```

Dans le programme :

```
supprimeDebut(&L1);
```

Suppression d'un élément

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

Fonction suppression d'un élément

```
void supprimeElement(int val, struct liste** L){
  struct liste* cour;
  struct liste* prec;
  if ((*L)!=NULL){ /*si L est vide, rien à faire*/
    cour=*L;
    if (cour->val==val){
      supprimeDebut(L);
    }else{
      prec=*L;
      cour=cour->suivant;
      while ((cour!=NULL) && (cour->val!=val)){
        prec=cour;
        cour=cour->suivant;
      }
      if (cour!=NULL){
        prec->suivant=cour->suivant;
        free(cour);  /* désallocation */
      }
    }
  }
}
```

Suppression d'un élément

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

Dans le programme :

```
supprimeElement(5, &L1);
```

Libération de la mémoire

Il ne faut pas oublier de librérer la mémoire utilisée par liste à la fin de son utilisation !

Fonction de suppression de tous les éléments de la liste

```
void supprimeListe(struct liste** L){
  while ((*L)!=NULL){
    supprimeDebut(L);
  }
}
```

Dans le programme :

```
supprimeListe(&L1);
```

Autre structure récursive : l'arbre

Vocabulaire

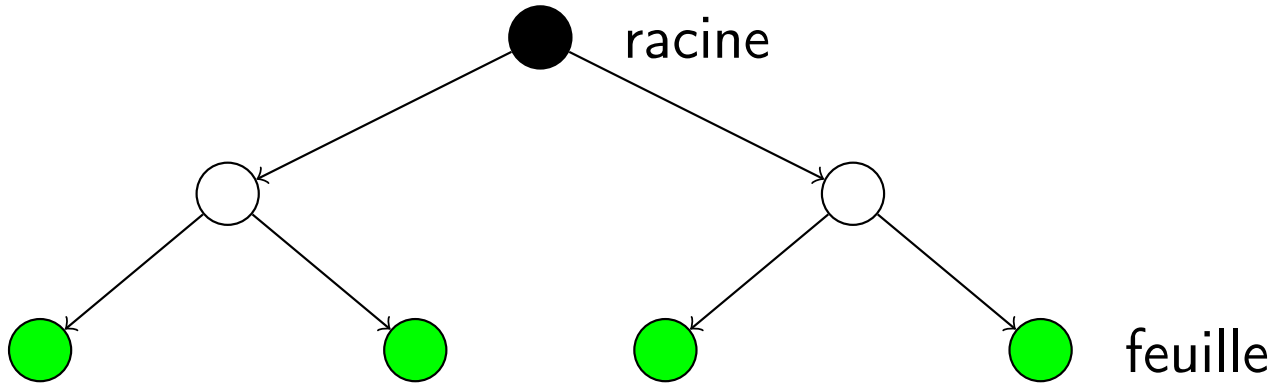
Noeud de l'arbre : Elément de base de l'arbre qui contient l'information ;

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;
}
```