# TYPES STRUCTURÉS

1. Les types en C
2. Le type structure
3. Opérations sur les structures
4. Définition de type
5. Remarques sur `struct`
6. Le type union
7. Remarques sur `union`
8. Énumération
9. Remarques sur `enum`
10. Le type tableau
11. Représentation
12. Tableau en argument
13. Tableau à plusieurs dimensions
14. Chaîne de caractères

# Les types en C

- Types de base
  - Caractères et entiers
  - Flottants
- Types structurés
  - Structures
  - Unions
  - Tableaux
- Autres types
  - Pointeurs (adresses)
  - Fonctions

Deux types sont compatibles s'ils ont la même spécification après développement des noms de types définis par `typedef` et normalisation (par exemple, remplacement de `long` par `long int`).

# Le type structure

Entité regroupant plusieurs objets de types divers

- Exemple : fiche individuelle d'une personne

| | |
|---|---|
| nom | Dupont |
| naissance | 1960 |
| salaire | 200000 |

- Description du type

```
struct perso {
     char nom[16];
     int naissance;
     long salaire;
};
```

`nom, naissance, salaire`
sont les *membres* (ou champs) de la structure.
`perso` en est le *label*.

- Déclaration de variables dans le type

```
struct perso anne, jean;
struct perso x = {"Dupont", 1960, 200000};
```

# Opérations sur les structures

Seules opérations possibles sur les structures :

- accès aux membres (.)

```
...
anne.naissance = 1965;
strcpy(anne.nom, "Durand");
...
```

- copie et affectation
  en particulier, une structure peut être argument d'une fonction, et être valeur de retour d'une fonction

```
...
jean = x;
x = anne;
...
```

- prise d'adresse (&)

```
long prime (struct perso * x) {
   return (*x).salaire += 20000;
}
...
printf("%ld\n", prime(&jean));
...
```

# Définition de type

- Un type peut être nommé au moyen de "`typedef`" : accroît la lisibilité et la portabilité

```
typedef struct perso {
     char nom[16];
     int naissance;
     long salaire;
} personne;
...
personne jean, anne;
struct perso x;
...
```

- Le label ("`perso`") de la structure est facultatif

```
typedef struct {
     char nom[16];
     int naissance;
     long salaire;
} personne;
...
personne jean, anne, x;
...
```

- Syntaxe de `typedef` identique à celle de `static` et `extern`. Le nom de la variable devient un nom de type

# Remarques sur `struct`

- Le type d'un membre de structure est quelconque

```
typedef struct {
     char mois[9];
     int annee;
} date;
typedef struct {
     char nom[16];
     date naissance;
     long salaire;
} personne;
...
jean.naissance.annee = 1960;
...
```

- Un même identificateur peut être utilisé pour désigner le type d'une structure, son label et un membre, ou une variable, un label et un membre. Mais pas pour un type *et* une variable.

  Deux types de structures peuvent avoir un même nom de membre (car les espaces de noms sont disjoints).

# Le type union

Entité permettant d'interpréter la valeur d'un objet selon différents types

- Exemple : date de naissance ou âge

| | | | |
|---|---|---|---|
| mois | Juillet | | âge |
| annee | 1960 | | |

| | | | |
|---|---|---|---|
| mois | | 34 | âge |
| annee | | | |

- Description du type

```
typedef union {
     date naissance;
     int age;
} etatcivil;
```

- Initialisation possible pour le premier membre uniquement

```
etatcivil etat = {"Juillet", 1960};
```

# Remarques sur `union`

- Même syntaxe, mêmes opérations, mêmes remarques que pour les structures
- La taille d'une union est celle du plus long type de ses membres
  Par exemple, `sizeof(etatcivil)` vaut `sizeof(date)`
- On peut affecter une valeur à une union ou extraire une variable par n'importe lequel de ses sélecteurs (noms de membres).
  Mais c'est à l'utilisateur d'en gérer le type, au moyen d'une structure, par exemple

```
...
typedef struct {
   char nom[16];
   char cas; /* 'd' pour date, 'i' pour int */
   union {
      date naissance;
      int age;
   } etatcivil;
   long salaire;
} personne;
...
if (jean.cas == 'i')
   printf("AGE = %d\n", jean.etatcivil.age);
...
```

# Énumération

Facilité pour la déclaration de constantes entières (ou de caractères)

- Exemples

```
enum {DATE, INT} cas;
enum COUL {bleu, blanc, rouge} couleur;
typedef enum {faux, vrai} boolean;
...
boolean test;
...
couleur = bleu;
test = vrai;
...
```

- La première constante vaut `0`, le deuxième vaut `1`, ..., sauf si des valeurs sont données

```
enum escapes {BELL = '\a', BACKSPACE = '\b',
              TAB = '\t', NEWLINE = '\n',
              VTAB = '\v', RETURN = '\r'};
enum jours {LUN=1, MAR, MER, JEU, VEN, SAM, DIM};
```

# Remarques sur `enum`

- Les constantes d'une énumération sont déclarées implicitement de type `int`
- Les noms des constantes d'une énumération ne peuvent être des identificateurs de variables, ni de fonctions, ni de types
- Le label d'une énumération est facultatif
- Synonymes : trois formulations équivalentes

```
enum {bleu, blanc = 7, rouge} couleur;
```

```
#define bleu  0
#define blanc 7
#define rouge 8
int couleur;
```

```
const int bleu = 0;
const int blanc = 7;
const int rouge = 8;
int couleur;
```

# Le type tableau

Regroupement indicé d'objets d'un même type de base

- Exemple : table de sept flottants

| | |
|---|---|
| 0 | 12.70 |
| 1 | 8.34 |
| 2 | 1.00 |
| 3 | 40.05 |
| 4 | 34.70 |
| 5 | 9.70 |
| 6 | 55.50 |

- Déclaration

```
double table[7];
```

- Déclaration avec initialisation

```
double table[7] = {12.70, 8.34, 1.00,
                   40.05, 34.70, 9.70, 55.50};
```

- Le type de base est quelconque

```
personne liste[TAILLE];
enum {bleu, blanc, rouge} couleur[20];
```

- Définition de types

```
typedef double type_table[7];
typedef personne type_liste[TAILLE];
typedef enum {bleu, blanc, rouge} type_couleur[20];
```

- Accès aux éléments par l'indice : il n'y a **pas de contrôle de débordement sur les indices**

```
double x, table[taille];
int i;
...
x = table[5];
...
for(i=0; i<taille; i++)
   table[i] = ...
...
personne jean, liste[TAILLE];
...
liste[i].salaire = 200000;
jean.nom[0] = 'D';
...
```

# Représentation

- Les éléments d'un tableau sont rangés à des adresses consécutives dans la mémoire
- Un tableau `table` est assimilé à l'adresse de son premier élément `table[0]`
- Ceci est valide, mais n'est qu'une copie d'adresse

```
int t[100];
int * x; /* adresse sur un entier */
...
/* copie l'adresse de t[0] dans x */
x = t;
/* instruction similaire */
x = &t[0];
...
```

- **En conséquence**, il n'y a pas d'opération globale sur le type tableau hormis l'initialisation.
  En particulier, on ne peut tester directement l'égalité de deux tableaux (avec `==`),
  ni les copier avec l'affectation (`=`).

# Tableau en argument

- Tableau en paramètre

```
void initialisation(double t[], int n){
     int i;
     for(i=0; i<n; i++)
         t[i] = 3.14;
}
```

- Tableau en argument

```
int taille = 100;
double table[taille];
...
initialisation(table, taille);
...
```

- L'adresse de `table[0]` est transmise au moment de l'appel à la fonction `initialisation`.
  Il y a *transmission par valeur* de l'adresse.
  Ceci revient à une *transmission par adresse* du tableau.

- **En conséquence**, il n'y a pas de copie des éléments du tableau au moment de l'appel. La fonction travaille directement sur le tableau de la fonction qui appelle.

| RETENIR : |
|---|
| **transmission par valeur pour structures et unions** |
| **transmission par adresse pour les tableaux** |

# Tableau à plusieurs dimensions

- Matrice $2 \times 3$ :

| | | |
|---|---|---|
| 12.70 | 8.34 | 1.00 |
| 40.05 | 34.70 | 9.70 |

- Un tableau à $n$ dimensions est un tableau dont le type de base est un type de tableau à $n-1$ dimensions

| | | | |
|---|---|---|---|
| matrice[0] | 12.70 | 8.34 | 1.00 |
| matrice[1] | 40.05 | 34.70 | 9.70 |

- Déclaration avec initialisation

```
double matrice[2][3] = { {12.70, 8.34, 1.00},
                         {40.05, 34.70, 9.70} };
```

`matrice` est un tableau de taille 2 (deux lignes) dont chaque entrée est un tableau de taille 3 et dont les éléments sont des flottants "`double`"

- Accès aux éléments

```
...
x = matrice[0][2];
matrice[i][j] = x;
...
```

# Chaîne de caractères

- Une chaîne est un tableau de caractères terminé par le caractère `NUL` (`'\0'`, entier zéro)

| b | o | n | j | o | u | r | \0 | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Comme un tableau ordinaire

```
char car, chaine[100];
char salut[10] =
   {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
...
car = salut[3];
...
```

- Initialisation simplifiée

```
char salut[10] = "bonjour";
...
car = salut[3];
...
```

- La bibliothèque `<string.h>` contient des fonctions de manipulation de chaînes de caractères

# EXERCICES

1. **Strutures, E/S** : ajouter des membres (champs) à la structure qui décrit la fiche individuelle d'une personne. Écrire les fonctions de saisie et d'affichage d'une fiche :

   ```
   personne saisie(void);
   void affichage(personne x);
   ```

2. **Structures, échange** : programmer la fonction “`echange`” qui échange les valeurs de deux structures (de type “`personne`”, par exemple).

3. **Tableaux** : calculer la somme cumulée des entrées d'un tableau de réels.

4. écrire une fonction de copie de tableau (de type donné).

5. initialiser un tableau `table[2][3][2]` de telle façon que `table[i][j][k]` soit égal à `i+j+k`.

6. **Position** : écrire une fonction qui donne l'indice d'un élément $x$ donné (sa position) dans un tableau $t$ de $n$ éléments (et donne $-1$ si $x$ n'apparaît pas dans le tableau).

7. **Dichotomie** : même question avec un tableau dont les éléments sont en ordre croissant ($t[0] \leq t[1] \leq \ldots \leq t[n-1]$). On utilisera la méthode par dichotomie : on compare $x$ à l'élément central, puis on ramène la recherche à la première moitié ou à la seconde moitié du tableau.

# Structures : entrées/sorties

```
personne saisie(void){
   personne x;
   printf("Saisie d'une fiche\n");
   printf("Nom : "); scanf("%s", x.nom);
   printf("Date de naissance (0) ou age (1) ? ");
   scanf("%d", &x.cas);
   if(x.cas == DATE){
      printf("Mois de naissance : ");
      scanf("%s", x.etatcivil.naissance.mois);
      printf("Annee de naissance : ");
      scanf("%d", &x.etatcivil.naissance.annee);
   } else {
      printf("Age : "); scanf("%d", &x.etatcivil.age);
   }
   printf("Salaire : "); scanf("%ld", &x.salaire);
   return x;
}

void affichage(personne x){
   printf("%s, ",  x.nom);
   if(x.cas == DATE){
      printf("%s, ",  x.etatcivil.naissance.mois);
      printf("%d, ",  x.etatcivil.naissance.annee);
   } else {
      printf("%d ans, ", x.etatcivil.age);
   }
   printf("%ld\n", x.salaire);
}
```

# Structures : échange

```
typedef struct {
   char mois[10];
   int annee;
} date;

typedef struct {
   char nom[16];
   date_age cas;
   union {
      date naissance;
      int age;
   } etatcivil;
   long salaire;
} personne;

personne jean, anne;

void echange(personne * x, personne * y){
   personne temp;
   temp = *x; *x = *y; *y = temp;
}

...
   echange(&jean, &anne);
...
```

# Tableaux

```
void affiche_table(double t[], int n){
   int i;
   for(i=0; i<n; i++)
      printf("%g ", t[i]);
   printf("\n");
}

void saisie_table(double t[], int n){
   int i;
   for(i=0; i<n; i++){
      printf("t[%d] = ", i); scanf("%g", &t[i]);
   }
}

double somme(double t[], int n){
   double cumul = 0;
   int i;
   for(i=0; i<n; i++)
      cumul += t[i];
   return cumul;
}
```

```
void copie(double r[], double t[], int n){
   int i;
   for(i=0; i<n; i++)
      r[i] = t[i];
}

typedef int type_tab[2][3][2];
type_tab tab;

void init_tab(type_tab tab){
   int i, j, k;
   for(i=0; i<2; i++)
      for(j=0; j<3; j++)
         for(k=0; k<2; k++)
            tab[i][j][k] = i+j+k;
}

void aff_tab(type_tab tab){
   int i, j, k;
   for(i=0; i<2; i++)
      for(j=0; j<3; j++)
         for(k=0; k<2; k++)
            printf("%d ", tab[i][j][k]);
   printf("\n");
}
```

# Position

```
int position(long x, long t[], int n){
   int pos;
   for(pos=0; pos<n; pos++)
      if(x == t[pos])  return pos;
   return -1;
}
```

```
int position(long x, long t[], int n){
   int pos = 0;
   while(pos<n && x !=t[pos])
      pos++;
   if(pos == n) return -1;
   else return pos;
}
```

# Dichotomie

```
int dicho(long x, long t[], int debut, int fin){
   /* debut <= fin */
   if(debut == fin)
      if(x == t[debut]) return debut;
      else return -1;
   else {
      int milieu = (debut+fin)/2;
      if(x <= t[milieu])
         return dicho(x, t, debut, milieu);
      else
         return dicho(x, t, milieu+1, fin);
   }
}

int position(long x, long t[], int n){
   /* n > 0 */
   return dicho(x, t, 0, n-1);
}
```