Fiche n°9 : structures

Une **structure** regroupe plusieurs informations pouvant avoir des types différents ou, comme dans cet exemple, identiques.

```
struct date
{
    int jour;
    int mois;
    int annee;
};
```

Déclaration d'un type **struct date** incluant trois champs (ou attributs) nommé **jour, mois** et **annee.**

```
struct date d1 = {29,1,2012};
```

Déclaration et initialisation d'une **variable** de type **struct date.**

Déclaration d'une variable de type struct date et assignation de ses champs.

```
struct date d2;

d2.jour=12;
d2.mois=5;
d2.annee=2012;
```

Déclaration simplifiée avec typedef.

```
typedef struct date typedate;
```

Déclaration d'un identificateur **typedate** équivalent à **struct date,** puis déclaration d'une variable de type **typedate.**

```
typedate d = {29,1,2012};
```

Version condensée

```
typedef struct date
{
    int jour;
    int mois;
    int annee;
} typedate ;
```

Deux identificateurs : **struct date** et **typedate.**

```
typedef struct
{
    int jour;
    int mois;
    int annee;
} typedate ;
```

Un identificateur **typedate.**

Imbrication de structures

Les champs d'une structure peuvent être des structures, comme dans cet exemple simplifié de représentation d'une fiche comportant le titre d'un film et sa date de sortie.

```
typedef struct
{
  char titre[256];
  typedate dateSortie;
}film;
```

```
film f = {"ET",{1,12,1982}};
```

Accès au champs d'une structure à partir de son adresse mémoire

Si **p** désigne l'adresse d'une structure ayant un champ **x,** alors la notation **(*p).x** peut être simplifiée en **p->x.**

Attention, ici il faut utiliser **strcpy** et **pas ==.**

Pourquoi ?

Après p il y a une flèche car **p** est une adresse. Ensuite il y a un point car **p->dateSortie** n'est pas une adresse.

```
film f1; film* p;
```

```
strcpy(p->titre,"Matrix");
```

```
p->dateSortie.annee = 1999;
```

Fiche n°10 : Structures et fonctions

Une fonction peut accepter en paramètre une donnée de type structure.

```
void printDate(typedate d)
{
    printf("%d %d %d\n",d.jour,d.mois,d.annee);
}
```

Attention : la fonction reçoit une **copie** qui **ne permet pas la modification** de la structure originale

Une fonction peut aussi accepter en paramètre un pointeur sur une structure. L'accès se fait alors en lecture et/ou écriture.

```
void setDate(typedate* d, int j, int m, int a)
{
    d->jour=j; d->mois=m; d->annee=a;
}
```

Une fonction peut retourner une structure.

La valeur de retour peut être soit transmise à une fonction, soit assignée à une variable de type approprié.

```
typedate makeDate(int j, int m, int a)
{
    typedate d;
    d.jour=j; d.mois=m; d.annee=a;
    return d;
}
```

Une valeur de type structure peut être assignée à une variable du type approprié.

```
typedate d;
```

Ces deux appels ont exactement le même effet.

```
d = makeDate(1,12,1982);
```

```
setDate(&d,1,12,1982);
```

Attention : On peut assigner des structures avec l'opérateur =, mais on ne peux pas les comparer avec == ou avec !=.

Pour comparer des structures, il faut réaliser une fonction dédiée.

```
typedate d1 = makeDate(1,12,1982);
typedate d2 = makeDate(2,12,1982);
```

```
typedate d = d1;
```

Autorisé

```
if(d1==d1) …
```

Interdit

Fiche n°11 : Tableaux à deux dimensions

IL existe deux représentations des tableaux à deux dimensions en langage C. Nous n'en verrons qu'une seule, qui pourrait être qualifiée de "moderne" car elle est utilisée en langage Java.

Elle consiste à représenter un tableau 2D sous la forme d'un tableau de pointeurs sur des tableaux 1D.

Cela suppose de pouvoir créer dynamiquement des tableaux à une dimension. Pour ce faire, nous utiliserons la fonction **malloc**, qui permet de réserver un bloc de mémoire utilisable pour réaliser un tableau.

Adresse du bloc réservé

```
void* malloc(unsigned int size)
```

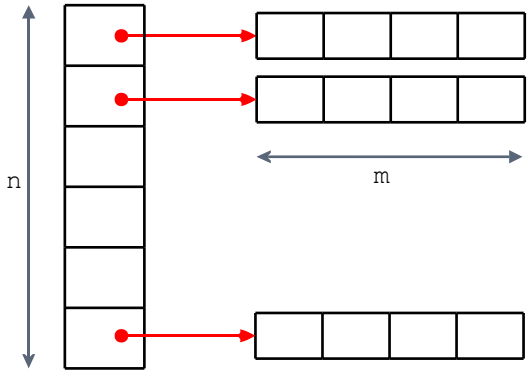
Taille en octets du bloc de mémoire à réserver.

Ce pointeur a un type particulier. Il pointe sur "n'importe quoi". Il doit faire l'objet d'une conversion explicite de type pour être utilisé.

Exemple de création d'un tableau 2D d'entiers

Pour créer un tableau de n par m, n+1 blocs mémoire sont réservés.

```
int** make2D(int n, int m)
{
    int** t=(int**)malloc(n*sizeof(int*));
    int i;
    for(i=0; i<n; i++)
    {
        t[i]=(int*)malloc(m*sizeof(int));
    }
    return t;
}
```



Un appel à **make2D** crée un tableau à deux dimensions de n lignes et m colonnes et retourne l'adresse de ce tableau sous la forme d'un pointeur sur un pointeur sur **int**.

```
int** tab = make2D(3,5);

tab[1][2]=10;
```

Pour bien comprendre cette approche, vous devez vous référer à la fiche introductive sur les tableaux à une dimension. En particulier, vous devez comprendre qu'un tableau à une dimension est identifié par un pointeur sur sa première cellule.

La technique présentée ici est généralisable à des tableaux de n'importe quels types, y compris par exemple des tableaux de structures. Elle est évidemment aussi utilisable pour créer des tableaux 1D. Plus tard, nous verrons comment modifier la taille des tableaux obtenus.

Fiche de préparation n°3

Définissez un type point représentant à point ayant deux coordonnées de type double, puis une variable représentant un tableau de 10 points.

Tableau

Type point

Réalisez une fonction qui compare deux dates définies avec le type tyepedate précédemment décrit.

La valeur de retour doit être négative si d1 est antérieure à d2, positive si d1 est postérieure à d2, et 0 sinon.

Essayer de faire une implantation aussi simple que possible.

Quelle modification faut il faire pour passer les deux dates par pointeurs ? Cela a t-il un intérêt pratique ?

```
int compDates(typedate d1, typedate d2)
{




}
```

Réalisez une fonction crée un tableau à deux dimensions de valeur de type double, avec n lignes et m colonnes, et qui initialise à 0.0 toutes les cellules de ce tableau.