TP 5 - LES STRUCTURES ET LES PILES

Année 2021/2022

1. Définition

Une **structure** est un regroupement de plusieurs objets, de **types différents** ou **non**. *Grosso modo*, une structure est finalement une boîte qui regroupe plein de données différentes.

1.1 Définition d'une structure

Une structure étant un regroupement d'objets, la première chose à réaliser est la description de celle-ci (techniquement, sa **définition**), c'est-à-dire préciser de quel(s) objet(s) cette dernière va se composer.

La syntaxe de toute définition est la suivante.

```
struct étiquette

{
 /* Objet(s) composant(s) la structure. */
};
```

Prenons un exemple concret : vous souhaitez demander à l'utilisateur deux mesures de temps sous la forme heure(s):minute(s):seconde(s) et lui donner la différence entre les deux en secondes. Vous pourriez utiliser six variables pour stocker ce que vous fournit l'utilisateur, toutefois cela reste assez lourd. À la place, nous pourrions représenter chaque mesure à l'aide d'une structure composée de trois objets : un pour les heures, un pour les minutes et un pour les secondes.

```
int main(void)
{
    int heures1 , heures2 ;
    int minutes1, minutes2 ;
    double secondes1, secondes2 ;


    return 0;
}
```

Ou bien :

```
struct temps {
    int heures;
    int minutes;
    double secondes;
};
```

Comme vous le voyez, nous avons donné un nom « **étiquette** » à notre structure : « temps ». Pour le reste, la composition de la structure est décrite à l'aide d'une suite de déclarations de variables. Ces différentes déclarations constituent les **membres** ou **champs** de la structure. Enfin, notez la présence d'un **point-virgule *obligatoire*** à la fin de la définition de la structure.

Exercice 1 : déclare deux structures date (exemple 11 novembre 2021) et heure (10:25:58).

1.2 Définition d'une variable de type structure

Une fois notre structure décrite, il ne nous reste plus qu'à créer une variable de ce type. Pour ce faire, la syntaxe est la suivante :

```
int main(void)
{
      struct étiquette identificateur;
```

La méthode est donc la même que pour définir n'importe quelle variable, si ce n'est que le type de la variable est précisé à l'aide du mot-clé **struct** et de l'étiquette de la structure. Avec notre exemple de la structure temps, cela donne ceci.

```
struct temps {
    int heures;
    int minutes;
    double secondes;
};


int main(void)
{
    struct temps t;

    return 0;
}
```

1.3 Initialisation

Comme pour n'importe quelle autre variable, il est possible d'initialiser une variable de type structure dès sa définition. Toutefois, à l'inverse des autres, l'initialisation s'effectue à l'aide d'une liste fournissant une valeur pour un ou plusieurs membres de la structure.

Initialisation séquentielle : L'initialisation séquentielle permet de spécifier une valeur pour un ou plusieurs membres de la structure en suivant l'ordre de la définition. Ainsi, l'exemple ci-dessous initialise le membre heures à 1, minutes à 45 et secondes à 30.560.

```
struct temps t = { 1, 45, 30.560 };
```

Initialisation sélective : L'initialisation séquentielle n'est toutefois pas toujours pratique, surtout si les champs que vous souhaitez initialiser sont par exemple au milieu d'une grande structure. Pour éviter ce problème, il est possible de recourir à une initialisation sélective en spécifiant explicitement le ou les champs à initialiser. L'exemple ci-dessous est identique au premier si ce n'est qu'il recourt à une initialisation sélective.

```
struct temps t = {.secondes = 30.560,.minutes =  45,.heures = 1 };
```

Notez qu'il n'est plus nécessaire de suivre l'ordre de la définition dans ce cas.

Accès à un membre : L'accès à un membre d'une structure se réalise à l'aide de la variable de type structure et de **l'opérateur . suivi** du nom du champ visé : `variable.membre`

Cette syntaxe peut être utilisée aussi bien pour obtenir la valeur d'un champ que pour en modifier le contenu. L'exemple suivant effectue donc la même action que l'initialisation présentée précédemment :

```
t.heures = 1;
t.minutes = 45;
t.secondes = 30.560;
```

Exercice 2 : Ecrire un programme C qui définit une structure ***point*** qui contiendra les deux coordonnées d'un point du plan (X et Y). Puis lit deux points et affiche la distance entre ces deux derniers.

Exercice 3 : Ecrire un programme C qui définit une structure ***etudiant*** où un étudiant est représenté par son nom, son prénom et une note. Lit ensuite une liste d'étudiants entrée par l'utilisateur et affiche les noms de tous les étudiants ayant une note supérieure ou égale à *10* sur *20*.

Exercice 4 : Ecrire un programme C, qui lit les noms complets des étudiants et leurs moyennes dans un tableau de structures. Puis actualise ces moyennes en ajoutant un bonus de:

- 1 point pour les étudiants ayant une note strictement inférieure à 10.
- 0.5 point pour les étudiants ayant une note entre 10 et 15 incluses.

N.B.: la structure doit avoir deux éléments: une chaîne de caractères et un réel.

Exercice 5 : Définition de nom de type

Définir un type **Date** pour des variables formées d'un numéro de jour, d'un nom de mois et d'un numéro d'année.
Ecrire des fonctions de lecture et d'écriture d'une variable de type Date. Dans un premier temps, on ne se préoccupera pas de la validité de la date entrée.

Exercice 6 : Définition de nom de type

Ecrire la déclaration d'un type Fiche permettant de mémoriser les informations sur un étudiant :

- son nom ;
- son prenom ;
- sa date de Naissance, de type Date ;
- sa formation, représentée par deux lettres ;
- s'il est redoublant ou non ;
- son groupe de TD, représenté par un entier ;
- ses notes, représentées par un tableau note d'au plus MAXNOTES rels;
- un entier nbnotes indiquant le nombre de notes valides dans le tableau note.

3.1 Ecrire les fonctions LireFiche et EcrireFiche de lecture et d'écriture d'une Fiche. Aucune note n'est entrée par la fonction LireFiche.
3.2 Ecrire une fonction AjouteNote qui reçoit une Fiche et ajoute une note, si cela est possible.
3.3 Ecrire une fonction Moyenne qui reçoit une Fiche et renvoie, si cela est possible, la moyenne des notes de l'étudiant.

Exercice 7 : Rationnel

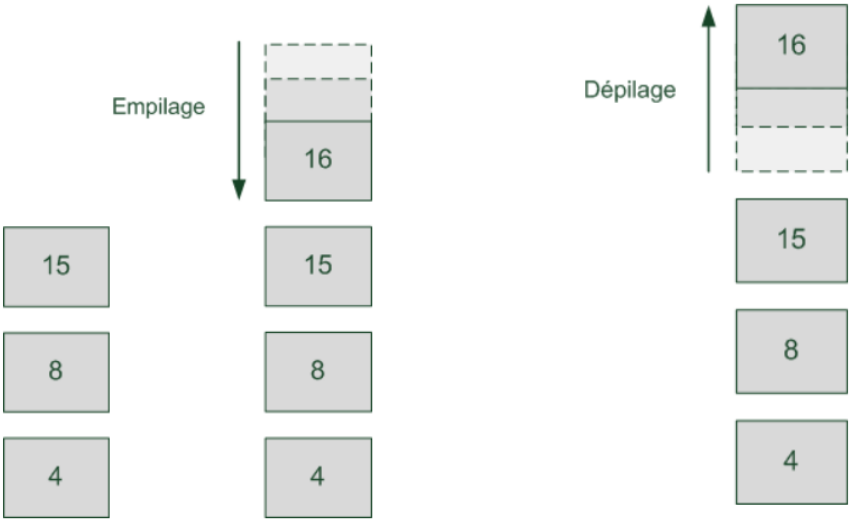
- Définir un type Rationnel composé de deux entiers: un numérateur et un dénominateur.
- Ecrire une fonction ***LireRationnel*** qui effectue la lecture d'un rationnel valide. Le rationnel mémorisé aura été simplifié.
- Ecrire une fonction ***SommeRationnel*** qui retourne la somme des deux rationnels valides passés en argument. Le rationnel retourné aura été simplifié.

Exercice 8 : Polynomes

- Définir un type Polynomes permettant de manipuler des polynomes à une variable, à coefficients réels de degré inférieur à DGMAX.
- Ecrire une fonction ***LirePolynome*** effectuant la saisie monome par monome d'un polynome. Pour chaque monome, on indiquera le degré puis le coefficient.

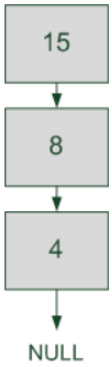
Les piles

Fonctionnement des piles : Le principe des piles en programmation est de **stocker des données** au fur et à mesure **les unes au-dessus des autres** pour pouvoir les récupérer plus tard.
Par exemple, imaginons une pile de nombres entiers de type int. Si j'ajoute un élément (on parle **d'empilage**), il sera placé au-dessus.



Le plus intéressant est sans conteste l'opération qui consiste à extraire les nombres de la pile. On parle **dedépilage**. On récupère les données une à une, en commençant par la **dernière** qui vient d'être posée tout en haut de la pile. On enlève les données au fur et à mesure, jusqu'à la dernière tout en bas de la pile. On dit que c'est un algorithme **LIFO**, ce qui signifie **« Last In First Out** ». Traduction : « Le dernier élément qui a été ajouté est le premier à sortir ».

Les éléments de la pile sont reliés entre eux à la manière **d'une liste chaînée**. Ils possèdent un **pointeur** vers l'élément suivant et ne sont donc pas forcément placés côte à côte en mémoire. Le dernier élément (tout en bas de la pile) doit pointer vers NULL pour indiquer qu'on a… touché le fond.



Création d'un système de pile : Maintenant que nous connaissons le principe de fonctionnement des piles, essayons d'en construire une. Comme pour les listes chaînées, il n'existe pas de système de pile intégré au langage C. Il faut donc le créer nous-mêmes. **Chaque élément de la pile aura une structure identique à celle d'une liste chaînée :**

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};
```

La structure de contrôle contiendra l'adresse du premier élément de la pile, celui qui se trouve tout en haut :

```
typedef struct Pile Pile;
struct Pile
{
    Element *premier;
};
```

Empilage : Notre fonction **empiler** doit prendre en paramètre la structure de contrôle de la pile (de type Pile) ainsi que le nouveau nombre à stocker. Je vous rappelle que nous stockons ici des **int**, mais rien ne vous empêche d'adapter ces exemples avec un autre type de données. On peut stocker n'importe quoi : des **double**, des **char**, des chaînes, des tableaux ou même d'autres structures !

```
void empiler(Pile *pile, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));
    if (pile == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = pile->premier;
    pile->premier = nouveau;
}
```

Dépilage : Le rôle de la fonction de dépilage est de supprimer l'élément tout en haut de la pile, ça, vous vous en doutiez. Mais elle doit aussi retourner l'élément qu'elle dépile, c'est-à-dire dans notre cas le nombre qui était stocké en haut de la pile. C'est comme cela que l'on accède aux éléments d'une pile : en les enlevant un à un. On ne parcourt pas la pile pour aller y chercher le second ou le troisième élément. On demande toujours à récupérer le premier. Notre fonction **depiler** va donc retourner un **int** correspondant au nombre qui se trouvait en tête de pile :

```
int depiler(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }

    int nombreDepile = 0;
    Element *elementDepile = pile->premier;
```

```
    if (pile != NULL && pile->premier != NULL)
    {
        nombreDepile = elementDepile->nombre;
        pile->premier = elementDepile->suivant;
        free(elementDepile);
    }

    return nombreDepile;
}
```

On récupère le nombre en tête de pile pour le renvoyer à la fin de la fonction. On modifie l'adresse du premier élément de la pile, puisque celui-ci change. Enfin, bien entendu, on supprime l'ancienne tête de pile grâce à *free*.

Affichage de la pile : Bien que cette fonction ne soit pas indispensable (les fonctions **empiler** et **depiler** suffisent à gérer une pile !), elle va nous être utile pour tester le fonctionnement de notre pile et surtout pour « visualiser » le résultat.

```
void afficherPile(Pile *pile)
{
    if (pile == NULL)
    {
        exit(EXIT_FAILURE);
    }
    Element *actuel = pile->premier;

    while (actuel != NULL)
    {
        printf("%d\n", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("\n");
}
```

Cette fonction étant ridiculement simple, elle ne nécessite aucune explication (et toc !). En revanche, c'est le moment de faire un **main** pour tester le comportement de notre pile :

```
int main()
{
    Pile *maPile = initialiser();

    empiler(maPile, 4);
    empiler(maPile, 8);
    empiler(maPile, 15);
    empiler(maPile, 16);
    empiler(maPile, 23);
    empiler(maPile, 42);

    printf("\nEtat de la pile :\n");
    afficherPile(maPile);
    printf("Je depile %d\n", depiler(maPile));
    printf("Je depile %d\n", depiler(maPile));
    printf("\nEtat de la pile :\n");
```

```
    afficherPile(maPile);

    return 0;
}
```

```
Etat de la pile :
42
23
16
15
8
4

Je depile 42
Je depile 23

Etat de la pile :
16
15
8
4
```