CM – Programmation C / C++

Chapitre 6 : Structures

```c
#include <stdio.h>
#include <conio.h>

struct employee {
    char name[100];
 int age;
 float salary;
 struct address {
        int houseNumber;
        char street[100];
    }location;
};

int main(){
   struct employee employee_one, *ptr;

   printf("Enter Name, Age, Salary of Employee\n");
   scanf("%s %d %f", &employee_one.name, &employee_one.age,
       &employee_one.salary);

   printf("Enter House Number and Street of Employee\n");
   scanf("%d %s", &employee_one.location.houseNumber,
       &employee_one.location.street);

   printf("Employee Details\n");
   printf(" Name : %s\n Age : %d\n Salary = %f\n House Number : %d\n Street : %s\n",
       employee_one.name, employee_one.age, employee_one.salary,
       employee_one.location.houseNumber, employee_one.location.street);

   getch();
   return 0;
}
```

I- Introduction :

Les structures servent à regrouper des éléments de types différents. Le mot clé pour déclarer une structure est `struct`. Elles sont souvent déclarées dans le header d'un projet.

```
struct Cylindre
{
    double Hauteur;
    double Rayon;
    string Couleur;
};
```

La structure possède un nom, ses éléments sont écrits dans un bloc en dessous et possèdent chacun un type et un nom. Le bloc se termine par un point-virgule `;`.

```
Cylindre C;
C.Hauteur=4.5;
C.Rayon=2.1;
C.Couleur="Rouge";
```

Créer une structure correspond à créer un nouveau type qui possède ses propres variables. On accède aux variables d'une structure en écrivant son nom (celui de la variable dont on a choisi comme type la structure) et celui de la variable séparés d'un point `.`.

II- Utilisation des structures :

Il est possible de créer une infinité de structures possédant autant de variables différentes que l'on souhaite. Cela permet de créer une multitude d'objets (et non de variables).

```
struct Vecteur
{
    double x;
    double y;
    double z;
};
```

Voici un nouvel exemple de structure qui permet de créer un vecteur (dans le sens mathématique, pas de programmation). Il est composé de trois variables réelles, donc il représente un vecteur à trois dimensions.

```
double produitScalaire(Vecteur v,Vecteur w)
{
    double m;
    m=v.x*w.x+v.y*w.y+v.z*w.z;

    return m;
}
```

Une structure peut servir comme paramètre d'une fonction, par exemple pour coder un produit scalaire entre deux vecteurs. Il est aussi possible d'avoir plusieurs objets (et non variables) d'un même type de structure.

Lorsqu'une structure est passée par copie comme paramètre d'une fonction, la consommation en ressources de l'ordinateur peut être grandement augmentée à force de passages. Afin d'éviter ce problème, il est judicieux de les passer par référence.

```
double produitScalaire(const Vecteur& v,const Vecteur& w)
{
    double m;
    m=v.x*w.x+v.y*w.y+v.z*w.z;

    return m;
}
```

En reprenant la fonction précédente, on a rajouté les symboles `&` devant chaque type des paramètres. Vu que les valeurs des paramètres ne vont pas varier après passage de la fonction, on utilise le mot clé `const` afin que les valeurs ne varient pas.

```
struct VecteurTab
{
    int Numéro;
    double Coordonnées[3];
};

struct TabDouble;
{
    int n;
    double *T;
};

struct Force
{
    VecteurTab Valeur;
    VecteurTab Point;
};
```

Une structure peut contenir des tableaux ou des pointeurs. Une structure peut même contenir une autre structure. Il est aussi possible de faire des tableaux d'objets, donc des tableaux de structures.

III- Surcharge d'opérateur :

Avec des objets de type primitifs (`int`, `double`, `float`, etc…), on peut utiliser les opérateurs arithmétiques et logiques définis par le langage de programmation. Dans le cas de l'opération `3.1+5.4`, l'opérateur est `+`, les opérandes sont `3.1` et `5.4`, et le résultat est `8.5`. On associe alors à cet opérateur la fonction : `double operator+(double gauche,double droite);` (avec `double gauche` l'opérande de gauche et `double droite` l'opérande de droite). Dans le cas de l'opération `3.1==5.4`, l'opérateur est `==`, les opérandes sont `3.1` et `5.4`, et le résultat est `false`. On associe alors à cet opérateur la fonction : `bool operator==(double gauche,double droite);`.

L'utilisation de structures ne permet pas de pouvoir utiliser les opérateurs arithmétiques et logiques directement avec, mais uniquement avec leurs variables internes. Afin de pouvoir faire des opérations avec les structures directement, il faut redéfinir les opérateurs que l'on veut utiliser avec ces nouveaux types. Pour cela, on utilise la syntaxe suivante pour n'importe quel opérateur `X` :

```
type operatorX(type gauche,type droite);
{
...
}
```

Cela correspond à une fonction dont le contenu sera la redéfinition de l'opérateur. On appelle cela la « surcharge d'opérateur ». Le sens des paramètres est important car il devra être le même lorsqu'on utilise l'opérateur.

On peut regarder ce que cela donne pour les surcharges de la multiplication `*` et des opérateurs de flux `<< >>`.

```
struct Vecteur
{
    double x;
    double y;
    double z;
};

double operator*(const Vecteur& gauche,const Vecteur& )
{
    return gauche.x*droite.x+gauche.y*droite.y+gauche.z*droite.z;
}

Vecteur operator*(double gauche,const Vecteur& droite)
{
    Vecteur res;
    res.x=gauche*droite.x;
    res.y=gauche*droite.y;
    res.z=gauche*droite.z;

    return res;
}
```

```
int main()
{
    Vecteur U,V;
    U.x=1.;
    U.y=2.;
    U.z=3.;
    V.x=1.;
    V.y=0.;
    V.z=0.;

    cout << "Produit scalaire : " << U*V << endl;

    Vecteur W=2.*U;
    cout << "W = {" << W.x << "," << W.y << "," << W.z << "}" << endl;

    return 0;
}
```

Ici, on crée deux fonctions permettant de redéfinir l'opérateur `*` dans deux cas distincts : le produit scalaire entre deux vecteurs (première fonction) et le produit d'un vecteur avec un réel (deuxième fonction). On peut surcharger un opérateur autant de fois que l'on souhaite, tant que les paramètres utilisés sont différents.

Dans le `main()`, on déclare deux vecteurs nommés `U` et `V` que l'on initialise pour chacune de leurs variables. On affiche le produit scalaire entre les deux en entrant l'opération `U*V`, donc c'est la première fonction qui est appelée (car on a deux vecteurs en opérandes). On déclare ensuite un troisième vecteur nommé `W` et qui est initialisé par l'opération `2.*W`, donc c'est la seconde fonction qui est appelée (car on a un réel et un vecteur en opérandes). On affiche ensuite les coordonnées de `W`.

```
struct Vecteur
{
    double x;
    double y;
    double z;
};

istream& operator>>(istream& flux,Vecteur& V)
{
    flux >> V.x >> V.y >> V.z;
    return flux;
}

ostream& operator<<(ostream& flux,const Vecteur& V)
{
    flux << "(" << V.x << "," << V.y << "," << V.z << ")";
    return flux;
}

int main()
{
    Vecteur V;

    cout << "Entrer un vecteur :" << endl;
    cin >> V;

    cout << "Voici le vecteur V : " << V << endl;

    return 0;
}
```

Ici, on crée deux fonctions permettant de redéfinir les opérateurs `<< >>` : la première permet de prendre en une fois les trois coordonnées du vecteur et la seconde permet d'afficher en une fois toutes ses coordonnées (dans ce cas, on ajoute un **`const`** car on ne veut pas modifier la variable). On remarque que les types **`istream`** et **`ostream`** sont les types génériques de flux. Dans le `main()`, on déclare un vecteur nommé `V` puis on demande à l'utilisateur d'entrer ses coordonnées. On écrit **`cin`** `>> V` donc c'est la première fonction qui est appelée (car on a un flux d'entrée et un vecteur en opérandes). On affiche ensuite les coordonnées de `V` en écrivant **`cout`** `<< V` donc c'est la seconde fonction qui est appelée (car on a un flux de sortie et un vecteur en opérandes).

IV- Exemple :

On souhaite écrire un programme permettant de saisir au clavier les caractéristiques d'un cylindre (sa hauteur, son rayon et sa couleur) et de trouver celui qui possède le plus grand volume entre plusieurs. On utilisera la structure de cylindre présentée dans l'introduction.

Le programme contiendra quatre fonctions :

- `Cylindre saisie()` pour saisir les caractéristiques d'un cylindre.
- `double volume(Cylindre C)` pour calculer le volume d'un cylindre.
- `void affiche(Cylindre C)` pour afficher les caractéristiques d'un cylindre.
- `Cylindre leplusgrand(vector<Cylindre> C)` pour renvoyer le cylindre ayant le plus grand volume parmi ceux du vecteur de cylindres.

Le programme comportera quatre fichiers :

- « Cylindre.h » qui contient la structure de cylindre.
- « Fonctions.h » qui contient le prototypage des fonctions énoncées juste avant.
- « Fonctions.cpp » qui contient les fonctions écrites.
- « main.cpp » qui contient le `main()` du programme.

```
#ifndef CYLINDRE_H_INCLUDED
#define CYLINDRE_H_INCLUDED

struct Cylindre
{
    double Hauteur;
    double Rayon;
    string Couleur;
};

#endif // CYLINDRE_H_INCLUDED
```

```
#ifndef FONCTIONS_H_INCLUDED
#define FONCTIONS_H_INCLUDED

Cylindre saisie();

double volume(Cylindre C);

void affiche(Cylindre C);

Cylindre leplusgrand(vector<Cylindre> C);

#endif // FONCTIONS_H_INCLUDED
```

Mettre la structure dans un header séparé assure qu'on ne la définit qu'une fois. Tous les prototypes sont regroupés dans un header séparé aussi.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "Cylindre.h"
#include "Fonctions.h"


Cylindre saisie()
{
    Cylindre C;

    do {
        cout << "Hauteur : ";
        cin >> C.Hauteur;
    } while (C.Hauteur<0);

    do {
        cout << "Rayon : ";
        cin >> C.Rayon;
    } while (C.Rayon<0);

    cout << "Couleur : ";
    cin >> C.Couleur;

    return C;
}
```

```
double volume(Cylindre C)
{
    return C.Hauteur*C.Rayon*C.Rayon*3.14159;
}

void affiche(Cylindre C)
{
    cout << "Cylindre (" << C.Hauteur << "," << C.Rayon << "," << C.Couleur << ")" << endl;
}

Cylindre leplusgrand(vector<Cylindre> C)
{
    int indice=0;
    int n=C.size();

    for(int i=0;i<n;i++)
        if(volume(C[i])>volume(C[indice]))
            indice=i;

    return C[indice];
}
```

Grâce à la première fonction, on saisie directement les variables de la structure et quand elle est retournée, ses trois variables le sont aussi. La seconde et troisième fonction sont simples. Pour la quatrième fonction, on utilise un vecteur de cylindres afin de regrouper tous les cylindres qui seront saisis, la structure est donc mise en paramètre. On renvoie un objet de type Cylindre à la fin de cette fonction.

```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "Cylindre.h"
#include "Fonctions.h"


int main()
{
    int n;

    do {
        cout << "Nombre de cylindres :";
        cin >> n;
    } while(n<1);

    vector<Cylindre> C(n);

    for(int i=0;i<n;i++)
        C[i]=saisie();

    Cylindre lpg=leplusgrand(C);

    cout << "Le plus grand cylindre est :" << endl;
    affiche(lpg);
    cout << "Son volume est :" << volume(lpg) << endl;

    return 0;
}
```

Dans le `main()`, on déclare une variable entière nommée `n` puis on crée une boucle **`do while`** afin d'obliger l'utilisateur à entrer une valeur supérieure ou égale à `1`. On déclare le vecteur de cylindres nommé `C` de taille `n` et, grâce à une boucle **`for`**, on remplit le vecteur avec `n` cylindres chacun saisis grâce à l'appel de la fonction `saisie()`. On déclare ensuite un nouveau cylindre nommé `lpg` et qui permet d'appeler la fonction `leplusgrand()` pour obtenir les variables du cylindre entré avec le plus grand volume. On affiche ensuite ce cylindre à l'utilisateur grâce à l'appel de la fonction `affiche()` entre autres.