Chapitre 11

Les types structurés

On appelle *structure de données complexe* une structure regroupant plusieurs données. Jusqu'à présent, la seule structure de données complexe que l'on a vu est le tableau, qui permet de stocker plusieurs valeurs de même type.

On peut parfois avoir besoin de regrouper sous un même nom des données de types différents afin de représenter plus directement les objets de notre problème. Un type structuré va nous permettre de définir de nouveaux types.

11.1 Intérêt des types structurés

Les types simples (`char`, `int`, `double`, etc.) sont suffisants pour des problèmes simples mais en règle générale, il est souvent utile de manipuler des types structurés.

Exemple 1. Pour écrire un programme qui manipule deux points dans un plan, si l'on utilise les types simples, la déclaration ressemble à :

```
int p1x;        /* l'abscisse du premier point */
int p2x;       /* l'abscisse du second point */
int p1y;       /* l'ordonnée du premier point */
int p2y;       /* l'ordonnée du second point */
```

Problèmes :

— La déclaration est **fastidieuse**,

— **Peu de lisibilité** : rien n'indique que `p1x` et `p1y` correspondent au premier point.

Solution : Avoir la possibilité de déclarer des variables de type **point** :

```
point p1;
point p2;
```

Mais le type point n'existe pas !

Exemple 2. Le problème est encore plus criant dans l'exemple suivant : On veut écrire un programme qui va travailler sur des étudiants (maximum 200). Chaque étudiant est défini par un *nom*, un *prénom* et une *date de naissance*. En utilisant les tableaux, on déclare alors :

```
string nomEt[200];            /* le tableau des noms des étudiants */
string prenomEtu[200];     /* le tableau des prénoms des étudiants */
int jjEt[200];      /* le tableau des jours de naissances des étudiants */
......
```

Le code est très peu lisible. De plus, les informations d'un étudiant se trouvent dans plusieurs tableaux, ce qui n'est pas pratique.

Solution : Avoir la possibilité de déclarer un tableau d'étudiants :

```
etudiant tabEtu[200];
```

Mais le type etudiant n'existe pas !

11.2 Les types structurés

11.2.1 Définition d'un type structuré

Une structure de données complexe est *le type structuré* (appelé aussi *enregistrement*). C'est une structure de données composée d'une juxtaposition de données quelconques appelées *champs* ou *membres*. Le mot clef `typedef` commence la déclaration d'un nouveau type.

En C++, un type structuré est déclaré par la forme générale :

```
    typedef struct { <ListeDeclarations> } nom;
```

Plusieurs éléments interviennent dans la définition d'un type structuré :

— Le nom `nom` est celui du type structuré et il va jouer un rôle de type pour déclarer des variables. On parlera alors de **variables structurées**.

— `<ListeDeclarations>` donne les déclarations des différents champs de la structure. Chaque champ peut être de n'importe quel type, c'est-à-dire de type `int`, `float`, `string`, etc. Mais le champ peut être lui aussi de type structuré ; on abordera cette question plus loin dans ce cours. Chaque champ a un type précis et un nom.

Attention : Il ne faut pas oublier le symbole `;` après la définition d'un type structuré.

L'intérêt des types structurés est de pouvoir par la suite déclarer des variables de type structuré. Le `typedef` qui sert à créer un alias de structure permet d'éviter d'avoir à réécrire une partie de la définition de la structure.

Remarque : Les types structurés se déclarent **en dehors du main()** avant les fonctions.

Attention : La définition d'un type structuré NE RÉSERVE PAS d'espace mémoire. Elle déclare un nouveau type de données.

Attention : Une fois un type structuré défini, on ne peut pas lui rajouter/retrancher un champs. La définition est statique.

11.2.2 Exemples de types structurés

Exemple 1. (suite) En ce qui concerne la représentation d'un *point du plan*, il faut :

1. une abscisse x de type `int`,
2. une ordonnnée y de type `int`.

On pourra définir le type structuré `point` comme suit :

```
typedef struct
{
  int x;
  int y;
} point;
```

Exemple 2. (suite)

1. Si l'on veut définir un type permettant de définir une *date*, il faut alors :
   (a) un *jour* de type `int`,
   (b) un *mois* de type chaîne de caractères `string`,
   (c) une *année* de type `int`.

   On définira le type `date` comme suit :

```
typedef struct
{
  int jour;    /* champs représentant le jour */
  string mois;  /* mois est une chaine représentant le mois */
  int annee; /* le champ année qui représente l'année */
} date;
```

2. Si l'on veut définir un type *étudiant* dont les caractéristiques sont les suivantes :
   (a) un *nom* de type chaîne de caractères,
   (b) un *prénom* de type chaîne de caractères,
   (c) une *date de naissance* de type **date**. Attention ! La **date** est un type structuré. Ce champ sera donc structuré et "encapsulera" le *jour*, le *mois* et l'*année* de naissance.

   On définira le type `etudiant` comme suit :

```
typedef struct
{
  string nom;
  string prenom;
  date datNaiss;              /* Ce champ est structuré et il se réfère
                                     au type struct date défini plus haut. */
} etudiant;
```

Les types `point`, `date` et `etudiant` deviennent ainsi des **nouveaux types**. Il sera possible de déclarer des variables ayant ces types.

11.2.3 Déclaration de variables de type structuré et initialisation

Pour déclarer une variable, la syntaxe reste celle utilisée habituellement :

```
    <type> <variable>;
```

Pour déclarer des variables de type structuré, on peut donc écrire :

```
point p1;
date d;
etudiant et1;
```

où :

— `p1` est une variable structurée composée de deux champs : `x` et `y`,

— `d` est une variable structurée composée de trois champs : `jour`, `mois` et `annee`.

— `et1` est une variable structurée composée de trois champs : `nom`, `prenom` et `date` (qui est lui-même un type structuré composé de trois champs : `jour`, `mois` et `annee`).

Comme pour les tableaux, les variables structurées peuvent être initialisées, lors de leur déclaration, avec des valeurs littérales. Ces valeurs sont données entre accolades séparées par des virgules. La première valeur correspond au premier champ, la deuxième au deuxième champ, etc.

Par exemple,

```
point p1 = {5, 4};
```

définit une variable `p1` de type `point`, où le champ `x` est initialisé à 5 et le champ `y` est initialisé à 4. `p1` correspond alors au point de coordonnées (5,4). S'il n'y a pas de valeur pour chacun des champs de la structure, les champs qui n'ont pas de valeur prennent la valeur 0 (ou `NULL` dans le cas des pointeurs) [1].

Attention : Lors de sa déclaration, une variable structurée n'a pas de valeur initiale par défaut.

11.2.4 Accès aux champs d'une variable structurée

Lorsque l'on a des variables de type structuré, il est possible d'accéder à un champ à l'aide du symbole `.` en écrivant `variable.nomChamp`. Le champ se manipule de la même manière qu'une variable correspondant à ce type. Ainsi, le code suivant permet d'afficher les valeurs des abscisse et ordonnée d'un point `p`.

```
point p = {5,4};
affichage("Abscisse du point p : ", p.x, '\n');
affichage("Ordonnée du point p : ", p.y , '\n');
```

De la même manière, il est possible de modifier les champs d'une date :

```
date d = {1,"janvier",1970};
d.jour = 31;
d.mois = "décembre"; //Permet de copier "décembre" dans le champ mois de la variable d
d.annee = 2016;
affichage("Vivement le", d.jour, " ", d.mois, " ", d.annee, " ! \n");
//Affiche Vivement le 31 décembre 2016 !
```

De plus, si un champ est de type structuré, on peut accéder aux champs de ce dernier en utilisant de nouveau le symbole point. Par exemple,

```
etudiant e = {"Jean", "Dupond"};
e.datNaiss.jour = 1;
e.datNaiss.mois ="janvier";
e.datNaiss.annee = 2000;
affichage("Informations de l'étudiant :", e.nom, " ", e.prenom, " né le",
e.datNaiss.jour, " ", e.datNaiss.mois, " ", e.datNaiss.annee, '\n')l;
//Affiche : Informations de l'étudiant : Jean Dupond né le 1 janvier 2000
```

11.2.5 Affectation de variables structurées

Il est possible d'affecter à une variable structurée une variable de même type. Dans ce cas, toutes les valeurs des champs (même pour les champs de type tableaux ou de type structuré) de la variable à droite du signe égal sont copiées dans les champs de la variable située à gauche. Ainsi, l'exécution du code :

```
point p1 = {12,20};
point p2;
p2 = p1;       /* Tous les champs de p1 sont affectés aux champs de p2 */
affichage("P2 : x =>", p2.x, ", y =>", p2.y, '\n');
```

affiche

```
P2 : x => 12, y => 20
```

▶ Sur ce thème : **Exercice 1 du TD 11**

1. De la même manière, s'il existe un champ de type tableau qui n'est pas initialisé, alors tous ses éléments sont dans ce cas initialisés à 0.

11.3 Les tableaux de type structuré

On peut déclarer des tableaux de type structuré selon :

```
<type structuré> <variable> [<dimension>] ...
```

Par exemple, pour avoir un tableau (variable `s1groupeB2`) constitué de 27 étudiants (type structuré `etudiant`), il faut alors déclarer :

```
etudiant s1groupeB2[27];
```

La dimension [27] suit donc le nom du tableau comme dans le cas de tableaux destinés à stocker des valeurs de types simples, par exemple `int x[100];`. On utilise cette variable comme tout autre tableau avec le *point* (`.`) pour chaque accéder aux champs de chaque élément du tableau.

```
int jour;
string s;
int i = 10;
etudiant s1groupeB2[27];
...
affichage("Nom du 5ème étudiant : ", s1groupeB2[4].nom);

jour = s1groupeB2[i].dateNaiss.jour; /* jour récupère le jour de naissance
                                        de l'étudiant d'indice i */
s = s1groupeB2[i].nom /* récupère le premier caractère du nom de
                                 l'étudiant d'indice i */
```

Attention : Lors de la déclaration d'un tableau, les éléments ne sont pas initialisés, même pour les tableaux de type structuré.

11.4 Types structurés en paramètre et retour de fonctions

Comme pour les types simples, il est possible de passer une variable de type structuré en paramètre d'une fonction ou comme retour d'une fonction.

11.4.1 Passage de types structurés par valeur

Considérons l'exemple suivant :

```
typedef struct
{
  int x;
  int y;
} point;

void afficherPoint (point p)
{
  affichage("Point de coordonnées ", p.x, ",", p.y);
}

int main ()
{
  point p1 = {10,50};
  afficherPoint(p1);

  return 0;
}
```

Toute modification de la variable structurée `p` dans la fonction `afficherPoint()` n'entraîne pas la modification de la variable `p1`. C'est ce que l'on appelle un **passage par valeur**.

11.4.2 Types structurés en retour de fonction

Il est possible de retourner une variable structurée dans une fonction.

```
typedef struct
{
  int x;
  int y;
} point;

point saisirPoint ()
{
  point pTemp; /* variable locale à la fonction */

  affichage("Donnez l'abscisse :\n");
  saisie(pTemp.x) ;
  affichage("Donnez l'ordonee :\n");
  saisie(pTemp.y) ;

  return pTemp;       // On retourne la variable structurée pTemps au programme appelant

}

int main ()
{
  point p1;
  p1 = saisirPoint();        /* p1 est affecté par bloc par la valeur retournée
                                par la fonction saisirPoint() */
  afficherPoint(p1);

  return 0;
}
```

Observations :

- `pTemp` est une variable locale à la fonction `saisirPoint()`, sa durée de vie et sa visibilité sont limitées à cette fonction,
- c'est au moment de l'instruction `return pTemp;` que les valeurs des champs de `pTemp` sont renvoyées au programme appelant,
- c'est du ressort du programme appelant de récupérer la valeur retournée, par affectation dans notre exemple.

11.4.3 Passage de types structurés par référence

Si l'on souhaite modifier une variable de type structuré passée en paramètre d'une fonction, il faut, comme dans le cas des types simples, passer cette variable par référence. Le paramètre est donc modifié. Voici un exemple :

```
typedef struct
{
  int x;
  int y;
} point;

```

```
7  /* p est un pointeur sur le type structuré struct point.
8     dx et dy sont transmis par valeur */
9  void deplacerPoint (point &p, int dx, int dy)
10 {
11   p.x += dx;
12   p.y += dy;
13 }
14
15 int main ()
16 {
17   point p1 = {3,8};
18   int deplX = 4;
19   int deplY = 9;
20
21   deplacerPoint(p1, deplX, deplY);
22
23   return 0;
24 }
```

▶ Sur ce thème : **Exercices 2 et 3 du TD 11**

TD11 : Les types structurés (Corrigé)

✓ Exercice 1 : Structures simples*

1. Définir une structure de livre contenant :
   — un titre
   — un auteur
   — une année de parution
2. Créer et initialiser une variable correspondant à un livre.
3. Afficher les informations de ce livre.
4. Modifier la date de parution de ce livre.
5. Afficher les informations de ce livre.

Correction :

```
typedef struct
{
  string titre;
  string auteur;
  int date;
} livre;



int main ()
{
  livre liv1={"A quelques secondes près", "Harlan Coben", 2012};
  livre liv2={"Vendredi ou la vie sauvage", "Michel Tournier", 1971};

  affichage(liv1.titre, liv1.auteur, liv1.date, '\n');
  affichage(liv2.titre, liv2.auteur, liv2.date, '\n');

  liv2.date=2013;

  affichage(liv2.titre,liv2.auteur, liv2.date, '\n');

  return 0;
}
```

✓ Exercice 2 : Structures d'étudiants**

1. Écrire une fonction `saisirDate()` qui demande la saisie au clavier d'une date et la retourne à l'appelant.
2. Écrire une fonction `saisirEtudiant()` qui demande la saisie au clavier d'un étudiant et la retourne à l'appelant.
3. Écrire une fonction `afficherEtudiant()` qui permet d'afficher un étudiant.
4. Écrire un programme principal qui déclare un tableau de 5 étudiants, le remplit "au clavier" et l'affiche.

Correction :

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

typedef struct
{
  string nom;
  string prenom;
  date datNaiss;
} etudiant;

date saisieDate()
{
  date d;
  affichage("Saisir le jour : \n");
  saisie(d.jour);
  affichage("Saisir le mois : \n");
  saisie(d.mois);
  affichage("Saisir l'annee : \n");
  saisie(d.annee);
  return d;
}

etudiant saisieEtudiant()
{
  etudiant e;
  affichage("Nom : \n");
  saisie(e.nom);
  affichage("prénom : \n");
  saisie(e.prenom);
  e.datNaiss = saisieDate();
  return e;
}


void affichageEtudiant(etudiant e)
{
  affichage(e.prenom, " ", e.nom, " est né le ", e.datNaiss.jour, " / ",
          e.datNaiss.mois, " / ", e.datNaiss.annee);
}

int main ()
{
  etudiant classe[5];
  int i =0;

  while (i < 5)
  {
    classe[i] = saisieEtudiant();
    i++;
  }
  affichage("Affichage des étudiants :\n");

  i=0;
```

```
58
59    while (i < 5)
60    {
61      affichageEtudiant(classe[i]);
62      i++;
63    }
64
65    return 0;
66  }
```

Exercice 3 : Création d'une pile de taille maximale fixe**

En informatique, une pile (en anglais stack) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (ou LIFO pour Last In, First Out), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés. Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée [2].

Dans cet exercice, les éléments empilés seront des entiers (`int`). La pile pourra contenir au plus 100 entiers. Les fonctions possibles pour la pile seront :

- `initialisePile()` qui initialise la pile ou la vide si cette dernière contenait quelque chose,
- `estVide()` qui renvoie `true` si la pile est vide, `false` sinon,
- `estPleine()` qui renvoie `true` si la pile est pleine (contient 100 entiers), `false` sinon,
- `empile()` qui ajoute un entier en haut de la pile si cette dernière n'est pas pleine,
- `depile()` qui supprime dans la pile le dernier entier et retourne sa valeur (si la pile n'est pas vide).

Question 3.1 : Définir une structure pile permettant d'empiler des entiers.
Correction :

```
1  //Pile d'entiers pouvant contenir au max 100 entiers
2  typedef struct
3  {
4    int nbE; //Nombre d'éléments dans la pile
5    int tab[100];//Tableau représentant la pile
6  } pile;
```

Question 3.2 : Implémenter les différentes fonctions données précédemment (les types retournés et les paramètres ne sont pas donnés).
Correction :

```
1  //Initialise une pile ("vide" la pile si elle contenait quelque chose)
2  void initialisePile(pile &p)
3  {
4    p.nbE = 0;
5  }
6
7  //Renvoie true si la pile est vide, false sinon
8  bool estVide(pile p)
9  {
10   return p.nbE == 0;
11 }
12
```

2. Définition donnée par Wikipédia

```
//Renvoie true si la pile est pleine, false sinon
bool estPleine(pile p)
{
  return p.nbE == 100;
}

//Empile un entier dans la pile si celle-ci n'est pas pleine
void empile(pile &p, int c)
{
  if (!estPleine(p))
    p.tab[p.nbE] = c;
    p.nbE++;
}

//Dépile un entier dans la pile et le renvoie si celle-ci n'est
//pas vide. Dans le cas contraire, retourne 0
int depile(pile &p)
{
  if (estVide(p))
    return 0;
  else
  {
        p.nbE--;
        return p.tab[p.nbE];
  }
}
```

Question 3.3 : Écrire une fonction `affichePile()` qui affiche le contenu de la pile.
Correction :

```
//Affiche le contenu de la pile
void affichePile(pile p)
{
  affichage("Affichage de la pile :\n");
  int i=0;

  while (i < p.nbE)
  {
    affichage(p.tab[i], '\n');
    i++;
  }
}
```

TP11 : Les types structurés (Corrigé)

Exercice 4 : Les planètes d'un système planétaire*

Les exercices suivants ont pour but la conception d'une application relative au traitement des données d'un système planétaire. On caractérise chaque planète d'un système planétaire par :

— Son *nom*,

— sa *densité* (`double`),

— la *distance moyenne* de l'étoile autour de laquelle elle gravite (`double`) en 10E6 km,

— son *nombre de satellites* (`int`).

Définir un type structuré permettant de représenter une planète.
Correction :

```
typedef struct
{
  string nom ;
  double densite ;
  double distance ;
  int nbSatellites ;
} planete;
```

◇

Exercice 5 : Compréhension des variables structurées*

Déclarer une planète *p1*, demander à l'utilisateur de remplir les différents champs de cette planète, et afficher les informations de cette planète.
Correction :

```
int main()
{
  planete p1;

  affichage(" nom ? \n");
  saisie(p1.nom);

  affichage(" densite \n");
  saisie(p1.densite);

  affichage(" distance (10E6 km) ? \n");
  saisie(p.distance) ;

  affichage("satellites ? \n");
  saisie(p1.nbSatellites) ;

  affichage(" nom ", p1.nom, '\n');
  affichage(" densite :", p1.densite, '\n');
  affichage(" distance en 10E6 km :", p1.distance, '\n');
  affichage(" satellites : ", p1.nbSatellites, '\n') ;

  return 0;
}
```

◇

Exercice 6 : Comprendre l'initialisation à la déclaration*

1. Déclarer et initialiser à la déclaration les planètes suivantes :

nom	densité	distance moyenne	nombre de satellites
Mercure	5.42	58	0
Venus	5.25	108.2	0
Terre	5.52	149.6	1

2. Afficher ces planètes.

Correction :

```
planete p1={"Mercure", 5.42,58,0};
planete p2={"Venus", 5.25,108.2,0};
planete p3={"Terre", 5.52,149.6,1};

affichage(" nom          : ", p1.nom, '\n');
affichage(" densite  :", p1.densit, '\n');
affichage(" distance en 10E6 km : ", p1.distance, '\n');
affichage(" satellites : ", p1.nbSatellites, '\n');

affichage(" nom   : ", p2.nom, '\n');
affichage(" densite : ", p2.densite, '\n');
affichage(" distance en 10E6 km : ", p2.distance, '\n');
affichage(" satellites :", p2.nbSatellites, '\n');
%%%%
affichage(" nom : ", p3.nom,'\n');
affichage(" densite :", p3.densite, '\n');
affichage(" distance en 10E6 km : ",p3.distance, '\n');
affichage(" satellites : ", p3.nbSatellites, '\n');
```

◇

Exercice 7 : Comprendre l'affectation par bloc*

Déclarer trois variables structurées de type `planete` et leur affecter les planètes précédentes. Affichez toutes les planètes.

Correction :

```
planete p1={"Mercure", 5.42,58,0};
planete p2={"Venus", 5.25,108.2,0};
planete p3={"Terre", 5.52,149.6,1};

planete p4=p1;
planete p5=p2;
planete p6=p3;

affichage(" nom : ", p4.nom, '\n');
affichage(" densite : ", p4.densite, '\n');
affichage(" distance en 10E6 km : ",p4.distance, '\n');
affichage(" satellites :", p4.nbSatellites, '\n');

//Idem avec p5 et p6
```

◇

Exercice 8 : Comprendre les paramètres et les valeurs retournées de types structurés**

Le nombre de lignes de code augmente pour chaque saisie et chaque affichage de planète.Il est alors approprié d'écrire des fonctions permettant de structurer cette application :

1. Écrire une fonction `planete creerPlanete()` qui invite à la saisie d'une planète au clavier et la retourne (par l'utilisation de `return`).

Correction :

```
planete creerPlanete()
{
  planete p ;

  affichage(" nom ? \n");
  saisie(p.nom) ;

  affichage(" densite ? \n");
  saisie(p.densite) ;

  affichage(" distance en 10E6 km ? \n");
  saisie(p.distance) ;

  affichage(" satellites ? \n") ;
  saisie(p.nbSatellites) ;

  return p ;
}
```

◇

2. Écrire une fonction `void afficherPlanete(planete p)`, qui affiche **clairement** les caractéristiques de la planète reçue en paramètre.

Correction :

```
void afficherPlanete(planete p)
{
  affichage(" nom : ", p.nom, '\n);
  affichage(" densite : ", p.densite, '\n');
  affichage(" distance : ", p.distance, "10E6 km\n");
  affichage(" satellites : ", p.nbSatellites, '\n') ;
}
```

◇

3. Écrire une fonction, `bool egales(planete p1, planete p2)` qui compare les caractéristiques des planètes `p1` et `p2`. La fonction retourne *false* si les planètes ont des caractéristiques différentes, et *true* si les deux planètes ont les mêmes caractéristiques.

Correction :

```
bool egales(planete p1, planete p2)
{
  bool test;
  test = (p1.nom == p2.nom) && (p1.densite == p2.densite) &&
         (p1.distance == p2.distance)&& (p1.nbSatellites == p2.nbSatellites);
  return test ;
}
```

◇

4. Tester les fonctions précédemment écrites en utilisant la fonction principale suivante :

```
int main()
{
  planete p1,p2;
  p1 = creerPlanete();
  p2 = creerPlanete();

  afficherPlanete(p1);
  afficherPlanete(p2);

  if (egales(p1,p2))
    affichage("p1 a les mêmes caractéristiques que p2?");
  else
    affichage("p1 n'a pas les même caractéristiques que p2");
  return 0;
}
```

Exercice 9 : Modification des caractéristiques d'une planète**

1. Écrire une fonction `void modifiePlanete(...)` qui modifie les caractéristiques d'une planète donnée :
   — sa densité doit être multipliée par un facteur *dens*,
   — on a découvert *nSat* supplémentaires,
   — *dens* et *nSat* feront partie des paramètres.
2. Modifier la planète Mercure en multipliant sa densité par 1.2 et en lui rajoutant 3 satellites.
3. Afficher la planète Mercure pour vérifier que les modifications ont bien été faites.

Correction :

```
void modifierPlanete(planete &p, double nd, int nSat)
{
  p.densite *= nd;
  p.nbSatellites += nSat;
}
```

◇

Exercice 10 : Fonctions permettant de traiter un système de planètes**

1. Écrire une fonction `...afficherSysteme(...)` qui affiche les `n` planètes contenues dans un tableau de planètes.

Correction :

```
void afficherSysteme(planete systeme[], int n)
{
  int i ;
  while (i < n )
  {
    afficherPlanete(systeme[i]);
    i++;
  }
}
```

◇

2. Écrire une fonction `...initSysteme(...)` qui initialise un système planétaire comportant `n` planètes.

Correction :

```
void initSysteme(planete systeme[], int n)
{
  int i =0 ;
  while (i < n)
  {
    systeme[i] = creerPlanete() ;
    i++;
  }
}
```

◇

3. Écrire une fonction `... nbMoy(..)` qui calcule le nombre moyen de satellites par planète ainsi que la densité moyenne des planètes d'un système de `n` planètes. Les valeurs calculées seront affectées à deux paramètres passés par référence.

Correction :

```
void nbMoyen(planete systeme[],int n, double &pnb, double &pdens)
{
  int i =0 ;
  int nombreSatellites = 0 ;
  double sommeDensite = 0;

  while (i < n)
  {
    nombreSatellites += systeme[i].nbSatellites ;
    sommeDensite += systeme[i].densite ;
    i++;
  }
  pnb = (double) nombreSatellites/n;
  pdens = sommeDensite/n;
}
```

◇

4. Écrire une fonction `... modifierSysteme(..)` qui modifie toutes les planètes du système planétaire de `n` planètes en multipliant leur densité par un facteur `dens` et rajoutant un nombre supplémentaire de satellite(s) découvert(s) `nPSup` ; ces données seront reçues en paramètres.

Correction :

```
void modifierSysteme(planete systeme[], int n, double facDensite, int nSat)
{
  int i=0;
  while (i < n)
  {
    modifierPlanete(systeme[i],facDensite,nSat);
    i++;
  }
}
```

◇

5. Écrire un programme principal qui :
   (a) déclare un système planétaire de 4 planètes,
   (b) initialise au clavier ce système,
   (c) affiche ce système,
   (d) calcule et affiche le nombre moyen de satellite(s) des planètes du système planétaire,
   (e) multiplie la densité de toutes les planètes par un coefficient demandé à l'utilisateur et rajoute un nombre de satellite(s) décidé également par l'utilisateur,
   (f) affiche le système modifié.

Correction :

```
int main ()
{
  planete systeme[4];
  double satM, densM;
  initSysteme(systeme,4);
  afficherSysteme(systeme,4);
  nbMoyen(systeme,4,satM,densM);
  affichage("Nombre moyen de satellites ",satM, '\n');
  affichage("Densite moyenne : ", densM, '\n');
  modifierSysteme(systeme,4,4.f,3);
  afficherSysteme(systeme,4);

  return 0;

}
```

Annexe : Caractéristiques de quelques planètes

nom	densité	distance moyenne	nombre de satellites
Mercure	5.42	58	0
Venus	5.25	108.2	0
Terre	5.52	149.6	1
Mars	3.94	227.9	2
Jupiter	1.314	778.3	16
Saturne	0.69	1427	17
Uranus	1.19	2869	15
Neptune	1.6	4496	2
Pluton	1.8	5900	1