# Programmation C++ (débutant)/Les structures

## Le cours sur les structures

### Présentation

Nous avons utilisé pour l'instant des types de base : ils contiennent des éléments de type `int`, `char`, `double` ou `bool`. Dans ce chapitre nous allons définir de nouveaux types qui vont permettre de représenter des entités plus complexes. Par exemple, on pourra définir un type `point` qui contiendra une abscisse et une ordonnée. Nous allons donc apprendre à créer et à utiliser des types structurés.
Nous verrons ensuite comment, à partir de ces types, nous pouvons créer des fonctions permettant de manipuler ces nouveaux types. L'association de ces types et de ces fonctions sera appelée module, chacun sera défini par un fichier d'en-tête `.h` et un fichier source `.cpp`. Nous verrons à cette occasion la compilation séparée. Notre programme ne sera plus constitué d'un seul fichier mais d'un ensemble de fichiers qui pourra éventuellement être très important. Nous étudierons finalement un type de structure très usuel : les listes. Dans ce chapitre, notre représentation des listes sera très sommaire mais sera l'occasion de découvrir une méthodologie de programmation en découpant notre programme en différents modules qui seront dans une certaine mesure indépendants. Cette méthodologie nous permettra de concevoir des programmes de taille plus importante.

### Idée

En général, pour représenter en C++ des données, il faut plusieurs variables.
Exemple :
- pour représenter un point, il faut une abscisse et une ordonnée.
- pour représenter une fraction, il faut un numérateur et un dénominateur .
- pour représenter une liste de notes, il faut un nombre de notes et un tableau d'entiers ...

### Les structures

On va créer de nouveaux types qui regroupent plusieurs variables. D'apparence anodine, cette idée est extrêmement puissante.
**Exemple :**

```
struct point
{
    double x,y;
};
```

### Variable de type structuré

On peut maintenant définir des variables de type `point` .
Par exemple :

```
point a , b;
```

`a` contient 2 réels appelés `a.x` et `a.y`.
L'abscisse du point `a` sera notée `a.x`.
`a.x` est de type `double`.
L'ordonnée du point `b` sera notée `b.y`, de type `double` également.
`a.x` et `a.y` peuvent être utilisés pour des affectations, des entrées-sorties, comme toute variable de type `double`.

### Des données de plus en plus complexes

En fait, en définissant de nouveaux types, on crée des types de données plus complexes qui pourront eux-mêmes donner naissance à des type de données encore plus complexes et ainsi de suite. Cette méthodologie permettra de concevoir des types de données extrêmement élaborés (un texte structuré comme dans un traitement de texte par exemple). Le programmeur finira par en oublier les différents champs les composant en ne retenant que les manipulations qu'il est capable d'effectuer sur ces données.

### Exemple 1 : le type point

```
#include<iostream>
using namespace std;

struct point
{
    double x,y;
};

int main()
{
    point a,b,c;

    a.x=3.2;
    a.y=6.4;

    cout << "Tapez l'abscisse de b : ";
    cin >> b.x;

    cout << "Tapez l'ordonnée de b : ";
    cin >> b.y;

    c.x = (b.x + a.x) / 2;
    c.y = (b.y + a.y) / 2;

    cout << "Abscisse de c : " << c.x << endl;
    cout << "Ordonnée de c : " << c.y << endl;

    return 0;
}
```

- **Explications**
- Dans cet exemple, nous définissons une structure point composée de 2 réels x et y.
- Dans la fonction main, on définit 3 points a,b et c.
- On fixe les coordonnées du point a à (3.2,6.4).
- On demande à l'utilisateur de saisir les coordonnées du point b.
- On calcule dans c les coordonnées du milieu du segment [ab].
- On affiche finalement les coordonnées du point c.
- **Exécution**

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :
Tapez l'abscisse de b : **5.8**
Tapez l'ordonnée de b : **3.6**
Abscisse de c : 4.5
Ordonnée de c : 5.0

### Structuration d'un programme

- On va définir des structures.
- On va ensuite écrire des fonctions permettant de manipuler ces structures.
- On peut ensuite définir de nouvelles structures plus complexes à partir des structures de base et ainsi de suite.
- Cette approche de la programmation par les structures de données est une véritable méthodologie de structuration du programme.

### Fonctions manipulant des points

Sur le type point, on peut envisager d'effectuer les opérations suivantes :

- saisir les coordonnées d'un point au clavier.
- afficher les coordonnées d'un point.
- calculer la distance entre 2 points.
- calculer le milieu de 2 points.
- Nous allons donc créer 4 fonctions nous permettant d'effectuer des manipulations sur les points.

### Exemple 2 : manipulation d'un point

```
#include<iostream>
using namespace std;
#include<cmath>

struct point
{
    double x,y;
};

void saisir_point(point &p)
{
    cout << "Tapez l'abscisse du point : "; cin >> p.x;
    cout << "Tapez l'ordonnée du point : "; cin >> p.y;
}

void afficher_point(point p)
{
    cout << "Abscisse du point : " << p.x << endl;
    cout << "Ordonnée du point : " << p.y << endl;
}

double distance(point a, point b)
{
    double dx,dy;
```

```
    dx = a.x - b.x;
    dy = a.y - b.y;
    return sqrt( dx*dx + dy*dy );
}

void milieu(point a, point b, point &m)
{
    m.x = (a.x + b.x) /2;
    m.y = (a.y + b.y) /2;
}

int main()
{
    point X,Y,Z;
    double d;

    cout << "SAISIE DE X" << endl;
    saisir_point(X);

    cout << "SAISIE DE Y" << endl;
    saisir_point(Y);

    d=distance(X,Y);
    cout << "La distance de X à Y est : " << d << endl;

    milieu(X,Y,Z);
    cout << "AFFICHAGE DU POINT Z" << endl;
    afficher_point(Z);

    return 0;
}
```

- **Explications**
  - Dans cet exemple, on définit une structure point.
  - On définit une fonction saisir_point(&p) qui a en paramètre un point passé par référence : en effet, cette fonction doit modifier la valeur du point p et y mettre les valeurs saisies par l'utilisateur. Il faut donc passer p par référence. p est un paramètre en sortie de cette fonction.
  - La fonction afficher_point(point p). Cette fonction affiche les coordonnées du point p. p est un paramètre en entrée de cette fonction et il est inutile de le passer par référence.
  - La fonction double distance(point a, point b) permet de calculer la distance entre les point a et b qui sont des paramètres en entrée de cette fonction. La distance est renvoyée par un return.
  - La fonction void milieu(point a, point b, point &m) calcule dans le point m le milieu du segment [ab]. a et b sont des paramètres en entrée de cette fonction. m est un paramètre en sortie qui est passé par référence.
  - La fonction main déclare 3 point X,Y et Z. On demande ensuite à l'utilisateur de taper les coordonnées de X et Y. On calcule en appelant la fonction distance la distance entre X et Y. On calcule dans le point Z le milieu de [XY]. On affiche finalement la distance XY et les coordonnées de Z.
- **Exécution**

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :
SAISIE DE X :
Tapez l'abscisse du point : **3**
Tapez l'ordonnée de a : **4**
SAISIE DE Y
Tapez l'abscisse du point : **2**
Tapez l'ordonnée de a : **5**
La distance de X à Y est : 1.414
AFFICHAGE DU POINT Z
Abscisse du point : 2.5
Ordonnée du point : 4.5

## Décomposition d'un programme en plusieurs fichiers

Nos programmes vont désormais être découpés en plusieurs fichiers : certains seront des fichiers d'en-tête et porteront l'extension `.h`. D'autres comporteront du code et porteront l'extension `.cpp`.
Chaque programme sera constitué de modules : chaque module sera défini par un fichier `.h` et un fichier `.cpp`. Chaque module aura une fonctionnalité bien définie. Il comportera en général la définition d'une structure et des fonctions permettant de manipuler ces structures.
Cette approche nous permet d'améliorer petit à petit notre méthodologie de programmation.

## Réécriture de l'exemple 2

- Nous allons réécrire l'exemple 2 en restructurant notre code selon cette approche modulaire.
- Nous allons définir un module point qui regroupera la définition d'un point et différentes manipulations sur les points.
- Ce module sera réutilisable dans d'autres contextes.

## Décomposition en modules

- On va créer un fichier point.h où on va regrouper :
  - la structure point.
  - les entêtes des fonctions manipulant point.
- On va écrire un fichier point.cpp comportant :
  - `#include "point.h"` .
  - les fonctions manipulant des points.

On a ainsi créé un module réutilisable permettant de manipuler des points. Lorsque le programme principal voudra utiliser notre module point, il faudra rajouter `#include <point.h>`.
Sur des petits programmes cette structuration semble être une perte de temps. En fait cela permet de gagner du temps de débogage et cette méthodologie devient indispensable dès que l'application devient conséquente.

### Le programme principal

Notre programme est désormais constitué de 3 fichiers :

- Les fichiers `point.h` et `point.cpp` qui définissent un module point.
- Le fichier main.cpp qui contient la fonction main. Ce fichier utilise le module point et commence donc par `#include "point.h"`.

### Les fichiers d'en-tête

Un programme peut être constitué d'un grand nombre de modules : par exemple imaginons qu'un programme soit constitué de 4 modules A, B, C et D.
On suppose que B et C utilisent les notions définies dans A : ils commencent donc tous les deux par `#include"A.h"`.
On suppose que le module D utilise les notions définies dans B et dans C. D commence donc par `#include"B.h"` et `#include"C.h"`.
Le module A sera donc défini 2 fois dans le module D une fois lors de l'inclusion de `B.h` et une deuxième fois lors de l'inclusion de `C.h`. Le compilateur refusera de compiler notre programme !

### Exemple 3 : programmation modulaire

**Fichier point.h**

```
#ifndef POINT_H
#define POINT_H

struct point
{
    double x,y;
};

void saisir_point(point &p);
void afficher_point(point p);
double distance(point a, point b);
void milieu(point a, point b, point &m);

#endif
```

**Fichier point.cpp**

```
#include "point.h"
#include <iostream>
#include <cmath>

using namespace std;

void saisir_point(point &p)
{
    cout << "Tapez l'abscisse du point : "; cin >> p.x;
    cout << "Tapez l'ordonnée du point : "; cin >> p.y;
}
```

```
void afficher_point(point p)
{
    cout << "Abscisse du point : " << p.x << endl;
    cout << "Ordonnée du point : " << p.y << endl;
}

double distance(point a, point b)
{
    double dx,dy;
    dx = a.x - b.x;
    dy = a.y - b.y;
    return sqrt( dx*dx + dy*dy );
}

void milieu(point a, point b, point &m)
{
    m.x = (a.x + b.x) /2;
    m.y = (a.y + b.y) /2;
}
```

**Fichier main.cpp**

```
#include "point.h"
#include <iostream>

using namespace std;

int main()
{
    point X,Y,Z;
    double d;

    cout << "SAISIE DE X" << endl;
    saisir_point(X);

    cout << "SAISIE DE Y" << endl;
    saisir_point(Y);

    d = distance(X,Y);
    cout << "La distance de X à Y est : " << d << endl;

    milieu(X,Y,Z);
    cout << "AFFICHAGE DU POINT Z" << endl;
    afficher_point(Z);

    return 0;
}
```

- **Explications**

- Le fichier point.h va contenir :
  - l'utilisation des directives `#ifndef`, `#define` et `#endif` pour éviter l'inclusion multiple des fichiers `.h`.
  - la définition de la structure point
  - les entêtes des fonctions manipulant des points.
- Le fichier `point.cpp` commence par `#include "point.h"`. Comme il utilise `cin` et `cout`, il contient aussi `#include<iostream>`. Comme il utilise la fonction `sqrt`, il contient aussi `#include<cmath>`. Il contient également les 4 fonctions du module.
- Le fichier `main.cpp` contient notre fonction `main()` qui est identique à celle de l'exemple 2.

### Structuration de notre programme et de notre réflexion

- Au lieu d'avoir un seul grand fichier contenant le main et toutes les fonctions , il y a plusieurs petits fichiers .
- Lorsqu'on écrit le fichier `point.h`, on réfléchit à la spécification de chaque fonction : QUE fait notre fonction ?
- Lorsqu'on écrit le fichier `point.cpp`, on réfléchit à comment on implémente ces spécifications : COMMENT RÉALISER cette fonctionnalité ?
- Lorsqu'on écrit notre fonction `main`, on réfléchit à l'utilisation des fonctions de `point.h` en vue de résoudre notre programme : COMMENT UTILISER ces fonctions ?

### Gérer une liste d'employés

On va maintenant réaliser un programme plus complexe : un service du personnel veut gérer des employés qui possèdent chacun un nom, un prénom et un salaire. On veut manipuler une liste d'employés grâce au menu suivant :
1. Ajouter un employé
2. Afficher la liste
3. Rechercher un employé
4. Quitter

### Cahier des charges

Le cahier des charges est un document écrit spécifiant avec la plus grande précision possible ce que doit faire notre programme.
Un employé est défini par un nom, un prénom et un salaire. Le nom et le prénom comporteront au maximum 9 caractères utiles. Lorsqu'on rajoute un employé dans la liste, on le rajoute à la fin de la liste.
Lorsqu'on affiche la liste des employés, on l'affiche dans le même ordre que celui de la saisie.
Lorsqu'on recherche un employé, on saisit un nom, et on affiche toutes les caractéristiques de tous les employés portant ce nom.

### Analyse du problème

On réfléchit au problème et on définit un ensemble de structures de données et de fonctions permettant de répondre au problème.
Il est bon de séparer les fonctions réalisant des entrées-sorties et les fonctions manipulant les données.
A la fin de cette analyse la structuration en modules de notre programme doit apparaître.

### Résultat de notre réflexion

Dans notre programme, il y a 3 entités :
Un employé composé d'un nom, d'un prénom, et d'un salaire.
Une liste d'employés sur laquelle on peut ajouter un employé, afficher la liste et faire des recherches.
Une interface homme machine sous la forme de menu en mode texte.

Chacune de ces entités correspondra à un module.

### Le module employé

Il contiendra la définition d'une structure employé.
Dans ce module, il y aura 2 fonctions : une pour saisir les caractéristiques d'un employé et une autre pour les afficher à l'écran.

### Le module liste

Notre liste d'employés sera représentée par un tableau statique de 100 employés et un entier nb qui représente le nombre d'employés réellement présents dans la liste. On définira une telle structure dans ce module.
On définira également 4 fonctions pour manipuler cette liste :

- une fonction pour initialiser la liste à la liste vide.
- une fonction pour afficher une liste d'employés.
- une fonction pour ajouter un employé dans la liste.
- une fonction pour rechercher tous les employés portant un nom donné.
- **Remarque :** il y a de nombreuses méthodes pour représenter notre liste, nous en verrons une autre au chapitre suivant.

### Le module menu

Il permet de gérer notre interface graphique en mode texte. Il comporte :

- une fonction menu qui contient la boucle principale de notre menu.
- une fonction choix qui affiche le menu et saisit le choix de l'utilisateur.
- une fonction traiter_choix qui effectue la manipulation souhaitée par l'utilisateur sur la liste.

### Exemple 4 : gérer une liste d'employés

**Fichier emp.h**

```
#ifndef EMP_H
#define EMP_H

struct employe
{
    char nom[10];
    char prenom[10];
    double salaire;
};

void saisir_employe(employe &e);
//Permet de saisir un employé

void affiche_employe(employe e);
//Permet d'afficher un employé
#endif
```

**Fichier emp.cpp**

```
#include "emp.h"
#include <iostream>
using namespace std;

void saisir_employe(employe &e)
{
    cout<<"Tapez le nom : "; cin>>e.nom;
    cout<<"Tapez le prenom : "; cin>>e.prenom;
    cout<<"Tapez le salaire : "; cin>>e.salaire;
}

void affiche_employe(employe e)
{
    cout<< e.nom <<" "<< e.prenom <<" " << e.salaire <<endl;
}
```

**Fichier liste.h**

```
#ifndef LISTE_H
#define LISTE_H

#include"emp.h"

const int liste_pleine=-1;
const int liste_nb_max=100;

struct liste
{
    int nb;
    employe t[liste_nb_max];
};

void init_liste(liste &l); //Initialise la liste à 0 employé
int ajoute(liste &l, employe e); // Ajoute en employé : renvoi -1 si le
 tableau est plein 0 sinon
void affiche(liste l); // Affiche la liste
void recherche(liste l1, char nom[],liste &l2); // met dans l2 tous les
 employés de l1 portant le nom nom
#endif
```

**Fichier liste.cpp**

```
#include "liste.h"
#include<iostream>
using namespace std;
#include<cstring>

void init_liste(liste &l)
{
```

```
    l.nb=0;
}

int ajoute(liste &l, employe e)
{
    int r;
    if (l.nb == liste_nb_max) r = liste_pleine;
    else { r=0; l.t[l.nb]=e; l.nb++; }
    return r;
}

void affiche(liste l)
{
    int i;
    if(l.nb==0) cout<<"LISTE VIDE"<<endl;
    for(i=0; i<l.nb; i++) affiche_employe(l.t[i]);
}

void recherche(liste l1, char nom[],liste &l2)
{
    int i;
    init_liste(l2);
    for(i=0; i<l1.nb; i++) if(strcmp(l1.t[i].nom, nom)==0) ajoute(l2, l1.t[i]);
}
```

**Fichier menu.h**

```
#ifndef MENU_H
#define MENU_H
#include "liste.h"

void menu(liste &l);
int choix();
bool traiter_choix(liste &l,int choix);
#endif
```

**Fichier menu.cpp**

```
#include "menu.h"
#include <iostream>
using namespace std;

int choix()
{
    int i;
    cout << "1.Ajoute un employe" << endl;
    cout << "2.Afficher la liste" << endl;
    cout << "3.Rechercher un employe" << endl;
    cout << "4.Quitter" << endl;
```

```
    cout << "Votre choix :"; cin >> i;
    return i;
}

void menu(liste &l)
{
    bool fini;
    int i;
    do {
        i = choix();
        fini = traiter_choix(l, i);
    }while(fini==false);
}

bool traiter_choix(liste &l,int choix)
{
    employe e;
    char nom[10];
    liste l2;
    int r;
    bool fini = false;

    switch(choix)
    {
      case 1:
        saisir_employe(e);
        r = ajoute(l, e);
        if(r == liste_pleine) cout << "La liste est pleine" << endl;
      break;

      case 2:
        affiche(l);
      break;

      case 3:
        cout << "Tapez le nom :"; cin >> nom; recherche(l, nom, l2);
        cout << "Voici le resultat de la recherche :" << endl;
        affiche(l2);
      break;

      case 4:
        fini = true;
      break;
    }
    return fini;
}
```

**Fichier main.cpp**

```
#include"liste.h"
#include"menu.h"
int main()
{
    liste l;
    init_liste(l);
    menu(l);
    return 0;
}
```

- **Explications**
  - Notre programme sera constitué de 7 fichiers :
    - emp.h et emp.cpp qui définissent notre module employé.
    - liste.h et liste.cpp qui définissent notre module liste.
    - menu.h et menu.cpp qui définissent notre module menu.
    - La fichier main.cpp qui contient notre programme fonction main().
  - **Un point technique**
    - Si l est une liste d'employé, le i-ième employé du tableau t de la liste l sera noté l.t[i] . Cet élément est du type emp.
    - Le salaire du i-ième employé du tableau t de la liste l sera noté l.t[i].salaire . Cet élément est du type double.
    - Le nom du i-ième employé du tableau t sera noté l.t[i].nom . Il s'agit d'un tableau de char.
    - Si on voulait accéder au j-ème caractère du nom du i-ième employé du tableau t de la liste l, il faudrait écrire l.t[i].nom[j]
    - **Il ne faut pas se laisser impressionner par ces notations !**
  - **le module employé**
    - La structure emp sera composé de 2 chaînes de caractères (le nom et le prénom) et d'un double (le salaire de l'employé).
    - La fonction void saisir_employe(employe &e) permet de saisir au clavier les caractéristique d'un employé : employe est un paramètre en sortie de cette fonction.
    - La fonction void affiche_employe(employe e) permet d'afficher à l'écran les caractéristiques d'un employé. employe est un paramètre en entrée de cette fonction.
  - **le module liste**
    - La structure liste sera composée d'un tableau d'employés et d'un entier nb qui est le nombre d'employés réellement ajoutés dans la liste. On peut définir des tableaux de structures de la même manière que n'importe quelle autre tableau. La taille du tableau est défini par la constante liste_nb_max.
    - La fonction void init_liste(liste &l) initialise la liste d'employés à la liste vide. Elle se contente de mettre à 0 le nombre d'employés.
    - La fonction int ajoute(liste &l, employe e) ajoute l'employé e à la fin de la liste d'employés l. Cette fonction renvoie 0 si tout s'est bien passé, elle renvoie liste_pleine si le tableau d'employés est plein.
    - La fonction void affiche(liste l) affiche la liste d'employés à l'écran : cette fonction utilise la fonction affiche_employe du module employé.
    - La fonction void recherche(liste l1, char nom[],liste &l2) recherche dans la liste l1 tous les employés portant le nom nom et met ces employés dans la liste l2. l1et nom sont des paramètres en entrée de la fonction affiche et l2 sera un paramètre en sortie.
  - **le module menu**
    - Ce module contient l'interface homme machine de notre application.

- La fonction void menu(liste &l) permet de gérer la liste l grâce à notre menu. Cette fonction contient la boucle do … while principale de notre programme.
- La fonction int choix() affiche le menu de notre programme, demande à l'utilisateur de faire un choix dans le menu et renvoie ce choix par un return.
- La fonction bool traiter_choix(liste &l,int choix) effectue le traitement désigné par l'entier choix sur la liste l. Si ce choix a pour conséquence la sortie du menu principal la fonction renvoie true, sinon elle renvoie false. Cette fonction est essentiellement constituée d'un switch et de l'appel à des fonction du module liste.

### Structuration de notre programme

Il y a de nombreuses fonctions courtes (moins de 20 lignes) réalisant chacune une fonction bien précise.
Ces fonctions sont rassemblées en 3 modules : employe, liste et menu plus bien sûr le programme principal.
Cette structuration (qui peut apparaître au départ fastidieuse) permet de développer plus rapidement en rendant plus courte la phase de débogage.
Il est très facile de rajouter de nouvelles fonctionnalités dans notre application.

### Approche modulaire et compilation séparée

Notre programme est constitué de 3 modules et de 7 fichiers.
Lorsqu'un seul des modules est modifié le compilateur ne va pas recompiler la totalité des fichiers de notre applications mais seulement ceux qui ont été modifiés.
Ce point peut être important lorsqu'un programme est constitué de centaines de modules.

### Conclusion

La notion de structure est une notion très importante en programmation et nous permet de définir des éléments plus complexes que les simples types de base.
Elles permettent de gagner un niveau d'abstraction supplémentaire par rapport aux type de base.
L'approche modulaire permet d'affiner notre méthodologie de programmation et de structurer notre réflexion en plusieurs phases :
- comment représenter une notion donnée ?
- quelles sont les fonctions nécessaires pour manipuler cette notion ?
- La phase technique : comment écrire ces fonctions ?

On constate que la phase technique, bien qu'indispensable, n'est pas l'élément crucial du développement d'un programme : en amont, il y a une phase d'analyse et de structuration bien plus importante.

## Exercices sur les structures

### EXERCICE 1

a) Écrire une structure fraction composée de 2 entiers : un numérateur et un dénominateur.
b) Écrire une fonction pgcd de 2 entiers positifs.
c) Écrire une fonction normalise qui normalise une fraction de la manière suivante :
- le numérateur et le dénominateur doivent être premier entre eux
- le dénominateur est positif

d) Écrire une fonction saisir qui permet de saisir une fraction.
e) Écrire une fonction affiche qui affiche une fraction.
f) Écrire une fonction somme qui calcule la somme de 2 fractions. Le résultat doit être normalisé.
g) Écrire 3 autres fonctions : différence, multiplie et divise comme au f)
h) Écrire un programme qui permet de saisir les fractions A, B, D, E et F et qui calcule (A+B)/(D-E*F) .

Le résultat sera affiché à l'écran.

## EXERCICE 2

On veut gérer des produits dans un entrepôt. Un produit est défini par 2 chaînes de caractères : le code produit ("H567" par exemple et qui comporte au maximum 9 caractères utiles), l'intitulé ("pots de peinture" et qui comporte au maximum 99 caractères utiles) et un entier qui indique la quantité en stock (803 par exemple). Il faut gérer une liste de produits en veillant à ce qu'il n'y ait pas deux produits avec le même code produit. Bien sûr la quantité en stock ne peut pas être négative. La liste comportera au maximum 100 produits.
Il faut gérer la liste grâce au menu suivant :
1. Ajouter un produit (on tape le code produit et l'intitulé, la quantité est initialisée à 0).
2. Afficher la liste de produits.
3. Supprimer un produit en tapant le code produit.
4. Acheter un produit en tapant le code produit et la quantité achetée.
5. Vendre un produit en tapant le code produit et la quantité vendue.
6. Quitter

On veillera à bien décomposer ce problème en différents modules et à mener une réflexion sur les fonctions nécessaires dans chaque module.

## EXERCICE 3

On veut écrire un programme qui permet de gérer une liste de disques. Un disque est défini par un code référence (une chaîne de caractères par exemple "H345"), nom de chanteur, un nom de chanson et prix. Le code référence fera au maximum 9 caractères utiles, le titre du disque et le nom du chanteur feront au maximum 99 caractères utiles.
On veut gérer une liste de disques grâce au menu suivant :
1. Ajouter un disque à la liste.
2. Afficher la liste.
3. Supprimer un disque grâce à son code référence.
4. Afficher tous les disques d'un chanteur donné.
5. Afficher tous les disques ayant un titre donné.
6. Quitter

On veillera à bien décomposer ce problème en différents modules et à mener une réflexion sur les fonctions nécessaires dans chaque module.

# Sources et contributeurs de l'article

**Programmation C++ (débutant)/Les structures** *Source*: http://fr.wikibooks.org/w/index.php?oldid=334756 *Contributeurs*: Charly, DavidL, JackPotte, Merrheim, Philippe.gouin, Sub, Zulul, 32 modifications anonymes

# Licence

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/