université PARIS-SACLAY

Correction TP de programmation n°5

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

Structures de données

Cette séance de travaux pratiques est dédiée à l'utilisation des structures.

▸ **Exercice 1. (Fractale de Mandelbrot)**

Le but de l'exercice est d'afficher le célèbre ensemble fractal de Mandelbrot. Pour ce faire, on considère la suite de nombres complexes paramétrée par un complexe donné c, définie par

$$z_0 = 0 \qquad \text{et} \qquad z_{n+1} = z_n^2 + c\,.$$

L'ensemble de Mandelbrot est l'ensemble des points dont les coordonnées complexes c sont telles que la suite associée ne tend pas vers l'infini.

1. Le type `Complexe`, la fonction `creeComplexe`, la fonction `egaleComplexe` et la fonction `ajouterComplexe` vous sont données dans le fichier `mandelbrot.cpp` fourni dans l'archive (ainsi que ci-dessous). Lire leur code pour bien les comprendre.

```
struct Complexe {
    float re, im;
};

Complexe creeComplexe(float reel, float imaginaire){
    Complexe c;
    c.re = reel;
    c.im = imaginaire;
    return c;
}

bool egaleComplexe(Complexe c1, Complexe c2){
    return (c1.re==c2.re)&&(c1.im==c2.im);
}

Complexe ajouterComplexe(Complexe c1, Complexe c2){
    Complexe resultat;
    resultat.re = c1.re + c2.re;
    resultat.im = c1.im + c2.im;
    return resultat;
}
```

2. Écrire la fonction `multiplierComplexe` qui prend en entrée deux complexes et renvoie leur produit. **Faites très attention à la formule** : $(a_1 + ib_1) * (a_2 + i * b_2) = a_1 * a_2 - b_1 * b_2 + i(a_1 * b_2 + a_2 * b_1)$. En effet, la moindre erreur risque de vous faire perdre une demi-heure (tests qui ne passent pas, mandelbrot tout bizarre).

```
Complexe multiplierComplexe(Complexe c1, Complexe c2){
    Complexe resultat;
    resultat.re = c1.re*c2.re - c1.im*c2.im;
    resultat.im = c1.im*c2.re + c1.re*c2.im;
    return resultat;
}
```

3. Proposer trois tests de la fonction `multiplierComplexe`, puis appeler la fonction de test dans le `main`, enfin exécuter pour vérifier. Indication : voici un exemple de test :

```
ASSERT(egaleComplexe(multiplierComplexe(creeComplexe(0, 0),
                                        creeComplexe(1, 1)),
                     creeComplexe(0, 0)));
```

```
void testMultiplierComplexe(){
    ASSERT(egaleComplexe(multiplierComplexe(creeComplexe(0, 0),
                                            creeComplexe(1, 1)),
                         creeComplexe(0, 0)));
    ASSERT(egaleComplexe(multiplierComplexe(creeComplexe(1, 0),
                                            creeComplexe(4, 6)),
                         creeComplexe(4, 6)));
    ASSERT(egaleComplexe(multiplierComplexe(creeComplexe(0, 1),
                                            creeComplexe(4, 6)),
                         creeComplexe(-6, 4)));
}
```

4. Écrire une fonction qui calcule le module d'un nombre complexe. On rappelle que $|a + ib| = \sqrt{a^2 + b^2}$ (on pourra utiliser la bibliothèque `cmath` vue dans un TP précédent). Tester votre fonction.

```
#include <cmath>

float moduleComplexe(Complexe c){
    return sqrt(pow(c.re,2)+pow(c.im,2));
}
```

5. Écrire une fonction booléenne `znResteBorne` qui prend un complexe c, et renvoie vrai si la suite z_n associée ne tend pas vers l'infini, faux sinon. On calculera la suite des termes z_n, jusqu'à ce que le module de z_n devienne plus grand que 1000 (auquel cas on dira que la suite tend vers l'infini), ou 1000 itérations ont été faites (auquel cas on dira que la suite reste bornée).

```
bool znResteBorne(Complexe c){
    Complexe z = creeComplexe(0,0);
    for(int i=0; i<1000; i++){
        if (moduleComplexe(z)>1000){
            return false;
        }
        z = ajouterComplexe(multiplierComplexe(z,z),c);
    }
    return true;
}
```

La procédure `mandelbrot` qui vous est donnée ci-dessous utilise la fonction `znResteBorne` que vous venez d'écrire pour afficher l'ensemble de Mandelbrot. Elle fait varier x entre -2 et 0.5 et y entre -1.5 et $+1.5$, et fait un affichage alphanumérique sur 80×80 caractères.

```
void mandelbrot(){
    const float xmin = -2;
    const float xmax = 0.5;
    const float ymin = -1.5;
    const float ymax = 1.5;
    const int resol = 79;
    for(int i=0; i<=resol; i++) {
        for(int j=0; j<=resol; j++) {
            if(znResteBorne(creeComplexe((resol-j)*xmin/resol+j*xmax/resol,
                                         (resol-i)*ymax/resol+i*ymin/resol))) {
                    cout << '#';
            }
            else {
                    cout << ' ';
            }
        }
        cout << endl;
    }
}
```

6. Exécuter `mandelbrot`, puis modifier cette procédure pour zoomer sur la figure, en faisant varier x et y sur des plus petits intervalles, autour de points pas trop loin de la frontière.

▸ Exercice 2. (Couples faisables)

1. Déclarer un type structure `Personne` pour représenter une personne avec seulement quatre informations : son nom, son prénom, son année de naissance, et son sexe. On pourra utiliser le type énuméré suivant :

   `enum genre {homme, femme};`

   Une fois ce type declaré, vous pouvez déclarer une variable de type genre, et tester si cette variable est égale à femme ou homme.

```
struct Personne {
    string nom;
    string prenom;
    int annee;
    genre sexe;
};
```

2. Écrire une fonction `nouvellePersonne` qui prend en entrée les informations d'une personne et renvoie une `Personne`.

```
Personne nouvellePersonne(string nom, string prenom, int annee, genre sexe){
    Personne p;
    p.nom=nom;
    p.prenom=prenom;
    p.annee=annee;
    p.sexe=sexe;
    return p;
}
```

3. On va utiliser un vecteur pour contenir une population de dix personnes. L'initialisation d'un vecteur de six entiers peut se faire comme ceci :
`vector<int> vecteur = { 1, 2, 3, 5, 6, 7 };`
Pour un vecteur de `Personne`, on procède de la même manière avec un appel à la fonction `nouvellePersonne`. Initialiser un tableau (vecteur) `population` de dix personnes, nées entre 1975 et 1990, directement dans le code.

```
int main(){
    vector<Personne> population = {
        nouvellePersonne("Bonnet", "Jean", 1979, homme),
        nouvellePersonne("Lefebvre", "Michel", 1981, homme),
        nouvellePersonne("Leroy", "Pierre", 1977, homme),
        nouvellePersonne("Petit", "Philippe", 1984, homme),
        nouvellePersonne("Morel", "Alain", 1990, homme),
        nouvellePersonne("Fournier", "Marie", 1985, femme),
        nouvellePersonne("Durand", "Nathalie", 1989, femme),
        nouvellePersonne("Dubois", "Isabelle", 1975, femme),
        nouvellePersonne("Moreau", "Catherine", 1982, femme),
        nouvellePersonne("Girard", "Sylvie", 1987, femme)
    };
    coupleFaisable(population);
    return 0;
}
```

4. Un couple de deux personnes est dit «faisable» si les deux personnes sont de sexes opposés, avec moins de sept ans de différence. Écrire une procédure `afficherCoupleFaisable` qui prend en entrée un tableau de `Personne` et qui affiche les couples faisables. Utiliser deux boucles imbriquées, qui itèrent toutes les deux sur les éléments du tableau. On pourra d'abord écrire une fonction booléenne `estFaisable` qui teste si deux personnes données peuvent former un couple. Tester `afficherCoupleFaisable`.

```
bool estFaisable(Personne p1, Personne p2){
    return ((abs(p2.annee-p1.annee) < 7) && (p1.sexe != p2.sexe));
}

void afficherCoupleFaisable(vector<Personne> population){
    int s = population.size();
    for(int i=0; i<s-1; i++){
        for(int j=i+1; j<s; j++){
            if (estFaisable(population[i],population[j])){
                cout << population[i].prenom << " " << population[i].nom << " et " << population[j].prenom
            }
        }
    }
}
```

5. On souhaite tenir compte du fait qu'environ 7% de la population est homosexuelle. Modifier le type structure, pour enregistrer si la personne est hétérosexuelle ou homosexuelle. Modifier la fonction `nouvellePersonne` et la procédure `afficherCoupleFaisable` pour en tenir compte (si on a pris soin de décomposer `afficherCoupleFaisable` comme suggéré à la question 3, seul le code de `estFaisable` est à modifier ; le code de `afficherCoupleFaisable` reste inchangé).

```
enum genre {homme, femme};

enum orient {homo, hetero};

struct Personne {
    string nom;
    string prenom;
    int annee;
    genre sexe;
    orient orientation;
};

Personne nouvellePersonne(string nom, string prenom, int annee, genre sexe, orient orientation){
    Personne p;
    p.nom=nom;
    p.prenom=prenom;
    p.annee=annee;
    p.sexe=sexe;
    p.orientation=orientation;
    return p;
}

bool estFaisable(Personne p1, Personne p2){
    if(p1.orientation==p2.orientation){
            if(p1.orientation=hetero){
                return ((abs(p2.annee-p1.annee) < 7) && (p1.sexe != p2.sexe));
            }
            else{
                return ((abs(p2.annee-p1.annee) < 7) && (p1.sexe == p2.sexe));
            }
```

```
    }
    else{
        return false;
    }
}
```

6. Donner un nom au type du vecteur `population` afin de simplifier la déclaration de type dans l'en-tête de `coupleFaisable`.

```
typedef vector<Personne> Population;

void coupleFaisable(Population population)
```

▸ **Exercice 3.** Réimplanter la fonction racine carrée du T.P. précédent, en utilisant la structure `Complexe` de l'exercice 1.