Correction TP de programmation n°3

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Surcharge des opérateurs

Dans ce TP, nous allons utiliser la surcharge des opérateurs pour comparer et afficher des structures et énumérations.

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

On rappelle qu'un nombre complexe est un nombre qui s'écrit $a + ib$ où a est appelé partie réelle et b partie imaginaire. On représentera un nombre complexe par une structure avec deux champs : `re` pour la partie réelle et `im` pour la partie imaginaire.

Le but de l'exercice est d'afficher le célèbre ensemble fractal de Mandelbrot. Cette fractale est définie de la manière suivante : on fixe un nombre complexe c. On considère la suite de nombres complexes définie par

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

En fonction des valeurs de c, la suite peut avoir divers comportements. Voici quelques exemples :

c	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	5	26	667	458330	210066388901	$\approx 4.4 \times 10^{22}$	$\approx 1.9 \times 10^{45}$
-1	0	-1	0	-1	0	-1	0	-1	0	-1
i	0	i	$i-1$	$-i$	$i-1$	$-i$	$i-1$	$-i$	$i-1$	$-i$

On voit que pour les valeurs 0, -1 et i, la suite reste bornée alors que pour 1, elle tend vers l'infini. L'ensemble de Mandelbrot est l'ensemble des points du plan dont les coordonnées complexes c sont telles que la suite associée reste bornée. On peut montrer que si le module de z_n dépasse 2 alors la suite va tendre vers l'infini. La fonction `znResteBorne` va calculer les 1000 premiers termes. Si le module de la suite n'a pas dépassé 2, on va considérer qu'elle reste bornée.

1. Le type `Complexe` et les opérateurs `==`, `!=`, `<<`, `+` et `-` vous sont donnés dans le fichier `complex.cpp`. Lire leur code pour bien les comprendre.

```
1
2  // Un nombre complexe se compose d'une partie réelle et d'une partie imaginaire
3  struct Complexe {
4      float re, im;
5  };
6
7  // Surcharge de l'operateur ==
8  bool operator==(Complexe z, Complexe w) {
9      return z.re == w.re and z.im == w.im;
10 }
11
12 // Surcharge de l'operateur !=
13 bool operator!=(Complexe z, Complexe w) { return not(z == w); }
14
15 // Surcharge de l'operateur <<
```

```
16 std::ostream &operator<<(std::ostream &out, Complexe z) {
17     if (z.im >= 0) {
18         out << z.re << "+i" << z.im;
19     } else {
20         out << z.re << "-i" << -z.im;
21     }
22     return out;
23 }
24
25 // Surcharge de l'operateur +
26 Complexe operator+(Complexe z, Complexe w) {
27     return {z.re + w.re, z.im + w.im};
28 }
29
30 // Surcharge de l'operateur - avec un seul nombre
31 Complexe operator-(Complexe z) { return {-z.re, -z.im}; }
32
33 // Surcharge de l'operateur - avec deux nombres
34 Complexe operator-(Complexe a, Complexe b) { return a + (-b); }
```

2. Surcharger l'opérateur de produit `*` qui prend en entrée deux complexes et renvoie leur produit. **Faites très attention à la formule** :

$$(a_1 + ib_1) * (a_2 + ib_2) = (a_1a_2 - b_1b_2) + i\,(a_1b_2 + a_2b_1)\,.$$

En effet, la moindre erreur risque de vous faire perdre une demi-heure (tests qui ne passent pas, dessin de l'ensemble de Mandelbrot tout bizarre).

- -

```
1 Complexe operator*(Complexe z, Complexe w) {
2     Complexe ans;
3     ans.re = z.re * w.re - z.im * w.im;
4     ans.im = z.re * w.im + z.im * w.re;
5     return ans;
6 }
```

- -

3. Proposer trois tests dans la fonction `testMultiplication` et exécuter pour vérifier. Indication : s'inspirer de l'exemple de test donné :

```
1     Complexe z1 = {2.4, 1};
2     Complexe z2 = {-5, 3.8};
3     Complexe z3 = {-15.8, 4.12};
4     CHECK (z1*z2 == z3);
```

- -

```
1 void testMultiplication() {
2     Complexe z1 = {2.4, 1};
3     Complexe z2 = {-5, 3.8};
4     Complexe z3 = {-15.8, 4.12};
5     CHECK (z1*z2 == z3);
6     //
```

```
7       Complexe z4 = {0, 0};
8       Complexe z5 = {1, 1};
9       CHECK (z4*z5 == z4);
10      Complexe z6 = {3, 2};
11      Complexe z7 = {3, -2};
12      Complexe z8 = {13, 0};
13      CHECK (z6*z7 == z8);
14      //
15  }
```

4. Surcharger la fonction `abs` qui calcule la valeur absolue (aussi appelée module) d'un nombre complexe. On rappelle que $|a + ib| = \sqrt{a^2 + b^2}$ (on pourra utiliser la fonction `sqrt` de la bibliothèque `cmath`).

```
#include <cmath>

1  float abs(Complexe z) {
2      //
3      return sqrt(z.re * z.re + z.im * z.im);
4      //
5  }
```

5. Tester votre fonction. Attention ! À cause de l'utilisation des `float` qui sont des approximations, il y a une forte probabilité pour que vous ayez des problèmes avec les tests de cette fonction (si vous n'en avez pas eu avant). On pourra tester l'égalité approximative de deux float avec la fonction `presqueEgal` qui vous est fournie et qui sera importante dans les exercices 3 et 4.

```
1  void testAbs() {
2      Complexe z1 = {0, 0};
3      CHECK(presqueEgal(abs(z1), 0));
4      //
5      Complexe z2 = {-5, 3.8};
6      CHECK(presqueEgal(abs(z2), 6.280127387));
7      Complexe z3 = {3, -2};
8      CHECK(presqueEgal(abs(z3), 3.605551275));
9      //
10 }
```

6. Surcharger la fonction `presqueEgal` pour les nombres complexes et proposer trois tests pour la vérifier. S'inspirer de l'exemple de test fourni :

```
1      Complexe z1 = {2.436545828384, 1.189346553893};
2      Complexe z2 = {-5.359712984532, 3.8326749867};
3      Complexe z3 = {-17.617565101, 2.963932082};
4      CHECK(presqueEgal(z1*z2, z3));
```

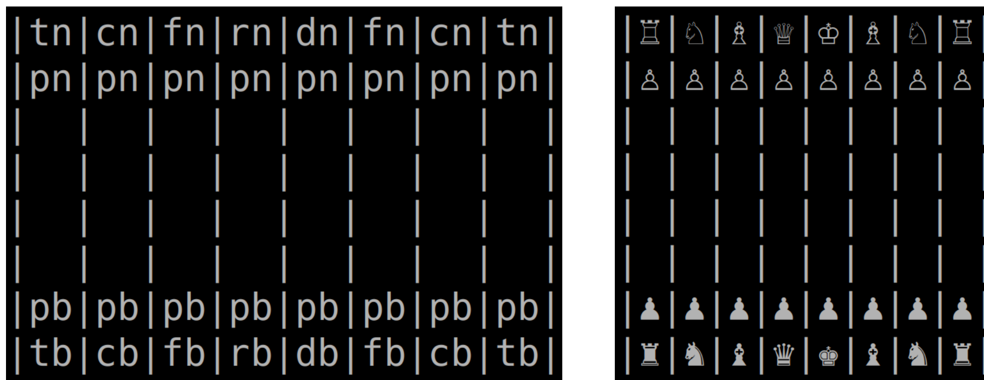
```
1     return presqueEgal(z.re, w.re) and presqueEgal(z.im, w.im);
2 }
3 void testPresqueEgal() {
```

7. Exécuter la fonction `mandelbrot`. Pour obtenir des détails plus intéressants dans la fractale, augmenter le paramètre `resol` (les valeurs 100, 250 et 500 sont recommandées) ou modifier les valeurs de `xmin`, `xmax`, `ymin` et/ou `ymax` autour de points proches de la frontière comme -2.

Les autres fonctions du fichier `complex.cpp` seront faites à l'exercice 4. Passez à l'exercice 2.

▸ Exercice 2.(Echecs)

Le jeu d'échec (voir https://fr.wikipedia.org/wiki/Échecs) est un jeu qui se joue sur un plateau 8×8. Il comporte 6 types de pièces différents : Le Roi (r, ♚), la dame (d, ♛), le fou (f, ♝), le cavalier (c, ♘), la tour (t, ♖) et le pion (p, ♙). Il y a deux joueurs, l'un possède un jeu de pièces blanches, l'autre un jeu de pièces noires. La figure ci-dessus montre deux affichages de la position initiale du jeu d'échec (un affichage simple et un utilisant les caractères unicode) :



Le but de cet exercice est de mettre en place la structure initiale d'un jeu d'échec. Le fichier `echec.cpp` contient un patron de cette structure. En particulier le fichier contient les déclarations des types suivants :

```
enum class CouleurPiece { noir, blanc, vide };
enum class SortePiece { pion, tour, cavalier, fou, dame, roi, vide };
struct Piece {
  CouleurPiece couleur;
  SortePiece sorte;
};
struct Coord {
  int lig;  // ligne
  int col;  // colonne
};
using Echiquier = array< array<Piece, 8>, 8>;
```

1. Surcharger les opérateurs `==`, `!=` pour le type `Coord`.
2. Compléter, compiler et exécuter la fonction `testEgalCoord` pour tester les deux opérateurs précédents.

3. Surcharger les opérateurs `==`, `!=` pour le type `Piece`.
4. Compléter, compiler et exécuter la fonction `testEgalPiece` pour tester les deux opérateurs précédents.
5. Surcharger l'opérateur `<<` pour les types `CouleurPiece` et `Piece` : pour l'instant on effectue un affichage simple, par exemple le roi blanc sera affiché `rb`.
6. Vérifier que l'affichage d'une pièce se passe bien.
7. On peut maintenant afficher tout un échiquier avec l'opérateur `<<`. Vérifier que ça marche bien. Pour ceci utilisez la fonction `createEchiquier` fournie qui permet de créer un échiquier à partir d'une chaîne de caractère. Voir les exemples `Einit` et `ETaverner1889`.
8. Surcharger la fonction `createEchiquier`, pour avoir une version qui ne prend pas de paramètre et qui initialise la position de départ. On pourra utiliser la constante fournie qui donne l'ordre des pièces sur les premières et dernières lignes.

```
static const array<SorteePiece, 8> lignePieces = {
  SortePiece::tour, SortePiece::cavalier, SortePiece::fou,
  SortePiece::roi, SortePiece::dame, SortePiece::fou,
  SortePiece::cavalier, SortePiece::tour };
```

9. Écrire une fonction `deplacePiece` qui prend un échiquier et deux coordonnées `ci` et `cf` et qui déplace la pièce depuis la position `ci` vers la position `cf`. Si la case `ci` est vide, on pourra déclencher une erreur avec la commande

```
throw invalid_argument("Pas de piece à cet emplacement");
```

Note : On ne demande pas de vérifier que le déplacement est valide.

Si votre environnement de travail est bien configuré, vous pouvez afficher les pièces directement en utilisant les caractères unicode. Ils sont stockés dans des chaînes de caractères spéciales préfixée par `u8`. On peut mettre une multitude de caractères différents dans ces chaînes allant des smileys aux runes celtique en passant par les idéogrammes égyptiens ! Vous pouvez tester si tout se passe bien en lançant la fonction `checkUnicode` définie en début de fichier.

ATTENTION ! Selon que vous travaillez avec un fond noir ou blanc, les couleurs des pièces d'échec seront inversées. Du coup, il est possible qu'une pièce blanche dans votre éditeur apparaisse noir dans le terminal...

10. La constante `pieces` définie en début de fichier contient un tableau des caractères unicode des différentes pièces. Modifiez vos fonctions d'affichage pour qu'elles utilisent les caractères unicode.

▸ **Exercice 3.(Calcul de la racine carrée)**

Soit a un nombre réel positif. La racine carrée $b = \sqrt{a}$ de a est l'unique nombre réel positif qui vérifie $b^2 = a$. On montre en mathématiques que, étant donné un réel positif a, la suite

$$u_0 := a, \qquad u_{n+1} := \frac{u_n + a/u_n}{2}$$

converge vers $\sqrt{a}$. Le programme `suite_sqrt.cpp` fourni affiche les 10 premiers termes de la suite u_n ainsi que u_n^2 pour $a = 2$:

```
1  #include <iostream>
2  #include <iomanip>
3
4  using namespace std;
```

```
5
6  int main(){
7    int n;
8    float a=2., un, un1; // u_n et u_{n+1}
9
10   n = 0;  un = a;
11   while (n<10) {
12     un1 = (un + a/un)/2.;        // calcul de u_{n+1}
13     n = n+1; un = un1;           // passage de n à n+1
14     cout << "n=" << n << ", un=" << un << ", un^2=" << un*un << endl;
15   }
16   return 0;
17 }
```

1. Observer l'affichage pour vérifier que u_n converge bien vers $\sqrt{2}$, et u_n^2 vers 2.
2. Modifier le programme précédent pour que le calcul se fasse à partir d'un nombre a donné par l'utilisateur. Tant que le nombre donné par l'utilisateur est négatif on demande un nouveau nombre.
3. Augmenter la précision de l'affichage en ajoutant `<< setprecision(10)` dans le `cout` qui affiche u_n (avant l'affichage de u_n), et lancer le calcul. Essayez de calculer les racines carrées de 2, 1.1 et 2000000. Que remarquez-vous ?

À cause des erreurs d'arrondi, dans le cas du calcul de $\sqrt{2}$, la valeur de u_n^2 ne tombe jamais sur 2 mais sur un nombre très proche. De plus, en fonction de a, les 10 itérations de la boucle sont pas toujours suffisantes pour avoir une bonne valeur approchée de $\sqrt{a}$. Il faut donc trouver un moyen d'arrêter le calcul au bon moment. Pour cela, on fixe une précision, par exemple $\epsilon = 10^{-6}$, ce qui s'écrit en C++ (au début du programme) :

```
const float EPSILON = 1e-6;
```

et on continue le calcul tant que u_n^2 n'est pas égal à a à ϵ près, c'est-à-dire tant que

$$\left|\frac{u_n^2}{a} - 1\right| \geq \epsilon\,.$$

4. Modifier le programme précédent pour calculer à ϵ près la racine carrée du nombre a donné par l'utilisateur.

✄ -

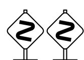
```
1  #include <iostream>
2  #include <iomanip>
3
4  using namespace std;
5
6  const float epsilon = 1e-6;
7
8  int main(){
9    float a, un, erreur;
10   do {
11     cout << "Donnez un nombre positif : ";
12     cin >> a;
13   } while (a <= 0.);
14
15   un = a;
```

```
16    erreur = (un*un/a) - 1;
17    if (erreur < 0) erreur = -erreur;
18    while (erreur >= epsilon)
19      {
20        un = (un + a/un)/2.;
21        erreur = (un*un/a) - 1;
22        if (erreur < 0) erreur = -erreur;
23      }
24    cout << "sqrt(" << a << ") = " << un << ", "<< un << "^2=" << un*un << endl;
25    return 0;
26  }
```

- ✂

▸ **Exercice 4.(Racine carrée des nombres complexes)**
On demande de compléter le programme précédent dans `complex.cpp` :

1. Écrire une fonction `normeCarre` qui retourne le carré de la norme d'un nombre complexe. On rappelle que
$$|a + ib|^2 = a^2 + b^2 \,. \tag{1}$$
2. Écrire une fonction `inverse` qui retourne l'inverse d'un nombre complexe. On rappelle que
$$\frac{1}{a + ib} = \frac{a - ib}{|a + ib|^2} \,. \tag{2}$$
3. Surcharger l'opérateur de division `/` pour les nombres complexe.
4. L'algorithme de calcul de la racine carrée de l'exercice précédent fonctionne en général encore sur les nombres complexes $z = a + ib$: la suite
$$u_0 := z, \qquad u_{n+1} := \frac{u_n + z/u_n}{2} \tag{3}$$
converge (presque toujours, voir la question suivante) vers une racine carrée de z. Surcharger la fonction `sqrt` pour calculer la racine carrée d'un nombre complexe. On considère que l'approximation u est correcte si
$$\frac{|u^2 - z|}{|z|} < 10^{-6} \,, \qquad \text{c'est-à-dire si} \qquad \frac{|u^2 - z|^2}{|z|^2} < 10^{-12} \,. \tag{4}$$
5. En fait, la suite précédente ne converge pas dans le cas particulier où z est un réel négatif. Pour qu'elle se mette à converger, il suffit de changer la valeur de u_0. On pourra dans ce cas choisir $u_0 = z + i\epsilon$. Modifier le programme en conséquence.



▸ **Exercice 5.(Echec Avancé)** Dans l'exercice sur les échecs, la fonction de déplacement de pièces ne vérifie pas si les déplacements sont valides. S'il vous reste du temps, nous vous proposons de coder la validation des déplacements de chaque pièce. Vous trouverez les règles de déplacement sur la page wikipedia https://fr.wikipedia.org/wiki/Échecs.

Selon les pièces, les règles de déplacement peuvent être plus ou moins compliquées. Nous conseillons de les faire dans l'ordre suivant : roi, cavalier, tour, pion, fou, dame. Pour le roi, on ne tiendra pas compte de la règle de l'échec. Pour chacune des pièces, on demande de (entête donné dans le cas du roi, à adapter pour les autres pièces) :

1. Écrire une fonction

   ```
   bool deplacementValideRoi(Echiquier e, Coord ci, Coord cf)
   ```

   qui répond si le déplacement est valide ou non. Une prise d'une pièce adverse est un déplacement valide.

   Le déplacement peut être invalide car
   — Il n'y a pas la pièce du bon type dans la case de départ ;
   — La case d'arrivée est occupée par une pièce de la même couleur ;
   — La trajectoire de la case de départ vers la case d'arrivée ne correspond pas à la sorte de pièce (ex : une tour doit se déplacer en ligne droite, un fou suivant une diagonale...) ;
   — Le déplacement ne se fait pas dans le bon sens (cas du pion) ;
   — Il y a une pièce qui bloque le passage (tour, fou, dame) ;
2. Tester la fonction de déplacement dans une fonction `testDeplacementValideRoi`. Vous pouvez fabriquer des exemples d'échiquiers avec `createEchiquier`.
3. Modifier la fonction `déplacePiece` pour qu'elle vérifie si le déplacement est valide.