MOOC Intro POO C++

Corrigés semaine 3

Les corrigés proposés donnent à chaque fois une solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante. On rappelle que plusieurs solutions sont en général possibles.

Exercice 9 : Affichages (niveau 1)

Par exemple pour les `Points3D` (Exercice 2 du MOOC semaine 1) :

```cpp
class Point3D
{
public:
    // modification du prototype de la methode affiche
    ostream& affiche(ostream& sortie) const;

private:
    double x, y, z;
};

ostream& Point3D::affiche(ostream& sortie) const
{
    // adaptation de la definition de la methode affiche
    sortie << '(' << x << ", " << y << ", " << z << ')';
    return sortie; // ne pas oublier ce return !
}

// surcharge externe de l'operateur d'affichage
ostream& operator<<(ostream& sortie, Point3D const& p)
{
    return p.affiche(sortie);
}
```

Exercice 10 : Nombres complexes (niveau 1)

Pour chaque étape, on ne fournit ici que les prototypes des éléments à réaliser, les définitions étant reproduites dans le listing complet en fin de corrigé.

1. On pourrait se contenter pour l'instant de ne définir que les deux attributs, mais il est préférable de prévoir tout de suite les méthodes d'accès à ces derniers.

   Remarquons que dans le cas présent il est suffisant de fournir, dans l'interface de la classe, uniquement l'accès en lecture (« accesseurs »). Dans le cas des nombres complexes, il n'est en effet pas strictement nécessaire de fournir les méthodes d'accès en écriture (« modificateur »), car l'affectation de valeurs aux attributs se fait plutôt avec les constructeurs et l'opérateur d'affectation.

   ```
   class Complexe {
   public:
       double reel() const;
       double imag() const;

   private:
       double re;
       double im;
   };
   ```

2. Il faut au minimum le constructeur par défaut (on décide naturellement que ce soit le complexe $(0, 0)$ qui soit construit par défaut), et un constructeur admettant deux arguments réels. On peut également ajouter un constructeur à un argument réel, permettant de plonger les réels dans le corps des complexes (partie imaginaire nulle). Non seulement, cet aspect est mathématiquement licite, mais un tel constructeur permet de plus d'éviter de surcharger les opérateurs arithmétiques prenant un réel en membre droit.

   En utilisant des valeurs par défaut pour les arguments, ces trois constructeurs peuvent être définis au moyen d'une seule méthode :

   ```
   class Complexe {
   public:
   Complexe(const double re = 0.0, const double im = 0.0);
   ...
   }
   ```

3. Pour permettre les affichages, il faut définir l'opérateur (interne) de test d'égalité, ainsi que l'opérateur (externe) d'insertion dans un flot :

   ```
   class Complexe {
   public:
       // ...
       bool operator==(const Complexe&) const;
       // ...
   };

   ostream& operator<<(ostream&, const Complexe&);
   ```

   Ce dernier opérateur devant avoir accès aux composantes réelle et imaginaire du nombre, il fera appel aux méthodes d'accès définies à l'étape 1. On pourrait également le déclarer comme « ami » (`friend`) de la classe, ce qui est une pratique courante pour cet opérateur.

4. Pour cette partie, seuls les quatre opérateurs ci-après sont nécessaires :

   ```
   class Complexe {
   public:
       // ...
       Complexe& operator+=(const Complexe&);
       Complexe& operator-=(const Complexe&);
       Complexe operator+(const Complexe&) const;
       // ...
   };

   ```

```
Complexe operator+(const double, const Complexe&);
```

Mais il est préférable à ce stade de définir également les soustractions correspondant aux additions :

```
class Complexe {
public:
    // ...
    Complexe& operator+=(const Complexe&);
    Complexe& operator-=(const Complexe&);
    Complexe operator+(const Complexe&) const;
    Complexe operator-(const Complexe&) const;
    // ...
};

Complexe operator+(const double, const Complexe&);
Complexe operator-(const double, const Complexe&);
```

Et pour le corps de ces opérateurs :

```
Complexe& Complexe::operator+=(const Complexe& c)
{
    re += c.re;
    im += c.im;
    return *this;
}
```

5. Puis :

```
class Complexe {
public:
    // ...
    Complexe operator*(const Complexe&) const;
    Complexe operator/(const Complexe&) const;
    Complexe& operator/=(const Complexe&);
    // ...
};
```

6. et :

```
class Complexe {
public:
    // ...
    Complexe& operator*=(const Complexe&);
    // ...
};

Complexe operator*(const double, const Complexe&);

/* Pas obligatoire, mais on preferera le definir egalement */
Complexe operator/(const double, const Complexe&);
```

Voici le code complet :

```
#include <cmath>
#include <iostream>

using namespace std;

class Complexe {
public:
    Complexe(const double re = 0.0, const double im = 0.0)
    : re(re), im(im)
    {}

```

```
    double reel() const { return re; }
    double imag() const { return im; }

    bool operator==(const Complexe&) const;

    Complexe& operator+=(const Complexe&);
    Complexe& operator-=(const Complexe&);
    Complexe& operator*=(const Complexe&);
    Complexe& operator/=(const Complexe&);

    Complexe operator+(const Complexe&) const;
    Complexe operator-(const Complexe&) const;
    Complexe operator*(const Complexe&) const;
    Complexe operator/(const Complexe&) const;

private:
    void set_reel(double x) { re = x; }
    void set_imag(double y) { im = y; }

    double re;
    double im;
};

ostream& operator<<(ostream&, const Complexe&);

Complexe operator+(const double, const Complexe&);
Complexe operator-(const double, const Complexe&);
Complexe operator*(const double, const Complexe&);
Complexe operator/(const double, const Complexe&);

int main()
{
    Complexe defaut;
    Complexe zero(0.0, 0.0);
    Complexe un(1.0, 0.0);
    Complexe i(0.0, 1.0);
    Complexe j;

    cout << zero << " ==? " << defaut;
    if (zero == defaut) cout << " oui" << endl;
    else cout << " non" << endl;

    cout << zero << " ==? " << i;
    if (zero == i) cout << " oui" << endl;
    else cout << " non" << endl;

    j = un+i;
    cout << un << " + " << i << " = " << j << endl;

    Complexe trois(un);
    trois += un;
    trois += 1.0;
    cout << un << " + " << un << " + 1.0 = " << trois << endl;

    Complexe deux(trois);
    deux -= un;
    cout << trois << " - " << un << " = " << deux << endl;

    trois = 1.0 + deux;
    cout << "1.0 + " << deux << " = " << trois << endl;

    Complexe z(i*i);
    cout << i << " * " << i << " = " << z << endl;
```

```
    cout << z << " / " << i << " = " << z/i << " = ";
    cout << (z/=i) << endl;

    Complexe k(2.0,-3.0);
    z = k;
    z *= 2.0;
    z *= i;
    cout << k << " * 2.0 * " << i << " = " << z << endl;
    z = 2.0 * k * i / 1.0;
    cout << " 2.0 * " << k << " * " << i << " / 1 = " << z << endl;

    return 0;
}

Complexe& Complexe::operator+=(const Complexe& c)
{
    set_reel( reel() + c.reel() );
    set_imag( imag() + c.imag() );
    return *this;
}

Complexe& Complexe::operator-=(const Complexe& c)
{
    set_reel( reel() - c.reel() );
    set_imag( imag() - c.imag() );
    return *this;
}

Complexe& Complexe::operator*=(const Complexe& c)
{
    // attention ici, comme reel() va etre modifie
    // par la premiere affectation, il faut preserver
    // la valeur d'origine pour le second calcul :
    const double anc_reel( reel() );

    set_reel( reel() * c.reel() - imag() * c.imag() );
    set_imag( anc_reel * c.imag() + imag() * c.reel() );

    return *this;
}

Complexe& Complexe::operator/=(const Complexe& c)
{
    const double anc_reel( reel() );
    const double r(c.reel()*c.reel() + c.imag()*c.imag());
    set_reel(( reel() * c.reel() + imag() * c.imag() ) / r);
    set_imag(( imag() * c.reel() - anc_reel * c.imag() ) / r);
    return *this;
}

bool Complexe::operator==(const Complexe& c) const
{
    return ((reel() == c.reel()) && (imag() == c.imag()));
}

Complexe Complexe::operator+(const Complexe& c) const
{
    return Complexe(*this) += c;
}

Complexe Complexe::operator-(const Complexe& c) const
{
    return Complexe(*this) -= c;
```

```
}

Complexe Complexe::operator*(const Complexe& c) const
{
    return Complexe(*this) *= c;
}

Complexe Complexe::operator/(const Complexe& c) const
{
    return Complexe(*this) /= c;
}

/* version minimaliste
ostream& operator<<(ostream& out, const Complexe& c)
{
    out << '(' << c.reel() << ',' << c.imag() << ')';
    return out;
}
*/

// version plus sophistiquee
ostream& operator<<(ostream& out, const Complexe& c)
{
    out << '(';
    if (c == Complexe(0.0, 0.0)) out << "0";
    else {
        if ( (c.reel() != 0.0) || ( (c.imag() != 1.0) && (c.imag() != -1.0) ) )
        {
            out << c.reel();
            if (c.imag() != 0.0) out << ',';
        }
        if (c.imag() != 0.0)
        {
            if (c.imag() == -1.0) out << "-i";
            else if (c.imag() == 1.0) out << "+i";
            else if (c.imag() > 0.0) out << "+" << c.imag() << "i";
            else out << c.imag() << "i";
        }
    }
    out << ')';
    return out;
}

// On utilise ici le plongement naturel des "double" dans "Complexe"
// offert par les constructeurs definis plus haut
Complexe operator+(const double x, const Complexe& c)
{
    return (Complexe(x) + c);
}

Complexe operator-(const double x, const Complexe& c)
{
    return (Complexe(x) - c);
}

Complexe operator*(const double x, const Complexe& c)
{
    return (Complexe(x) * c);
}

Complexe operator/(const double x, const Complexe& c)
{
    return (Complexe(x) / c);
```

```
}
```