MOOC Intro. POO C++
Corrigés semaine 3

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 9 : affichages

Par exemple pour les `Points3D` :

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

private:
  double x, y, z;
};

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

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

Exercice 10 : nombres complexes

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;
};
```

- Il faut au minimum le constructeur par défaut (on décide naturellement que le complexe (0,0) soit par défaut construit) 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 (plongement des nombres réels dans les nombres complexes), 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);
    ...
};
```

- 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.

- 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;
}
```

- puis :

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

- et

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

```
};

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

/* Pas obligatoire, mais on préférera le définir également */
Complexe operator/(const double, const Complexe&);
```

Et 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 être modifié
    // par la première affectation, il faut préserver
```

```
    // 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()) and (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 sophistiquée
ostream& operator<<(ostream& out, const Complexe& c)
{
    out << '(';
    if (c == Complexe(0.0, 0.0)) out << "0";
    else {
        if ( (c.reel() != 0.0) or
```

```
              ( (c.imag() != 1.0) and (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();
            else out << c.imag();
        }
    }
    out << ')';
    return out;
}

// On utilise ici le plongement naturel des "double"
// dans "Complexe" offert par les constructeurs définis
// 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);
}
```

Exercice 11 : toujours plus de polynômes

> Cet exercice correspond à l'exercice n°54 (pages 131 et 314) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Cet exercice ne présente aucune difficulté. Il est niveau 2 uniquement en raison de sa longueur.

Pour la première partie, voir le tutoriel.

Concernant la seconde partie :

1. *Ajoutez les opérateurs pour l'addition et la soustraction* :

   À ajouter à la classe `Polynome` :

   ```
   Polynome& operator+=(const Polynome&);
   Polynome& operator-=(const Polynome&);
   Polynome operator+(const Polynome&) const;
   Polynome operator-(const Polynome&) const;
   ```

   Les opérateurs + et − sont effectivement `const` puisqu'il ne modifient pas la classe concernée mais crée un nouveau résultat.

   Pour la définition des opérateurs, écrire hors de la classe :

   ```
   Polynome& Polynome::operator+=(const Polynome& q) {
     // garantit que le degré de p (résultat) est au moins le degré de b
     while (degre() < q.degre()) p.push_back(0);

     for (Degre i(0); i <= q.degre(); ++i) p[i] += q.p[i];

     return *this;
   }

   Polynome& Polynome::operator-=(const Polynome& q)
   {
     // garantit que le degré de p (résultat) est au moins le degré de b
     while (degre() < q.degre()) p.push_back(0);

     for (Degre i(0); i <= q.degre(); ++i) p[i] -= q.p[i];

     // garantit la bonne forme du polynome
     simplifie();

     return *this;
   }
   ```

   Ce code est directement adapté du cours 13.

   Pour la fonction `simplifie` elle devient maintenant une méthode privée :

   ```
   void Polynome::simplifie()
   {
     while ((not p.empty()) and (top() == 0.0)) p.pop_back();
     if (p.empty()) p.push_back(0.0);
   ```

```
}
```

2. *Ajoutez les opérateurs de comparaison == et !=*

```
bool Polynome::operator==(const Polynome& q) const {
  return p == q.p;
}

bool Polynome::operator!=(const Polynome& q) const {
  return p != q.p;
}
```

3. *Ajoutez une méthode `top` qui retourne la valeur du coefficient de plus haut degré du polynôme.*

```
double top() const { return p[degre()]; }
```

4. *Ajoutez maintenant la méthode (publique) `divise()`*

   Il suffit simplement de réécrire l'ancienne fonction `division` avec les nouveaux opérateurs :

```
void Polynome::divise(const Polynome& denominateur,
        Polynome& quotient, Polynome& reste) const
{
  quotient = 0;
  reste = *this;
  for (Degre dq(reste.degre() - denominateur.degre());
       (dq >= 0) // il est important ici que dq soit signé
  and (reste != 0);
       dq = reste.degre() - denominateur.degre()) {
    quotient.met_coef(reste.top() / denominateur.top(), dq);
    reste -= Polynome(dq, quotient.p[dq]) * denominateur;
  }
}
```

5. *Terminer en implémentant les opérateurs de division et modulo*

   Aucune surprise ici : prototypes :

```
Polynome& operator/=(const Polynome&);
Polynome operator/(const Polynome&) const;
Polynome operator%(const Polynome&) const;
```

   et définitions :

```
Polynome Polynome::operator/(const Polynome& q) const {
  Polynome r,s;
  divise(q,r,s);
  return r;
}

Polynome Polynome::operator%(const Polynome& q) const {
  Polynome r,s;
  divise(q,r,s);
  return s;
}
```

```
Polynome& Polynome::operator/=(const Polynome& q) {
  return *this = *this / q;
}
```

Voici le code complet :

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

typedef unsigned int Degre;

class Polynome {
public:
  // constructeurs
  Polynome(double coef = 0.0, int degre = 0); // monome coef * X^deg

  // methodes publiques
  Degre degre() const { return p.size()-1; }
  void affiche_coef(ostream& out, Degre puissance,
                    bool signe = true)              const;
  void divise(const Polynome& denominateur,
       Polynome& quotient, Polynome& reste) const;
  double top() const { return p[degre()]; }

  // operateurs internes
  bool operator==(const Polynome&) const;
  bool operator!=(const Polynome&) const;
  Polynome& operator+=(const Polynome&);
  Polynome& operator-=(const Polynome&);
  Polynome& operator*=(const Polynome&);
  Polynome& operator*=(const double);
  Polynome& operator/=(const Polynome&);
  Polynome& operator/=(const double);
  Polynome operator+(const Polynome&) const;
  Polynome operator-(const Polynome&) const;
  Polynome operator-() const;
  Polynome operator*(const double) const;
  Polynome operator*(const Polynome&) const;
  Polynome operator/(const Polynome&) const;
  Polynome operator/(const double) const;
  Polynome operator%(const Polynome&) const;

private:
  // attributs privés
  vector<double> p;

  // méthodes privées
  void simplifie();
  void met_coef(const double c, const Degre d);
};

// opérateur externe
Polynome operator*(const double x, const Polynome& p);

const Polynome X(1.0,1);
```

```
const Polynome X2(1.0,2);
const Polynome X3(1.0,3);
const Polynome X4(1.0,4);
const Polynome X5(1.0,5);

void affiche_res(const Polynome& p, const Polynome& q,
   const Polynome& r, const Polynome& s);

// ====================================================================
int main()
{
  Polynome p(5*X3 + 4*X2 - 2*X + 3);
  Polynome q(-2*X2 + 2*X + 1);
  Polynome r,s;

  p.divise(q, r, s);
  affiche_res(p,q,r,s);

  // ------------------------------------------------------------
  q = p;
  p.divise(q, r, s);
  affiche_res(p,q,r,s);

  // ------------------------------------------------------------
  q = 3;
  p.divise(q, r, s);
  affiche_res(p,q,r,s);

  // ------------------------------------------------------------
  q = 6*X4 + 5*X3 + 4*X2 - 2*X + 3;

  p.divise(q, r, s);
  affiche_res(p,q,r,s);

  // ------------------------------------------------------------
  q = Polynome(5.0,3);

  p.divise(q, r, s);
  affiche_res(p,q,r,s);

  return 0;
}

/* ********************************************************************** *
 * ********************************************************************** */
void Polynome::affiche_coef(ostream& out, Degre puissance,
                            bool signe) const
{
  double const c(p[puissance]);
  if (c != 0.0) {
    if (signe and (c > 0.0))  out << "+";
    out << c;
    if (puissance > 1)
      out << "*X^" << puissance;
    else if (puissance == 1) out << "*X";
  } else if (degre() == 0) {
    // degré 0 : afficher quand meme le 0 si rien d'autre
    out << 0;
```

```
  }
}

/* ************************************************************************ *
 * ************************************************************************ */
void Polynome::simplifie()
{
  while ((!p.empty()) and (top() == 0.0)) p.pop_back();
  if (p.empty()) p.push_back(0.0);
}

/* ************************************************************************ *
 * ************************************************************************ */
ostream& operator<<(ostream& sortie, const Polynome& polynome)
{
  // plus haut degré : pas de signe + devant
  Degre i(polynome.degre());
  polynome.affiche_coef(sortie, i, false);

  // degré de N à 0 : +a*X^i
  if (i > 0) {
    for (i--; i > 0; i--) polynome.affiche_coef(sortie, i);
    polynome.affiche_coef(sortie, 0);
  }

  return sortie;
}

/* ************************************************************************ *
 * ************************************************************************ */
void Polynome::met_coef(const double c, const Degre d)
{
  // garantit que le degré de p est au moins d
  while (degre() < d) p.push_back(0.0);
  p[d] = c;
}

/* ************************************************************************ *
 * ************************************************************************ */
void Polynome::divise(const Polynome& denominateur,
        Polynome& quotient, Polynome& reste) const
{
  quotient = 0;
  reste = *this;
  for (int dq(reste.degre() - denominateur.degre());
       (dq >= 0) // il est important ici que dq soit signé
  and (reste != 0);
       dq = reste.degre() - denominateur.degre()) {
    quotient.met_coef(reste.top() / denominateur.top(), dq);
    reste -= Polynome(quotient.p[dq], dq) * denominateur;
  }
}


/* ************************************************************************ *
 * ************************************************************************ */
void affiche_res(const Polynome& p, const Polynome& q,
   const Polynome& r, const Polynome& s)
```

```
{
  cout << "La division de " << p << " par " << q << endl;
  cout << "  a pour quotient " << r << endl;
  cout << "  et pour reste " << s << endl;
}


/* ********************************************************************** *
 * ********************************************************************** */
Polynome::Polynome(const double coef, const int deg)
: p(deg+1) // garantit que le degré de p est au moins deg
 {
   p[deg] = coef;
}

/* ********************************************************************** *
 * ********************************************************************** */
bool Polynome::operator==(const Polynome& q) const {
  return p == q.p;
}

/* ********************************************************************** *
 * ********************************************************************** */
bool Polynome::operator!=(const Polynome& q) const {
  return p != q.p;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator+=(const Polynome& q) {
  // garantit que le degré de p (résultat) est au moins le degré de b
  while (degre() < q.degre()) p.push_back(0);

  for (Degre i(0); i <= q.degre(); i++) p[i] += q.p[i];

  return *this;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator-=(const Polynome& q)
{
  // garantit que le degré de p (résultat) est au moins le degré de b
  while (degre() < q.degre()) p.push_back(0);

  for (Degre i(0); i <= q.degre(); ++i) p[i] -= q.p[i];

  // garantit la bonne forme du polynome
  simplifie();

  return *this;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator*=(const Polynome& q) {
  return *this = *this * q;
}
```

```
/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator*=(const double x) {
  for (Degre i(0); i <= degre(); i++)
    p[i] *= x;
  return *this;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator/=(const double x) {
  for (Degre i(0); i <= degre(); i++)
    p[i] /= x;
  return *this;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome& Polynome::operator/=(const Polynome& q) {
  return *this = *this / q;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator+(const Polynome& q) const
{ return Polynome(*this) += q; }

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator-(const Polynome& q) const
{ return Polynome(*this) -= q; }

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator-() const {
  Polynome q;
  for (Degre i(0); i <= degre(); ++i)
    q.p.push_back(-p[i]);
  return q;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator*(const double x) const
{ return Polynome(*this) *= x; }

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator*(const Polynome& q) const {
  Polynome r;
  // notice that r = 0 here, i.e. already contains one 0
  // therfore degre()+q.degre()-1 push_backs are enough
  // instead of degre()+q.degre()
  for (Degre i(0); i < degre() + q.degre(); i++) r.p.push_back(0.0);
  for (Degre i(0); i <= degre(); i++)
    for (Degre j(0); j <= q.degre(); j++)
      r.p[i+j] += p[i] * q.p[j];
```

```
  return r;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator/(const Polynome& q) const {
  Polynome r,s;
  divise(q,r,s);
  return r;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator/(const double x) const
{ return Polynome(*this) /= x; }

/* ********************************************************************** *
 * ********************************************************************** */
Polynome Polynome::operator%(const Polynome& q) const {
  Polynome r,s;
  divise(q,r,s);
  return s;
}

/* ********************************************************************** *
 * ********************************************************************** */
Polynome operator*(const double x, const Polynome& p) {
  return Polynome(p) *= x;
}
```
