# MOOC Intro. POO C++
# Corrigés semaine 1

---

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 1 : la classe `Cercle`**

> Cet exercice correspond à l'exercice n°46 (pages 109 et 291) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Cet exercice se fait de façon assez similaire au tutoriel sur les rectangles.

Comment commencer ? À ce niveau, il suffit de suivre l'enoncé :

- « *définissez une classe* `Cercle` » :

```
class Cercle {
};
```

- « *ayant comme attributs privés* »

```
class Cercle {
private:
};
```

- « *le rayon du cercle (de type double), et les coordonnées de son centre.* »

```
class Cercle {
private:
    double rayon;
    double x;  // abscisse du centre
    double y;  // ordonnée du centre
};
```

**Remarque :** les meilleurs analystes peuvent faire remaquer que le centre est un «point» et écrire, de façon encore plus propre, le code suivant :

```
struct Point {
    double x;  // abscisse
    double y;  // ordonnée
};

class Cercle {
private:
    double rayon;
    Point centre;
};
```

**Remarque 2 :** les meilleurs analystes et programmeurs « objet » en feront bien sûr une classe plutôt qu'une struct...
...avec une encapsulation correcte (voir l'exercice 3).

On continue ensuite à suivre l'énoncé à la lettre : *Déclarez ensuite les méthodes «get» et «set» correspondantes* :

```
class Cercle {
```

```
  void getCentre(double& x_out, double& y_out) const {
    x_out = x;
    y_out = y;
  }
  void setCentre(double x_in, double y_in) {
    x = x_in;
    y = y_in;
  }
  double getRayon() const {
    return rayon;
  }
  void setRayon(double r) {
    if (r < 0.0) r = 0.0;
    rayon = r;
  }
private:
    double rayon;
    double x;  // abscisse du centre
    double y;  // ordonnée du centre
};
```

Attention à bien faire la différence entre `x` en tant qu'argument de la méthode et `x` en tant qu'attribut de l'instance. Dans les cas ambigus comme ci-dessus, il faut lever l'ambiguité en utilisant le pointeur `this` (pour indiquer l'attribut), ou alors changer de noms.

Ah! QUESTION : ces méthodes sont-elles privées ou publiques ?

Publiques évidemment car on doit pouvoir les utiliser hors de la classe (elles font partie de l'interface).

```
class Cercle {
public:
    void getCentre(double &x, double &y) const {
    ... // comme avant
};
```

**Remarque 3 :** les meilleurs analystes continueraient ici sur leur lancée et choisiraient les prototypes suivants pour `getCentre` et `setCentre` :

```
    Point getCentre() const { return centre; }
    void setCentre(const Point& p) { centre = p; }
```

On termine ensuite notre classe de façon très similaire à ce qui précède :

```
#include <cmath> // pour M_PI

class Cercle {
public:
    double surface() const { return M_PI * rayon * rayon; }
    bool estInterieur(double x1, double y1) const {
        return (((x1-x) * (x1-x) + (y1-y) * (y1-y))
                <= rayon * rayon);
    }
    void getCentre(double& x, double& y) const {
    ... // suite comme avant
```

Il ne reste plus qu'à tester :

```
#include <iostream> // pour cout et endl
#include <cmath> // pour M_PI et sqrt()
using namespace std;

// ... la classe Cercle comme avant
```

```
int main () {
    Cercle c1, c2;

    c1.setCentre(1.0, 2.0);
    c1.setRayon(sqrt(5.0)); // passe par (0, 0)
    c2.setCentre(-2.0, 1.0);
    c2.setRayon(2.25);  // 2.25 > sqrt(5) => inclus le point (0, 0)

    cout << "Surface de C1 : " << c1.surface() << endl;
    cout << "Surface de C2 : " << c2.surface() << endl;

    cout << "position du point (0, 0) : ";
    if (c1.estInterieur(0.0, 0.0)) cout << "dans";
    else                           cout << "hors de";
    cout << " C1 et ";
    if (c2.estInterieur(0.0, 0.0)) cout << "dans";
    else                           cout << "hors de";
    cout << " C2." << endl;

    return 0;
}
```

On pourrait même faire une fonction pour tester les points :

```
void testPoint(double x, double y, Cercle c1, Cercle c2) {
    cout << "position du point (" << x << ", " << y << ") : ";
    if (c1.estInterieur(x, y)) cout << "dans";
    else                       cout << "hors de";
    cout << " C1 et ";
    if (c2.estInterieur(x, y)) cout << "dans";
    else                       cout << "hors de";
    cout << " C2." << endl;
}

int main () {
    ... // comme avant

    testePoint(0.0, 0.0, c1, c2);
    testePoint(1.0, 1.0, c1, c2);
    testePoint(2.0, 2.0, c1, c2);

    return 0;
}
```

## Solutions finales

### Version suffisante

```
#include <iostream>
#include <cmath>    // pour M_PI et sqrt()

using namespace std;

class Cercle {
public:
    double surface() const { return M_PI * rayon * rayon; }
    bool estInterieur(double x1, double y1) const {
        return (((x1-x) * (x1-x) + (y1-y) * (y1-y))
                <= rayon * rayon);
    }
    void getCentre(double& x_out, double& y_out) const {
```

```
        x_out = x;
        y_out = y;
    }
    void setCentre(double x_in, double y_in) {
       x = x_in;
       y = y_in;
    }
    double getRayon() const { return rayon; }
    void setRayon(double r) {
       if (r < 0.0) r = 0.0;
       rayon = r;
    }
private:
    double rayon;
    double x;  // abscisse du centre
    double y;  // ordonnée du centre
};

int main () {
    Cercle c1, c2;

    c1.setCentre(1.0, 2.0);
    c1.setRayon(sqrt(5.0)); // passe par (0, 0)
    c2.setCentre(-2.0, 1.0);
    c2.setRayon(2.25);  // 2.25 > sqrt(5) => inclus le point (0, 0)

    cout << "Surface de C1 : " << c1.surface() << endl;
    cout << "Surface de C2 : " << c2.surface() << endl;

    cout << "position du point (0, 0) : ";
    if (c1.estInterieur(0.0, 0.0)) cout << "dans";
    else                           cout << "hors de";
    cout << " C1 et ";
    if (c2.estInterieur(0.0, 0.0)) cout << "dans";
    else                           cout << "hors de";
    cout << " C2." << endl;

    return 0;
}
```

**Version perfectionnée**

```
#include <iostream> // pour cout et endl
#include <cmath> // pour M_PI et sqrt()
using namespace std;

// ------------ un point dans le plan ------------
struct Point {
    double x;  // abscisse
    double y;  // ordonnée
};

// ------------ la classe Cercle ------------
class Cercle {
// --- interface ---
public:
    double surface() const { return M_PI * rayon * rayon; }

   bool estInterieur(const Point& p) const {
        return (((p.x-centre.x) * (p.x-centre.x) +
                 (p.y-centre.y) * (p.y-centre.y))
                <= rayon * rayon);
    }
```

```
    // interface des attributs
    Point getCentre() const { return centre; }
    void setCentre(const Point& p) { centre = p; }
    double getRayon() const { return rayon; }
    void setRayon(double r) {
       if (r < 0.0) r = 0.0;
       rayon = r;
    }
// --- -------- ---
private:
    double rayon;
    Point centre;
};

// ----------------------------------------------------------------------
void dans(bool oui) {
  if (oui) cout << "dans"; else cout << "hors de";
  /* Note : peut aussi s'écrire :           *
   *  cout << (oui ? "dans" : "hors de");   */
}

// ----------------------------------------------------------------------
void test(Point p, Cercle c1, Cercle c2) {
    cout << "position du point (" << p.x << ", " << p.y << ") : ";
    dans(c1.estInterieur(p));
    cout << " C1 et ";
    dans(c2.estInterieur(p));
    cout << " C2." << endl;
}

// ----------------------------------------------------------------------
int main () {
    Cercle c1, c2;
    Point p;

    p.x = 1.0; p.y = 2.0;
    c1.setCentre(p);
    c1.setRayon(sqrt(5.0)); // passe par (0, 0)

    p.x = -2.0; p.y = 1.0;
    c2.setCentre(p);
    c2.setRayon(2.25);  // 2.25 > sqrt(5) => inclus le point (0, 0)

    cout << "Surface de C1 : " << c1.surface() << endl;
    cout << "Surface de C2 : " << c2.surface() << endl;

    p.x=0.0; p.y=0.0;
    test(p,c1,c2);

    p.x=1.0; p.y=1.0;
    test(p,c1,c2);

    return 0;
}
```

**Exercice 2 : petit tour de magie**

> Cet exercice correspond à l'exercice n°48 (pages 111 et 297) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Voici une solution possible (lisez les commentaires) :

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

// Un bout de papier... pour ce tour de magie
class Papier {
public:
  void ecrire(unsigned int un_age, unsigned int de_l_argent) {
    age    = un_age;
    argent = de_l_argent;
  }

  unsigned int lire_age()   const { return age   ; }
  unsigned int lire_somme() const { return argent; }

private:
  /* Ces 2 attributs spécifiquement, car d'une façon ou d'une autre
   * le spectateur rend intelligible l'information contenue sur le papier.
   */
  unsigned int age;
  unsigned int argent;
};

// -------------------------------------
class Assistant {
public:
  void lire(const Papier& billet);
  void calculer();
  unsigned int annoncer();

private:
  /* l'assistant mémorise dans son cerveau les valeurs lues
   * et le resultat du calcul.
   */
  unsigned int age_lu;
  unsigned int argent_lu;
  unsigned int resultat;
};


// -------------------------------------
class Spectateur {
public:
  void arriver();  /* lorsqu'il entre dans la salle (avant
                    *    il n'"existe" pas pour nous)
                    */
  void ecrire();    // écrit sur le papier
  Papier montrer(); // montre le papier

private:
  // ses spécificités
  unsigned int age;
  unsigned int argent;

  /* Dans cette version nous faisons l'hypothèse que
   * c'est le spectateur qui a un papier.
```

```
   * Dans d'autres modélisations, ce papier pourrait aussi bien
   * appartenir au magicien (variable locale à Magicien::tourDeMagie),
   * à l'assistant ou même "être dans la salle" (i.e. variable du main)
   */
  Papier paquet_cigarettes;
};


// ---------------------------------
class Magicien {
public:
  void tourDeMagie(Assistant& asst, Spectateur& spect);
  /* Pour faire son tour, le magicien a besoin d'au moins
   * un spectateur et d'un assistant.
   */

private:
  /* partie privée ici car seul le magicien sait ce qu'il doit
   * faire dans son tour.
   */
  void calculer(unsigned int resultat_recu);
  void annoncer();

  unsigned int age_devine;
  unsigned int argent_devine;
};

// ===========================================================================
int main()
{
  // L'histoire générale :
  Spectateur thorin;    // Il était une fois un spectateur...
  thorin.arriver();     // ...qui venait voir un spectacle (!!)...

  Magicien gandalf;                   // ...où un magicien...
  Assistant bilbo;                    // ...et son assistant...
  gandalf.tourDeMagie(bilbo, thorin); // ...lui firent un tour fantastique.

  return 0;
}

// ---------------------------------------------------------------------------
void Assistant::lire(const Papier& billet)
{
  cout << "[Assistant] (je lis le papier)" << endl;
  age_lu    = billet.lire_age();
  argent_lu = billet.lire_somme();
}

void Assistant::calculer()
{
  cout << "[Assistant] (je calcule mentalement)" << endl;
  resultat = age_lu * 2;
  resultat += 5;
  resultat *= 50;
  resultat += argent_lu;
  resultat -= 365;
}

unsigned int Assistant::annoncer()
{
  cout << "[Assistant] J'annonce : " << resultat << " !" << endl;
  return resultat;
}
```

```
// ---------------------------------------------------------------------------
void Spectateur::arriver()
{
  cout << "[Spectateur] (j'entre en scène)" << endl;
  cout << "Quel âge ai-je ? ";
  cin >> age;
  do {
    cout << "Combien d'argent ai-je en poche (<100) ? ";
    cin >> argent;
  } while (argent >= 100);
  cout << "[Spectateur] (je suis là)" << endl;
}

void Spectateur::ecrire()
{
  cout << "[Spectateur] (j'écris le papier)" << endl;
  paquet_cigarettes.ecrire(age, argent);
}

Papier Spectateur::montrer()
{
  cout << "[Spectateur] (je montre le papier)" << endl;
  return paquet_cigarettes;
}

// ---------------------------------------------------------------------------
void Magicien::tourDeMagie(Assistant& fidele, Spectateur& quidam)
{
  cout << "[Magicien] un petit tour de magie..." << endl;
  // le magicien donne ses instructions :
  quidam.ecrire();
  fidele.lire(quidam.montrer());
  fidele.calculer();
  calculer(fidele.annoncer());
  annoncer();
}

void Magicien::calculer(unsigned int resultat_recu) {
  resultat_recu += 115;
  age_devine    = resultat_recu / 100;
  argent_devine = resultat_recu % 100;
}

void Magicien::annoncer() {
  cout << "[Magicien] " << endl
       << "  - hum... je vois que vous êtes agé de " << age_devine << " ans" << endl
       << "    et que vous avez " << argent_devine << " francs en poche !" << endl;
}
```

**NOTE :** La méthode `arriver` du spectateur correspond en fait à son initialisation. Nous verrons la semaine prochaine qu'il existe des méthodes spécifiques pour cela : les constructeurs.

---

**Exercice 3 : coordonnées 3D**

> Cet exercice correspond à l'exercice n°47 (pages 110 et 294) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Définissons la classe `Point3D`, dont le nom et la structure sont en fait imposés par l'énoncé au vu des lignes :

```
point1.init(1.0, 2.0, -0.1);
point2.init(2.6, 3.5,  4.1);
```

Cette classe doit contenir trois attributs, disons `x`, `y`, et `z`, lesquels doivent être initialisés par une méthode `init`.

Cela conduit donc à :

```
class Point3D
{
public:
  void init(double x0, double y0, double z0) {
    x = x0; y = y0; z = z0;
  }
private:
  double x, y, z;
};
```

L'énoncé nous demande d'ajouter les méthodes `affiche` et `compare`. Quel est leur prototype ?

Rappelons que les attributs d'une classe sont accessibles aux méthodes de cette classe et n'ont donc pas besoin d'être passés en paramètres.

Ainsi, `affiche` n'a besoin d'aucun argument et `compare` ne nécessite que de recevoir le point avec lequel il faut se comparer.

On arrive donc à :

```
class Point3D
{
public:
  bool compare(const Point3D& autre) const;
  void affiche() const;
  // ... (comme avant)
};
```

**NOTES :**

- Ces deux méthodes ne modifiant pas les instances de la classe (elles ne modifient pas les attributs `x`, `y` ni `z`), on l'indique dans leur prototype en les terminant par « `const` ».
- On peut optimiser légèrement les appels à la méthode `compare` en évitant la copie de son argument en le passant par référence constante : `bool compare(const Point3D& autre) const;`

  C'est cette version qui est gardée pour la suite, mais si l'on n'est pas à l'aise avec le passage par référence constante, il vaut mieux à ce stade garder le passage par valeur précédent.

Il ne reste plus qu'à définir ces deux méthodes. Elles sont ici définies hors de la classe pour illustrer comment cela se fait.

Hors de la classe le nom de la méthode doit être précédé du nom de la classe suivi de l'opérateur de résolution de portée `::` :
`bool Point3D::compare(const Point3D& autre) const {}`

Pour la méthode `compare`, il faut retourner la valeur booléenne « vrai » lorsque les trois coordonnées correspondent. Cela peut se faire simplement avec la formule logique suivante (même s'il n'est pas recommandé de comparer des valeurs `double` avec ==, mais ce n'est pas le propos ici) :

```
bool Point3D::compare(const Point3D& autre) const
{
  return (autre.x == x) and (autre.y == y) and (autre.z == z);
}
```

La méthode `affiche` ne devrait pas poser de difficulté particulière. Nous aboutissons alors à la solution complète suivante :

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

class Point3D
{
public:
  bool compare(const Point3D& autre) const;
  void affiche() const;
  void init(double x0, double y0, double z0) {
    x = x0; y = y0; z = z0;
  }
private:
  double x, y, z;
};

bool Point3D::compare(const Point3D& autre) const
{
  return (autre.x == x) and (autre.y == y) and (autre.z == z);
}

void Point3D::affiche() const
{
  cout << '(' << x << ", " << y << ", " << z << ')' << endl;
}

int main() {
  Point3D point1;
  Point3D point2;
  Point3D point3;

  point1.init(1.0, 2.0, -0.1);
  point2.init(2.6, 3.5,  4.1);
  point3 = point1;

  cout << "Point 1 :";
  point1.affiche();

  cout << "Point 2 :";
  point2.affiche();

  cout << "Le point 1 est ";
  if (point1.compare(point2)) {
    cout << "identique au";
  } else {
    cout << "différent du";
  }
  cout << " point 2." << endl;

  cout << "Le point 1 est ";
  if (point1.compare(point3)) {
    cout << "identique au";
  } else {
    cout << "différent du";
  }
  cout << " point 3." << endl;
```

```
  return 0;
}
```

---

**Exercice 4 : triangles**

Voici une solution possible (voir les commentaires) :

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

class Point3D { // repris de l'exercice précédent
public:
  void init(double x0, double y0, double z0) {
    x = x0; y = y0; z = z0;
  }
  void affiche() const {
    cout << '(' << x << ", " << y << ", " << z << ')' << endl;
  }
  double getx() const { return x; }
  double gety() const { return y; }
  double getz() const { return z; }

  void init() {
    cout << "Construction d'un nouveau point" << endl;
    cout << "   Veuillez entrer x : ";
    cin >> x;
    cout << "   Veuillez entrer y : ";
    cin >> y;
    cout << "   Veuillez entrer z : ";
    cin >> z;
  }

private:
  double x, y, z;
};

double distance(Point3D const& p1, Point3D const& p2) {
  return sqrt(   (p1.getx() - p2.getx()) * (p1.getx() - p2.getx())
               + (p1.gety() - p2.gety()) * (p1.gety() - p2.gety())
               + (p1.getz() - p2.getz()) * (p1.getz() - p2.getz())
             );
}

class Triangle {
public:
  void init() {
    array<Point3D, 3> sommets; // Les 3 sommets du triangle.
    for (auto& p : sommets) { p.init(); }

    /* On pourrait aussi l'écrire de façon plus compacte
     * avec une boucle for et l'opérateur modulo, mais j'ai
     * préféré ici expliciter ces formules.
     */
    cotes[0] = distance(sommets[0], sommets[1]);
    cotes[1] = distance(sommets[1], sommets[2]);
    cotes[2] = distance(sommets[2], sommets[0]);
  }

  bool est_isocele() const {
    /* Voici une version possible.
     * Si vous n'aimez pas cette forme logique compacte,
     * vous pouvez aussi l'écrire avec une série de if-else.
     */
    return    (cotes[0] == cotes[1])
           or (cotes[1] == cotes[2])
```

```
            or (cotes[2] == cotes[0]) ;
  }

  double perimetre() const {
    /* Ici aussi, on pourrait aussi l'écrire de façon plus compacte
     * avec une boucle for, mais j'ai également préféré expliciter la
     * formule.
     */
    return cotes[0] + cotes[1] + cotes[2];
  }

private:
  /* On pourrait aussi représenter un triangle comme 3 points :
   *    array<Point3D, 3> points;
   * mais cela n'est pas très utile dans cet exercices qui
   * finalement n'utilise que les longueurs des cotés.
   */
  array<double, 3> cotes;
};

int main() {
  Triangle t;
  t.init();
  cout << "Périmètre : " << t.perimetre() << endl;
  cout << "Ce triangle ";
  if (t.est_isocele()) {
    cout << "est";
  } else {
    cout << "n'est pas";
  }
  cout << " isocèle !" << endl;
  return 0;
}
```

---