MOOC Intro. POO C++
Corrigés semaine 4

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 12 : véhicules

> Cet exercice correspond à l'exercice n°55 (pages 139 et 315) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Pour la classe `Vehicule` :

```
class Vehicule {
protected:
  string       marque       ;
  unsigned int date_achat   ;
  double       prix_achat   ;
  double       prix_courant ;
};
```

Ces attributs sont «`protected`» car on ne souhaite pas pouvoir y accéder en dehors de la classe, mais on imagine tout de même pouvoir y accéder depuis «les héritiers».

Pour le constructeur, voici une solution possible :

```
class Vehicule {
public:
  Vehicule(string marque, unsigned int date, double prix)
  : marque(marque), date_achat(date), prix_achat(prix),
    prix_courant(prix)
  {}
  //...suite comme avant
```

Pour la méthode `affiche` :

```
class Vehicule {
public:
  // ...
  void affiche(ostream& affichage) const;
  // ...
};
void Vehicule::affiche(ostream& affichage) const {
  affichage << "marque : "         << marque
            << ", date d'achat : " << date_achat
            << ", prix d'achat : " << prix_achat
            << ", prix actuel : "  << prix_courant
            << endl;
}
```

Cette méthode est `const` car elle ne modifie pas l'état de l'objet.

NOTE : Il serait préférable ici de surcharger l'opérateur externe `ostream& operator<<(ostream&, const Vehicule&)` mais ce n'est pas le sujet de cet exercice.

Dans la dirrection opposée, on pourrait aussi déclarer `affiche` sans lui passer de paramètre et la faire directement opérer sur `cout`.

Pour la classe `Voiture`, elle doit hériter de la classe `Vehicule` ; on ajoute ensuite ses champs spécifiques :

```
class Voiture : public Vehicule {
protected:
  double       cylindree   ;
  unsigned int nb_portes   ;
  double       puissance   ;
  unsigned int kilometrage ;
};
```

On procéde de même pour la classe `Avion` :

```
enum Type_Avion { HELICES, REACTION };

class Avion : public Vehicule {
protected:
  Type_Avion   moteur     ;
  unsigned int heures_vol ;
}
```

Pour les constructeurs et méthodes d'affichage :

```
class Voiture : public Vehicule {
public:
  Voiture(string marque, unsigned int date, double prix,
          double cylindree, unsigned int portes, double cv, unsigned int km);
  void affiche(ostream&) const;
protected:
  double       cylindree   ;
  unsigned int nb_portes   ;
  double       puissance   ;
  unsigned int kilometrage ;
};
```

Ces deux méthodes doivent bien entendu être publiques puisqu'elles sont précisément faites pour être utilisées hors de la classe. Voici un exemple possible pour leur définition :

```
Voiture::Voiture(string marque, unsigned int date, double prix,
                 double cylindree, unsigned int portes, double cv,
                 unsigned int km)
  : Vehicule(marque, date, prix),  cylindree(cylindree),
    nb_portes(portes), puissance(cv),  kilometrage(km)
{}

void Voiture::affiche(ostream& affichage) const {
  affichage << " ---- Voiture ----" << endl;
  Vehicule::affiche(affichage);
  affichage << cylindree << " litres, "
            << nb_portes << " portes, "
            << puissance << " CV, "
            << kilometrage << " km." << endl;
```

```
}
```

Notons que pour le constructeur de `Voiture`, on fait appel au constructeur de `Vehicule`, et que `affiche` appelle la méthode de même nom de la classe `Vehicule` en la « démasquant » à l'aide de l'opérateur « `::` ».

Les méthodes de la classe `Avion` s'implémentent de même :

```
class Avion : public Vehicule {
public:
  Avion(string marque, unsigned int date, double prix,
        Type_Avion moteur, unsigned int heures);
  void affiche(ostream&) const;

protected:
  Type_Avion   moteur     ;
  unsigned int heures_vol ;
};

Avion::Avion(string marque, unsigned int date, double prix,
             Type_Avion moteur, unsigned int heures)
  : Vehicule(marque, date, prix), moteur(moteur), heures_vol(heures)
{}

void Avion::affiche(ostream& affichage) const {
  affichage << " ---- Avion à ";
  if (moteur == HELICES) affichage << "hélices";
  else                   affichage << "réaction";
  affichage << " ----" << endl;
  Vehicule::affiche(affichage);
  affichage << heures_vol << " heures de vol." << endl;
}
```

et pour `calculePrix()` (exemple pour l'année 2015):

```
class Vehicule {
public:
  Vehicule(string marque, unsigned int date, double prix);
  void affiche(ostream&) const;
  void calculePrix();
...
};
...
void Vehicule::calculePrix() {
  double decote((2015 - date_achat) * .01);
  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}
```

Le prototype est le même pour `Voiture` et `Avion` que pour `Vehicule`, par contre les définitions diffèrent :

```
void Voiture::calculePrix() {
  double decote((2015 - date_achat) * .02);
  decote += 0.05 * (kilometrage / 10000);
  if (marque == "Fiat" or marque == "Renault")
    decote += 0.1;
  else if (marque == "Ferrari" or marque == "Porsche")
    decote -= 0.2;
```

```
  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}

...

void Avion::calculePrix() {
  double decote;
  if (moteur == HELICES)
    decote = 0.1 * heures_vol / 100.0;
  else
    decote = 0.1 * heures_vol / 1000.0;

  prix_courant = max(0.0, (1.0 - decote) * prix_achat);
}
```

Exercice 13 : vecteurs 3D

> Cet exercice correspond à l'exercice n°56 (pages 141 et 318) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Nous commençons donc par reprendre la classe `Point3D` de l'exercice 3 et à la revoir à la lumière des nouvelles connaissances (constructeurs et surcharge d'opérateurs).

Ceci conduit à :

- mettre les attributs `x`, `y` et `z` « `protected` », vu l'utilisation que l'on souhaite en faire ici (héritage par `Vecteur`) ;
- introduire le(s) constructeur(s) correspondant(s) ;
- remplacer la méthode `affiche` par l'opérateur externe << ;
- remplacer la méthode `compare` par l'opérateur ==.

On aboutit alors au code suivant :

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

// ----------------------------------------------------------------------
class Point3D
{
public:
  // constructeur par défaut
  Point3D() : x(0.0), y(0.0), z(0.0) {}
  // constructeur par 3 coordonnees
  Point3D(double x, double y, double z) : x(x), y(y), z(z) {}

  // ex compare()
  bool operator==(const Point3D& autre) const;

  // methode d'acces en lecture
  double get_x() const { return x; }
  double get_y() const { return y; }
  double get_z() const { return z; }

protected:
  double x, y, z;
};

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

// ----------------------------------------------------------------------
ostream& operator<<(ostream& flot, const Point3D& p)
{
  flot << '(' << p.get_x() << ", " << p.get_y() << ", " << p.get_z() << ')';
  return flot; // à ne pas oublier !
}

// ======================================================================
int main()
```

```
{

  return 0;
}
```

Deux constructeurs sont définis : un constructeur par défaut qui construit le point de coordonnées (0,0,0), et un constructeur à partir des trois coordonnées.

Pour faire le constructeur par défaut, pourquoi ne pas avoir utilisé des arguments avec valeur par défaut, comme par exemple avec :

```
Point3D(double x = 0.0, double y = 0.0, double z = 0.0)
```

La réponse est claire : parce que cette façon de faire ne définit pas deux, mais *quatre* constructeurs ! Il serait en effet possible avec une telle approche de ne passer qu'une ou deux coordonnées :

```
  Point3D p1(3.2, 5.5);
  Point3D p2(2.5);
```

Cela ne nous semble pas satisfaisant et nous avons préféré restreindre la construction aux deux seuls cas : soit on donne les trois coordonnées, soit on n'en donne aucune (point origine).

Passons maintenant à la classe `Vecteur`. Elle *hérite* de la classe `Point3D`. Prenons égalemen le bon réflexe de définir tout de suite les constructeurs ; ce qui nous donne :

```
class Vecteur : public Point3D
{
public:
  Vecteur() : Point3D() {}
  Vecteur(double x, double y, double z) : Point3D(x, y, z) {}
};
```

Ceux-ci appellent ici simplement les constructeurs correspondant de la super-classe.

La suite relève plus du chapitre précédent, et ne devrait donc plus poser problème :

```
class Vecteur : public Point3D
{
public:
  Vecteur() : Point3D() {}
  Vecteur(double x, double y, double z) : Point3D(x, y, z) {}

  // opérateurs
  Vecteur& operator+=(const Vecteur& autre)
  {
    x += autre.x; y += autre.y; z += autre.z;
    return *this;
  }

  Vecteur& operator-=(const Vecteur& autre)
  {
    x -= autre.x; y -= autre.y; z -= autre.z;
    return *this;
  }

  // l'opposé
  const Vecteur operator-() const
  { return Vecteur(-x, -y, -z); }
```

```
  Vecteur& operator*=(double scal)
  {
    x *= scal; y *= scal; z *= scal;
    return *this;
  }

  double norme() const;
};

// ----------------------------------------------------------------------
const Vecteur operator+(Vecteur un, const Vecteur& autre)
{
  un += autre;
  return un;
}
const Vecteur operator-(Vecteur un, const Vecteur& autre)
{
  un -= autre;
  return un;
}
double operator*(const Vecteur& un, const Vecteur& autre)
/* Vu l'utilisation des accesseurs, une surcharge interne serait *
 *   justifiée ici...                                            */
{
  return    un.get_x() * autre.get_x()
          + un.get_y() * autre.get_y()
          + un.get_z() * autre.get_z();
}
const Vecteur operator*(Vecteur un, double x)
{
  un *= x;
  return un;
}
const Vecteur operator*(double x, Vecteur const& v)
{ return v * x; }

double Vecteur::norme() const
{ return sqrt(*this * *this); }

double angle(const Vecteur& un, const Vecteur& autre)
{ return acos((un * autre) / (un.norme() * autre.norme())); }
```

Notons la façon de systématiquement définir l'opérateur arithmétique (e.g. « + ») à partir de l'opérateur d'affectation correspondant (e.g. « += »). De même l'opérateur `operator*(double x, const Vecteur& v)` est défini à partir de l'opérateur symétrique. Cette façon de faire évite de dupliquer du code, i.e. de redéfinir la même chose à plusieurs endroits différents, et garantit la cohérence.

Voici le programme complet :

```
#include <iostream>
#include <cmath> // pour sqrt et acos
using namespace std;

// ----------------------------------------------------------------------
class Point3D
{
public:
```

```
  // constructeur par défaut
  Point3D() : x(0.0), y(0.0), z(0.0) {}
  // constructeur par 3 coordonnees
  Point3D(double x, double y, double z) : x(x), y(y), z(z) {}

  // methode d'acces en lecture
  double get_x() const { return x; }
  double get_y() const { return y; }
  double get_z() const { return z; }

protected:
  double x, y, z;
};

// ex compare()
bool operator==(const Point3D& un, const Point3D& autre)
{
  return     (autre.get_x() == un.get_x())
         and (autre.get_y() == un.get_y())
         and (autre.get_z() == un.get_z());
}

// ------------------------------------------------------------------
ostream& operator<<(ostream& flot, const Point3D& p)
{
  flot << '(' << p.get_x() << ", " << p.get_y() << ", " << p.get_z() << ')';
  return flot; // à ne pas oublier !
}

// ------------------------------------------------------------------
class Vecteur : public Point3D
{
public:
  Vecteur() : Point3D() {}
  Vecteur(double x, double y, double z) : Point3D(x, y, z) {}

  // opérateurs
  Vecteur& operator+=(const Vecteur& autre)
  {
    x += autre.x; y += autre.y; z += autre.z;
    return *this;
  }

  Vecteur& operator-=(const Vecteur& autre)
  {
    x -= autre.x; y -= autre.y; z -= autre.z;
    return *this;
  }

  // l'opposé
  const Vecteur operator-() const
  { return Vecteur(-x, -y, -z); }

  Vecteur& operator*=(double scal)
  {
    x *= scal; y *= scal; z *= scal;
    return *this;
  }
```

```
  double norme() const;
};

// ----------------------------------------------------------------------
const Vecteur operator+(Vecteur un, const Vecteur& autre)
{
  un += autre;
  return un;
}
const Vecteur operator-(Vecteur un, const Vecteur& autre)
{
  un -= autre;
  return un;
}
double operator*(const Vecteur& un, const Vecteur& autre)
/* Vu l'utilisation des accesseurs, une surcharge interne serait *
 *   justifiée ici...                                            */
{
  return    un.get_x() * autre.get_x()
          + un.get_y() * autre.get_y()
          + un.get_z() * autre.get_z();
}
const Vecteur operator*(Vecteur un, double x)
{
  un *= x;
  return un;
}
const Vecteur operator*(double x, Vecteur const& v)
{ return v * x; }

double Vecteur::norme() const
{ return sqrt(*this * *this); }

double angle(const Vecteur& un, const Vecteur& autre)
{ return acos((un * autre) / (un.norme() * autre.norme())); }


// ======================================================================
int main()
{
  Vecteur v1(1.0, 2.0, -0.1);
  Vecteur v2(2.6, 3.5, 4.1);

  cout << "Vecteur V1 : " << v1 << endl;
  cout << "Vecteur V2 : " << v2 << endl;

  cout << "V1 + V2  : " << v1 + v2  << endl;
  cout << "V1 - V2  : " << v1 - v2  << endl;
  cout << " -V1     : " << -v1      << endl;
  cout << "3.2 * V1 : " << 3.2 * v1 << endl;
  cout << "V1 * V2  : " << v1 * v2  << endl;

  cout << "norme de V1    : " << v1.norme() << endl;
  cout << "norme de V2    : " << v2.norme() << endl;
  cout << "norme de V1+V2 : " << (v1 + v2).norme() << endl;

  cout << "angle de V1 et V2 : " << angle(v1, v2) << endl;
  cout << "angle de V2 et V1 : " << angle(v2, v1) << endl;
```

```
  return 0;
}
```

Exercice 14 : vecteurs unitaires

> Cet exercice correspond à l'exercice n°57 (pages 141 et 324) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

L'héritage se fait sans difficulté :

```
class VecteurUnitaire : public Vecteur {};
```

Par contre, comme le souligne l'énoncé, il faut prêter une attention particulière au fait que tous les `VecteurUnitaire` existants soient toujours bien *unitaires*.

Il faut pour cela commencer par les constructeurs :

- Pour le constructeur par défaut, il faut choisir une valeur par défaut différente du vecteur nul (lequel n'est pas unitaire !), ou alors interdir la construction par défaut de cette classe (ce qui se fait simplement en précisant un autre constructeur, sans donner de constructeur par défaut).

- Tout constructeur de vecteur unitaire doit effectivement normaliser le vecteur construit. C'est ce que nous faisons au moyen de la méthode `normalise`. Cette méthode est mise en `private`, car *a priori* seule la classe `VecteurUnitaire` en a besoin.

  Il faut de plus faire attention de ne pas faire de division par zéro, dans le cas où cette méthode serait appelée avec le vecteur nul. Dans ce cas, nous pouvons choisir de plutôt renvoyer la valeur par défaut choisie précédement. Dans un cadre plus général, il serait toutefois préférable de « lancer une exception » (non abordées dans ce cours).

On aboutit alors au code suivant :

```
class VecteurUnitaire : public Vecteur
{
public:
  /* constructeur par défaut : choix d'une valeur arbitraire,       *
   * ici : le premier vecteur de base.                              */
  VecteurUnitaire() : Vecteur(1.0, 0.0, 0.0) {}

  // constructeur général : effectue la normalisation si nécessaire.
  VecteurUnitaire(double x, double y, double z) : Vecteur(x, y, z)
  { normalise(); }

  /* Copie d'un vecteur en un vecteur unitaire.
   * Construit le vecteur unitaire colinaire et de même sens
   * que le vecteur à copier.                                       */
  VecteurUnitaire(const Vecteur& a_copier) : Vecteur(a_copier)
  { normalise(); }

private:
  /* Nouvelle méthode normalisant le vecteur.
   * Cette méthode est privée car c'est de la
   * «cuisine interne» qui ne regarde personne d'autre */
  void normalise() {
    const double n(Vecteur::norme());
    /* Ce test devrai être rendu plus robuste :
     *      abs(n) < precision
     *  avec une precision à definir, voire passée en paramètre. */
    if (n == 0.0) {
      /* Si l'on cherche à normaliser le vecteur nul en tant que      *
       * vecteur unitaire (!!), nous choisissons de mettre à la place  *
```

```
        * la valeur par defaut choisie pour les vecteurs unitaires (en  *
        * utilisant ici le constructeur par défaut).                     */
       *this = VecteurUnitaire();
     } else {
       x /= n; y /= n; z /= n;
     }
   }
};
```

Il faut maintenant garantir que les vecteurs unitaires construits *restent* unitaires... et c'est ici que l'exercice devient de niveau 2 !

En effet, un `VecteurUnitaire` étant par héritage un `Vecteur`, on peut très bien écrire

```
v1 += w;
```

avec « `v1` » un vecteur unitaire et `w` un autre vecteur (unitaire ou non, peu importe). Cette opération, valide du point de vue du C++, a pour effet de rendre `v1` non unitaire (mathématiquement), alors que c'est toujours un `VecteurUnitaire` (informatiquement). Vous me suivez ?...

Il est donc *nécessaire* de redéfinir proprement tous les opérateurs qui risquent de transformer un vecteur unitaire en un vecteur non unitaire.

À ce stade plusieurs options sont possibles lorsque le vecteur cesse d'être unitaire. On peut choisir :

- d'afficher un message d'erreur ;
- de transformer le vecteur non unitaire en un vecteur unitaire colinéaire et de même sens, en divisant simplement celui-ci par sa norme (méthode `normalise` précédemment introduite) ;
- de rendre « privées » les méthodes « fautives » afin d'interdire leur utilisation (extérieure) ;
- de lancer une exception (non abordé dans de ce cours).

En pratique, il faut évaluer ce qui convient le mieux en fonction du contexte d'utilisation. Ici, sans autre indication, nous choisissons de présenter la seconde option. On redéfinit donc tous les opérateurs d'affectation :

```
  // redéfinition des opérateurs d'affectation
  VecteurUnitaire& operator+=(const Vecteur& autre) {
    Vecteur::operator+=(autre);
    normalise();
    return *this;
  }
  VecteurUnitaire& operator-=(const Vecteur& autre) {
    Vecteur::operator-=(autre);
    normalise();
    return *this;
  }
  VecteurUnitaire& operator*=(double x) {
    Vecteur::operator*=(x);
    normalise();
    return *this;
  }
```

Soulignons une fois de plus la réutilisation de l'existant (ici les opérateurs correspondants de la classe `Vecteur`) afin de ne pas avoir à redéfinir plusieurs fois les mêmes choses.

Ayant fait le choix précédent (forcer la normalisation colinéaire en cas de non-respect de la norme), il semble naturel de prévoir un constructeur de vecteur unitaire à partir d'un vecteur quelconque. Cela offre la possibilité de facilement construire le vecteur unitaire colinéaire et de même sens à un vecteur donné :

```
  // copie d'un vecteur en un vecteur unitaire.
  // construit le vecteur unitaire colinaire et de meme sens
  // que le vecteur à copier
  VecteurUnitaire(const Vecteur& a_copier) : Vecteur(a_copier)
  { normalise(); }
```

Revenons maintenant à l'énoncé et masquons la méthode `norme` et surchargeons la fonction `angle` ; mais attention, il faut alors, dans `normalise`, démasquer l'appel de la méthode `norme` héritée :

```
const double n(Vecteur::norme());
```

Voici donc le code complet :

```
#include <iostream>
#include <string> // pour message d'erreur
#include <cmath>  // pour sqrt et acos
using namespace std;

// ----------------------------------------------------------------------
class Point3D
{
  // etc... comme dans le corrigé précédent
};

// ----------------------------------------------------------------------
class Vecteur : public Point3D
{
  // etc... comme dans le corrigé précédent
};

// ## NOUVELLE PARTIE
// ----------------------------------------------------------------------
class VecteurUnitaire : public Vecteur
{
public:
  /* constructeur par défaut : choix d'une valeur arbitraire,       *
   * ici : le premier vecteur de base.                              */
  VecteurUnitaire() : Vecteur(1.0, 0.0, 0.0) {}

  // constructeur général : effectue la normalisation si nécessaire.
  VecteurUnitaire(double x, double y, double z) : Vecteur(x, y, z)
  { normalise(); }

  /* Copie d'un vecteur en un vecteur unitaire.
   * Construit le vecteur unitaire colinaire et de même sens
   * que le vecteur à copier.                                       */
  VecteurUnitaire(const Vecteur& a_copier) : Vecteur(a_copier)
  { normalise(); }

  // Redéfinition des opérateurs d'affectation
  VecteurUnitaire& operator+=(const Vecteur& autre) {
    Vecteur::operator+=(autre);
    normalise();
    return *this;
  }
  VecteurUnitaire& operator-=(const Vecteur& autre) {
    Vecteur::operator-=(autre);
    normalise();
```

```
    return *this;
  }
  VecteurUnitaire& operator*=(double x) {
    Vecteur::operator*=(x);
    normalise();
    return *this;
  }

  // Méthode masquante, plus efficace
  double norme() const { return 1.0; }

private:
  /* Nouvelle méthode normalisant le vecteur.
   * Cette méthode est privée car c'est de la
   * «cuisine interne» qui ne regarde personne d'autre */
  void normalise() {
    const double n(Vecteur::norme());
    /* Ce test devrai être rendu plus robuste :
     *      abs(n) < precision
     *  avec une precision à definir, voire passée en paramètre. */
    if (n == 0.0) {
      /* Si l'on cherche à normaliser le vecteur nul en tant que       *
       * vecteur unitaire (!!), nous choisissons de mettre à la place  *
       * la valeur par defaut choisie pour les vecteurs unitaires (en  *
       * utilisant ici le constructeur par défaut).                    */
      *this = VecteurUnitaire();
    } else {
      x /= n; y /= n; z /= n;
    }
  }
};

// Surcharge, plus efficace que la verions générale pour les Vecteurs
double angle(const VecteurUnitaire& un, const VecteurUnitaire& autre) const
{ return acos(un * autre); }


// ============================================================================
int main()
{
  VecteurUnitaire v1(1.0, 2.0, -0.1);
  VecteurUnitaire v2(2.6, 3.5, 4.1);

  cout << "Vecteur V1 : " << v1 << endl;
  cout << "Vecteur V2 : " << v2 << endl;
  cout << "norme de V1 : " << v1.Vecteur::norme() << endl;
  cout << "norme de V2 : " << v2.Vecteur::norme() << endl;
  // Note : nous appelons ici la méthode norme de Vecteur
  // a des fins de *VERIFICATION* (pour lesquelles appeler la méthode
  // norme optimisée ne nous apprendrait rien ici !)

  cout << "V1 + V2       : " << v1 + v2  << endl;
  VecteurUnitaire v3(v1 + v2);
  cout << "V3 = V1 + V2  : " << v3  << endl;

  cout << "norme de V1+V2 : " << (v1 + v2).norme() << endl;
  cout << "norme de V3    : " << v3.Vecteur::norme() << endl;

  v3 = Vecteur(1,2,3);
```

```
  cout << "V3 = (1,2,3) : " << v3  << endl;
  cout << "norme de V3    : " << v3.Vecteur::norme() << endl;

  v3 += v1;
  cout << "V3 += V1 : " << v3  << endl;
  cout << "norme de V3    : " << v3.Vecteur::norme() << endl;

  cout << "angle de V1 et V2 : " << v1.angle(v2) << endl;

  return 0;
}
```

Exercice 15 : algèbre élémentaire

> Cet exercice correspond à l'exercice n°58 (pages 142 et 324) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Cet exercice est de niveau 2, non pas en raison de sa difficulté informatique mais pour le niveau d'abstraction mis en œuvre. Il est en fait assez facile si l'on suit rigoureusement les directives d'implémentation.

Définir la classe `EnsembleFini` ... :

```
class EnsembleFini {
protected:
  unsigned int p;
};
```

`p` est « `protected` » car il est propre à la classe (i.e. pas d'accès public), mais on souhaite tout de même pouvoir y avoir accès dans les sous-classes (héritage).

Ajouter le constructeur correspondant :

Error: "src-book/codeCorr48-1.cc" is not a file!

Définir ensuite la classe `Groupe` ... :

```
class Groupe : public EnsembleFini {
public:
  unsigned int add(unsigned int x, unsigned int y) const {
    return (x+y) % p;
  }
};
```

Mais il ne faut pas oublier le constructeur :

```
class Groupe : public EnsembleFini {
public:
  Groupe(unsigned int p) : EnsembleFini(p) {}
  ...
};
```

En C++11, on pourrait aussi ici demander l'héritage du constructeur (puisqu'il n'y a pas d'attribut additionnel à initialiser) :

```
class Groupe : public EnsembleFini {
public:
  using EnsembleFini::EnsembleFini;
  ...
};
```

Définir ensuite la classe `Anneau` ... :

Cela se fait de la même façon :

```
class Anneau : public Groupe {
public:
  Anneau(unsigned int p) : Groupe(p) {}
```

```
  unsigned int mult(unsigned int x, unsigned int y) const {
    return (x*y) % p;
  }
};
```

Terminer par la classe `Corps` *[...]* :

```
class Corps : public Anneau {
public:
  Corps(unsigned int p) : Anneau(p) {}
  unsigned int inv(unsigned int x) const;
  unsigned int div(unsigned int x, unsigned int y) const {
    return mult(x, inv(y));
  }
};
```

Pour l'implémentation de `inv`, il suffit de réutiliser la division euclidienne généralisée comme indiqué par l'énoncé. En voici une version un tout petit peu adaptée (sans le calcul du coefficient de Bezout `v`, inutile ici) :

```
unsigned int Corps::inv(unsigned int x) const {
  // algorithme d'Euclide

  if (x != 0) {
    int prev_u(1), y(p), u(0), new_u, q, r;

    while (y != 1) {
      q = x/y;
      r = x%y;
      x = y;
      y = r;
      new_u = prev_u - q * u;
      prev_u = u;
      u = new_u;
    }

    return (u + p) % p; /* u%p seul n'est pas correct car ne          *
                         * fonctionne pas pour des questions de signe */
  } else
    return 0; // mieux: lever une exception ici
}
```

Tester le programme : Le résultat est le suivant :

Error: "src-book/codeCorr48-7.cc" is not a file!
