MOOC Intro POO C++
Exercices semaine 4

Exercice 12 : véhicules (niveau 1)

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

Dans un fichier `Vehicule.cc`, définir une classe `Vehicule` qui a pour attributs des informations valables pour tout type de véhicule : sa marque ; sa date d'achat ; son prix d'achat ; son prix courant.

Définir un constructeur prenant en paramètre les trois attributs correspondant à : la marque, la date d'achat et le prix d'achat (le prix courant sera calculé plus tard).

Définir une méthode publique `void affiche(ostream&) const;` qui affiche l'état de l'instance, i.e. la valeur de ses attributs.

Définir ensuite deux classes `Voiture` et `Avion`, *héritant* de la classe `Vehicule` et ayant les attributs supplémentaires suivants :

- pour la classe `Voiture` :
  - sa cylindrée ;
  - son nombre de portes ;
  - sa puissance ;
  - son kilométrage ;
- pour la classe `Avion` :
  - son type (hélices ou réaction) ;
  - son nombre d'heures de vol.

Pour chacune de ces classes, définir un constructeur qui permette l'initialisation explicite de l'ensemble des attributs, ainsi qu'une méthode affichant la valeur des attributs. Constructeurs et méthode d'affichage devront utiliser les méthodes appropriées de la classe parente !

Ajouter ensuite à la classe `Vehicule`, une méthode `void calculePrix()` qui donne le prix courant. On calculera ce prix courant en soustrayant au prix d'achat 1% par année écoulée depuis la date d'achat.

Redéfinir cette méthode dans les deux sous-classes `Voiture` et `Avion`, de sorte à calculer le prix courant en fonction de certains critères, et mettre à jour l'attribut correspondant au prix courant :

- Pour une voiture, le prix courant est égal au prix d'achat, moins :
  - 2% pour chaque année depuis sa fabrication jusqu'à la date actuelle ;
  - 5% pour chaque tranche de 10'000 km parcourus (on arrondit à la tranche la plus proche) ;
  - 10% s'il s'agit d'un véhicule de marque « Renault » ou « Fiat » (choix totalement arbitraire qu'on est bien sûr libre de modifier) ;
  - et *plus* 20% s'il s'agit d'un véhicule de marque « Ferrari » ou « Porsche » (même remarque que ci-dessus).
- Pour un avion, le prix courant est égal au prix d'achat, moins :
  - 10% pour chaque tranche de 1000 heures de vol s'il s'agit d'un avion à réaction ;

- 10% pour chaque tranche de 100 heures de vol s'il s'agit d'un avion à hélices.

Le prix doit rester positif (i.e., s'il est négatif, on le met à 0).

Afin de tester les méthodes implémentées ci-dessus, compléter le `main` comme suit :

```
int main()
{
  vector<Voiture> garage;
  vector<Avion>   hangar;

  garage.push_back(Voiture("Peugeot", 1998, 147325.79, 2.5, 5, 180.0,
                           12000));
  garage.push_back(Voiture("Porsche", 1985, 250000.00, 6.5, 2, 280.0,
                           81320));
  garage.push_back(Voiture("Fiat",    2001,   7327.30, 1.6, 3,  65.0,
                           3000));

  hangar.push_back(Avion("Cessna",      1972, 1230673.90, HELICES,
                         250));
  hangar.push_back(Avion("Nain Connu", 1992, 4321098.00, REACTION,
                         1300));

  for (auto voiture : garage) {
    voiture.calculePrix();
    voiture.affiche(cout);
  }

  for (auto avion : hangar) {
    avion.calculePrix();
    avion.affiche(cout);
  }

  return 0;
}
```

Exemple de déroulement (pour l'année 2015)

```
 ---- Voiture ----
marque : Peugeot, date d'achat : 1998, prix d'achat : 147326,
prix actuel : 88395.5
2.5 litres, 5 portes, 180 CV, 12000 km.
 ---- Voiture ----
marque : Porsche, date d'achat : 1985, prix d'achat : 250000,
prix actuel : 48350
6.5 litres, 2 portes, 280 CV, 81320 km.
 ---- Voiture ----
marque : Fiat, date d'achat : 2001, prix d'achat : 7327.3,
prix actuel : 4433.02
1.6 litres, 3 portes, 65 CV, 3000 km.
 ---- Avion a helices ----
marque : Cessna, date d'achat : 1972, prix d'achat : 1.23067e+06,
prix actuel : 923005
250 heures de vol.
```

 ---- Avion a reaction ----
marque : Nain Connu, date d'achat : 1992, prix d'achat : 4.3211e+06, prix actuel : 3.75936e+06
1300 heures de vol.

Exercice 13 : vecteurs 3D (niveau 1)

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

On s'intéresse ici à créer la classe représentant les vecteurs en dimension 3. Du point de vue conception, la classe `Vecteur` hérite naturellement de la classe `Point3D` présentée dans l'exercice 3.

Commencer par reprendre cet exercice en l'adaptant au vu des nouvelles connaissances : constructeurs et surcharges d'opérateurs.

Passer ensuite à la classe `Vecteur`. En plus de toutes les méthodes héritées, cette nouvelle classe doit pouvoir effectuer les opérations suivantes :

- addition et soustraction de deux vecteurs :

```
Vecteur& Vecteur::operator+=(const Vecteur&);
Vecteur& Vecteur::operator-=(const Vecteur&);
const Vecteur operator+(Vecteur, const Vecteur&);
const Vecteur operator-(Vecteur, const Vecteur&);
```

- opposé d'un vecteur :

```
const Vecteur Vecteur::operator-() const;
```

- multiplication par un scalaire :

```
Vecteur& Vecteur::operator*=(double);

const Vecteur operator*(Vecteur, double);
const Vecteur operator*(double, const Vecteur&);
```

- produit scalaire de deux vecteurs :

```
double operator*(const Vecteur&, const Vecteur&);
```

- calcul de la norme du vecteur (racine carrée du produit scalaire avec lui-même) ;
- angle entre deux vecteurs, comme l'arc-cosinus de leur produit scalaire divisé par le produit des normes :

```
double angle(const Vecteur& v1, const Vecteur& v2)
{ return acos((v1 * v2) / (v1.norme() * v2.norme())); }
```

Tester ces opérations sur des vecteurs de son choix, par exemple :

```
(1 2 -0.1) + (2.6 3.5 4.1) = (3.6 5.5 4)
(2.6 3.5 4.1) + (1 2 -0.1) = (3.6 5.5 4)
(1 2 -0.1) + (0 0 0) = (1 2 -0.1)
(0 0 0) + (1 2 -0.1) = (1 2 -0.1)
(1 2 -0.1) - (2.6 3.5 4.1) = (-1.6 -1.5 -4.2)
(2.6 3.5 4.1) - (2.6 3.5 4.1) = (0 0 0)
- (1 2 -0.1) = (-1 -2 0.1)
- (2.6 3.5 4.1) + (1 2 -0.1) = (-1.6 -1.5 -4.2)
```

```
3 * (1 2 -0.1) = (3 6 -0.3)
(1 2 -0.1) * (2.6 3.5 4.1) = 9.19
(2.6 3.5 4.1) * (1 2 -0.1) = 9.19
||(1 2 -0.1)|| = 2.23830292855994
||(2.6 3.5 4.1)|| = 5.98498120297800
angle( (2.6 3.5 4.1), (1 2 -0.1)) = 0.814797988557
```

Exercice 14 : vecteurs unitaires (niveau 2)

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

On souhaite maintenant écrire une classe représentant des vecteurs unitaires (i.e. des vecteurs de norme 1). Il semble naturel que cette classe hérite de la classe `Vecteur` (un vecteur unitaire « est un » vecteur).

Prêter une attention particulière à ce que le vecteur *reste effectivement* unitaire.

Une fois la définition de la classe et ses éventuelles méthodes terminée, masquer la méthode `angle` héritée de `Vecteur` par une version plus efficace qui ne divise pas par le produit des normes (ce qui est inutile dans le cas de vecteurs *unitaires* !).

Essayer d'appliquer la même idée à la méthode `norme`.

Exercice 15 : un peu d'algèbre élémentaire [...désolé !] (niveau 3)

Cet exercice correspond à l'exercice n°58 (pages 142 et 328) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Voici une version informatique, orientée objets, d'un cours d'algèbre élémentaire (ceux qui n'aiment pas les maths passeront simplement leur chemin ;-))

Un *groupe* est un *ensemble* muni d'une loi de composition interne qui possède un certain nombre de propriétés (associativité, existence d'un élément neutre et d'un symétrique).

Par ailleurs, un *anneau* est un groupe muni d'une seconde loi de composition interne, associative, distributive par rapport à la loi du groupe et admettant aussi un élément neutre.

Finalement, un *corps* est un anneau pour lequel la seconde loi est aussi une loi de groupe.

Dans une vision orientée objets, cela veut dire que la classe `Corps` hérite de la classe `Anneau`, qui hérite de la classe `Groupe`, laquelle hérite finalement de la classe `Ensemble`.

C'est ce que nous proposons d'implémenter dans cet exercice, en se restreignant ici (pour des raison de simplicité) aux anneaux finis Z/pZ (Z, ensemble des entiers relatifs) et de tester avec le corps fini $Z/5Z$, c.-à-d. en effectuant les opérations modulo 5 (par exemple 4+3 = 2).

Définir de façon simple la classe `EnsembleFini` représentant l'ensemble des entiers naturels entre 0 et p : il suffit simplement de stocker l'entier (positif) p.

Ajouter le constructeur correspondant.

Définir ensuite la classe `Groupe` qui hérite de la classe `EnsembleFini` et contient une méthode `add`, prenant deux entiers non signés en argument et retournant l'entier non signé correspondant à leur somme modulo `p`.

Définir ensuite la classe `Anneau` qui hérite de la classe `Groupe` et contient une méthode `mult`, prenant deux entiers non signés en argument et retournant l'entier non signé correspondant à leur produit modulo `p`.

Terminer par la classe `Corps` qui hérite de la classe `Anneau` et contient deux méthodes supplémentaires `inv` et `div`.

`inv(x)` calcule l'inverse (au sens du corps), c.-à-d. l'entier (positif) `y` tel que `mult(x, y) = 1`.

Pour le calcul de cet inverse, p est un nombre premier ici et on cherche y tel que $$(x\cdot y = 1) \mod p$$, c.-à-d. k tel que $$x \cdot y + k \cdot p = 1$$, ce qui se trouve à l'aide de l'algorithme d'Euclide que nous avions donné comme exercices supplémentaire dans notre MOOC de Septembre et que je reproduis ci-dessous pour ceux qui ne l'ont pas encore fait et voudraient aller au bout de cet exercice.

Pour sa part, la méthode `div` effectue simplement la multiplication par l'inverse.

Tester avec le `main` suivant :

```
int main()
{
  Corps k(5); // c'est le corps Z/5Z
```

```
  cout << "0 + 1 = " << k.add(0, 1)  << endl;
  cout << "3 + 3 = " << k.add(3, 3)  << endl;
  cout << "3 * 2 = " << k.mult(3, 2) << endl;
  cout << "1/2 = "   << k.inv(2)     << endl;
  cout << "3 * 4 = " << k.mult(3, 4) << endl;

  return 0;
}
```

Algorithme d'Euclide (calcul de PGCD)

> Cet exercice correspond à l'exercice n°38 (pages 90 et 272) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

(PGDC = plus grand diviseur commun)

Buts

Écrivez le programme `pgdc.cc` qui :

1. demande à l'utilisateur d'entrer deux entiers strictement positifs `a` et `b`;
2. teste si `a` et `b` sont bien strictement positifs, et dans le cas contraire les redemande à l'utilisateur.
3. trouve les entiers `u`, `v` et `p` satisfaisant l'identité de Bezout (i.e. une équation à valeurs entières) : `u a + v b = p`, avec `p` le plus grand commun diviseur de `a` et `b`.

Méthode

La méthode utilisée est **l'algorithme d'Euclide**.

On procédera par itération, comme suit (en notant `x / y` le quotient et `x % y` le reste de la division entière de `x` par `y`) :

0 : initialisation			$u_0 = 1$	$v_0 = 0$
	$x_1 = a$	$y_1 = b$	$u_1 = 0$	$v_1 = 1$
	...	...	...	...
i+1 : itération	$x_{i+1} = y_i$	$y_{i+1} = x_i \% y_i$	$u_{i+1} = u_{i-1} - u_i(x_i / y_i)$	$v_{i+1} = v_{i-1} - v_i(x_i / y_i)$
	...	...	...	...
Valeurs finales	x_{k-1}	y_{k-1} **!= 0**	u_{k-1}	v_{k-1}
k : condition d'arrêt quand $y_k = 0$	**$p = x_k$**	$y_k = 0$	inutile	inutile

C'est-à-dire que l'on va calculer de proche en proche les valeurs de `x`, `y`, `u` et `v`. En calculant à chaque fois les nouvelles valeurs en fonction des anciennes (et en faisant bien attention à mémoriser ce qui est nécessaire à un calcul correct, voir les indications ci-dessous).

Par exemple, $y_{i+1} = x_i \% y_i$ veut dire : "la nouvelle valeur de `y` vaut l'ancienne valeur de `x` modulo l'ancienne valeur de `y`".

Programmez ces calculs dans une boucle, qui s'execute tant que la condition d'arrêt n'est pas vérifiée.

Pensez à initialiser correctement vos variables avant d'entrer dans la boucle.

Indications

Vu les dépendances entre les calculs, vous aurez besoin de définir (par exemple) les variables : `x`, `y`, `u`, `v` **et** `q=x/y`, `r=x%y`, `prev_u`, `prev_v`, `new_u` et `new_v`.

Vous mettrez ces variables à jour à chaque itération, à l'aide des formules de la ligne *i+1* et des relations temporelles évidentes entre elle (par exemple `prev_u = u`).

Testez si `y` est non nul avant d'effectuer les divisions !

Exemple d'execution

```
Entrez un nombre entier supérieur ou égal à 1 : 654321
Entrez un nombre entier supérieur ou égal à 1 : 210
Calcul du PGDC de 654321 et 210

  x         y         u         v
210       171         1     -3115
171        39        -1      3116
 39        15         5    -15579
 15         9       -11     34274
  9         6        16    -49853
  6         3       -27     84127
  3         0        70   -218107
PGDC(654321, 210)=3
```

Note

- Remarquez que pour le seul calcul du PGDC, le calcul de `x` et `y` par l'algorithme ci-dessus suffit, pas besoin de `u` et `v`. Ils ont été introduits ici pour trouver l'équation de Bezout (et vous faire programmer des suites imbriquées). Par exemple sur l'exemple précédent on a :
  -27 * 654321 + 84127 * 210 = 3.
