**Université Paris Diderot** **Langages à objets avancés**
**M1** **Année 2021-2022**

# Surcharge d'opérateurs

**Exercice 1**

1. Definir une classe `Vecteur` qui représente un vecteur de $\mathbb{R}^2$ avec ses constructeurs.
2. Surcharger les opérateurs `==` et `!=`, afin de pouvoir tester si deux vecteurs sont égaux ou différents.
3. Surcharger les opérateurs `+` et `-` afin de pouvoir faire la somme et la différence de deux vecteurs, et l'opérateur `*` afin de pouvoir réaliser le produit scalaire de deux vecteurs.
4. Surcharger l'opérateur `[]` afin de pouvoir accéder à une coordonnées du vecteur. Tester en particulier :

```
Vecteur v;
v[0] = 9;
```

5. Ajouter une méthode `double norm()` renvoyant la norme du vecteur.
6. Faire en sorte que l'on puisse afficher un vecteur `v` en invoquant l'instruction `cout << v << endl`.

**Exercice 2** [Figures et diamants] Dans cet exercice, on utilise l'héritage pour définir des classes représentant des figures 2-dimensionnelles. On pourra utiliser la classe définie dans l'exercice précédent pour définir ces classes.

1. Déclarer une classe `Figure`. Cette classe aura un attribut de type `string` représentant le nom de la figure et un constructeur prenant ce `string` en paramètre. Déclarer de plus des méthodes virtuelles `void print()` et `double area()`. Comme la classe `Figure` n'est pas censée être instanciée directement, donnez des définitions arbitraires à ces méthodes ou, mieux, déclarez-les *virtuelles pures*.
2. Déclarer une classe `Triangle` qui hérite de `Figure` avec un constructeur avec le nombre d'arguments adéquat. Un triangle sera la donnée de trois points. Définir la méthode `print()` de façon à afficher "`triangle [nom]: `" suivi des coordonnées des trois points (où `[nom]` aura été remplacé par le nom de la figure). Définir la méthode `area()` de façon à calculer la surface du triangle.
3. Déclarer une classe `Quadrilatere` qui hérite de `Figure` avec un constructeur. Un objet `Quadrilatere` sera la donnée de quatre points dans le plan. Dans le cas où vous savez calculer l'aire d'un quadrilatère, définissez les méthodes `print()` et `area()` comme pour `Triangle`.
4. Déclarer une classe `Rectangle` qui héritera de `Quadrilatere` avec un constructeur. Un objet `Rectangle` sera la donnée de quatre points (associés au quadrilatère) formant un rectangle. Définir les méthodes `print()` et `area()` comme pour `Triangle`.

5. Déclarer une classe `Losange` qui hérite de `Quadrilatere` avec un constructeur. Un objet `Losange` sera la donnée de quatre points formant un losange. Définir les méthodes `print()` et `area()`.
6. Déclarer une classe `Carre` qui héritera à la fois de `Rectangle` et `Losange` avec un constructeur. Un objet `Carre` sera la donnée de quatre points formant un carré. Faites en sorte qu'un `Carre` ne soit associé qu'à un seul `Quadrilatere`. Définir les méthodes `print()` et `area()`.
7. Tester sur des exemples simples. Vérifier en particulier que

```
Figure *p = new Carre(...); // ... à remplir
p->print();
```

   affiche bien `carre [nom]: ...` et que la valeur renvoyée par `area()` est correcte.

**Exercice 3** — Opérateur de sortie et opérateurs arithmétiques

On souhaite écrire une classe `Fraction` qui représente les nombres en fractions entières. L'un des constructeurs de cette classe prendra en entrée deux entiers $n$ et $d$ (numérateur et dénominateur) et stockera les deux entiers correspondant à la fraction irréductible : (c.à.d tel que seul $n$ peut être négatif, et tel que $n$ et $d$ ont été divisés par leur *pgcd*)

1. Définissez cette classe et écrivez la méthode statique privée qui calcule le *pgcd*. Pour gagner du temps, voici le code du *pgcd* (adaptez en fonction de vos besoins) :

```
int pgcd(int x, int y) {
  if (x<0) return pgcd(-x,y);
  if (y<0) return pgcd(x,-y);
  if (x==0) return y;
  if (x>y) return pgcd(y,x);
  return pgcd(x,y-x);
}
```

2. Écrivez deux constructeurs pour la classe `Fraction` : un constructeur par copie `Fraction(const Fraction&)` et le constructeur `Fraction(int, int)` décrit au début de l'exercice.
3. Redéfinissez l'opérateur de sortie `<<` pour permettre un affichage et écrivez un petit main pour la tester.

   Rappel : si vous avez besoin de déclarer cette fonction comme amie de la classe `Fraction` vous pourrez le faire en ajoutant cette déclaration à la classe :

```
friend std::ostream& operator<<(std::ostream&, const Fraction&);
```

4. Surchargez les opérateurs `+` et `-` sur les fractions.
5. Vérifiez qu'une opération $\frac{1}{2} + 1$ ne compile pas. Ajoutez simplement un constructeur de fractions à un seul argument, le second valant 1. Vérifiez que l'expression précédente est à présent évaluée. Quel est le mécanisme qui a été mis en œuvre ?
6. Complétez afin de pouvoir évaluer également $1 + \frac{2}{4}$.
7. Relisez vos déclarations de fonctions pour utiliser au maximum le mot-clé `const` et des variables références au lieu de variables copiées lorsqu'elles sont passées en argument.

**Exercice 4** — Opérateur « () »

On définit l'« interface » (i.e. classe avec seulement des méthodes virtuelles pures) ci-dessous.

```
class ValeursAdmises {//"interface"
 public :
  virtual  bool operator()(char val) = 0;
};
```

On y indique que l'opérateur « `()` » est défini comme prenant un argument `val` de type `char`, cela permettra d'utiliser une notation fonctionnelle sur l'objet pour vérifier qu'une valeur `val` répond à certains critères.

1. Ecrivez une sous-classe concrète `Intervalle` implémentant l'interface dont les objets sont définis par une valeur `char min` et une valeur `char max`, et dont la « fonction » correspondant à l'opérateur redéfini vérifiera que la valeur donnée en argument est dans cet intervalle.

   *Exemple d'utilisation :*

   ```
   Intervalle inter {'a', 'd'};
   if(inter('e'))  // utilisation d'operator(char)
    cout << "la valeur 'e' est ok" << endl;
   else
    cout << "la valeur 'e' n'est pas ok" << endl;
   if(inter('c'))
    cout << "la valeur 'c' est ok" << endl;
   else
    cout << "la valeur 'c' n'est pas ok" << endl;
   ```

2. Faites de même pour `TableauValeurs` dont les objets encapsulent un tableau de caractères et dont la « fonction » va vérifier que la valeur donnée en argument est dans ce tableau.

   *Exemple d'utilisation :*

   ```
   char tab[] {'b', 'o', 'n', 'j', 'u', 'r'};
   TableauValeurs tableau { tab, 6 };
    if(tableau('j'))
     cout << "la valeur 'j' est ok" << endl;
   else
      cout<< "la valeur 'j' n'est pas ok"<< endl;
   if(tableau('c'))
     cout << "la valeur 'c' est ok" << endl;
   else
      cout << "la valeur 'c' n'est pas ok" << endl;
   ```

3. Écrivez une fonction :

   ```
   std::vector<char> filtre(const std::vector<char>&,
                            const ValeursAdmises&);
   ```

   qui va prendre en argument un tableau d'éléments de type `char` et un objet de type `ValeursAdmises` et retournera un tableau ne contenant que les éléments `l` qui sont admis. Testez la.

   *Exemple d'utilisation :*

```
vector<char> res =
  filtre(vector<char> {'a', 'b','z','o'}, tableau);

for(char const& val: res)
  cout << val << ";";

cout << endl;
```