Université Paris Diderot **Langages à objets avancés**
M1 **Année 2019-2020**

Surcharge d'opérateurs, héritage multiple

Surcharge d'opérateur

Exercice 1 — 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 (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 2 — 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;
```

Héritage Multiple

Exercice 3 On veut représenter les articles d'un magasin et pour chacun d'eux savoir son nom, son prix et, si les données sont disponibles et ont un sens, le coût écologique (en grammes de CO2) et le nombre de calories qu'ils représentent.

On aura 4 classes :

— `Article` pour les articles dont on ne connaît que le prix ;
— `ArticleCaloriMesurable` pour ceux dont on connaît le prix et le nombre de calories ;
— `ArticleEcoMesurable` pour les articles dont on connaît le prix et le coût écologique ;
— `ArticleMesurable` pour ceux pour lesquels on a toutes les données.

Écrivez ces 4 classes. On y mettra une méthode `virtual affiche()const` que l'on redéfinira dans toutes les classes. Pensez à définir également les destructeurs et les constructeurs par copies. Et testez tout cela :

```
ArticleMesurable am { 42, 36, 54 };

am.affiche(); cout << endl;
am.ArticleEcoMesurable::affiche(); cout << endl;
am.ArticleCaloriMesurable::affiche(); cout << endl;
am.Article::affiche();
```

Que se passe-t'il si, dans `ArticleMesurable`, on ne redéfinit pas la fonction `affiche()` ?