IN204
Programmation Orientée Objet – Examen de mise en œuvre des notions de C++

Examen du 16 novembre 2021

B. Monsuez

NOM :	
PRENOM :	

Question n°1 : Constructeurs

Nous nous intéressons à la classe suivante qui définit un nombre rationnel sous la forme d'une paire de deux entiers a et b représentant le nombre rationnel

$$\frac{a}{b}$$

```
class rational
{
private:
    int numerator;
    int denominator;
public:
    rational();
    rational(int anInteger);
    rational(int aNumerator, int aDenominator);
};
```

Question n°1.1 :

Pour chacun des constructeurs, expliquer la fonction du constructeur et dites si le constructeur est un constructeur que C++ génère automatiquement en son absence ou pas.

```
rational();
```

```
rational(int anInteger);
```

(Expliquer pourquoi il n'y a pas le mon clé explicit.)

```
rational(int aNumerator, int aDenominator);
```

Question n°1.2 :

Ecrivez le code d'initialisation qu'effectue le constructeur.

```
rational();
```

```
explicit rational(int anInteger);
```

```
rational(int aNumerator, int aDenominator);
```

```
        rational(const rational& anotherRational);
```

```
```

Question n° 2 : Accéder aux données internes stockées dans les champs

Question n° 2.1 : Champ privé

Est-il possible d'accéder aux champs `numerator` et `denominator`? Pourquoi donc ?

Question n° 2.2 : Proposez des méthodes pour accéder en lecture aux champs stockées dans la classe `rational`.

Question n° 3 : Opérateurs de conversions

Nous souhaitons convertir un nombre rationnel en un nombre flottant ayant pour type `double`. Cet opérateur doit effectuer la division de `numerator` par `denominator` et d'en retourner le résultat.

Ecrivez cet opérateur de conversion.

Question n° 4 : Opérateurs de comparaison

Définissez les opérations de comparaisons entre valeurs rationnelles.

```
class rational
{
private:
    int numerator;
    int denominator;
public:
    rational();
    explicit rational(int);
    rational(int aNumerator, int aDenominator);

    ...

    bool operator == (const rational&) const;
    bool operator != (const rational&) const;
};
```

Question 4.1

Proposer le code pour les deux opérateurs suivants :

```
    bool operator == (const rational&) const;
    bool operator != (const rational&) const;
```

Question n° 5 : Opérations arithmétiques

Définissez les opérateurs '+', '-', '*', et '/' pour les nombres rationnels. On rappelle les formules suivantes :

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd}$$

$$\frac{a}{b}.\frac{c}{d}=\frac{ac}{bd}$$

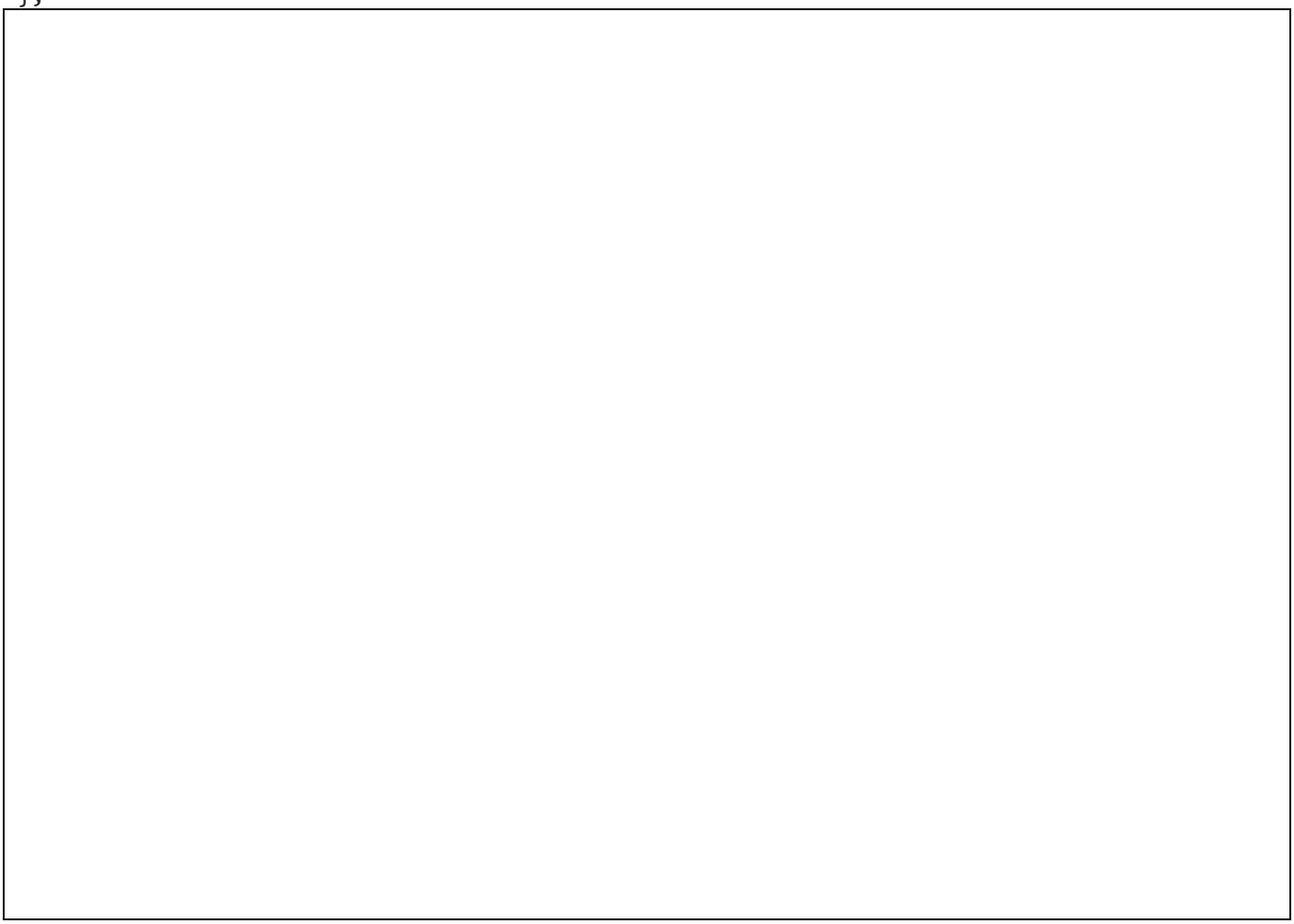
Question n° 6 : Patrons

La classe `rational` est définie à partir des entiers. Cependant, parfois nous avons besoin de nombres rationnels pouvant être exprimés par des numérateurs et dénominateurs de plus petites tailles. Dans ce cas, il n'est pas besoin de 2*32 = 64 bits de mémoire mais uniquement de par exemple 2*16 bits de mémoire (type `int16_t`), voir même 1*8 bits de mémoire (type `int8_t`). Dans d'autres cas, nous aimerions gagner en précision et au contraire avoir des numérateurs et dénominateurs de plus grandes tailles. Dans ce cas, nous pouvons imaginer avoir comme type pour les numérateurs et dénominateurs le type `int64_t` par exemple.

Question n°6.1 :

Transformer la classe « `rational` » en la paramétrant par le type entier servant à stocker les numérateurs et dénominateurs dans la classe. Pour rappel, le squelette de la classe est le suivant :

```
class rational
{
private:
    int numerator;
    int denominator;
public:
    rational();
    explicit rational(int);
    rational(int aNumerator, int aDenominator);
    ...
};
```



Question n°6.2 :

Dites parmi les définitions suivantes

- Celles qui sont correctes,
- Celles qui compilent et ne sont pas correctes (le comportement n'est pas celui attendu)
- Celles qui ne compilent pas

(Mettre une croix dans les cases du tableau suivant)

	Compile	Correct
`rational<int>`		
`rational<bool>`		
`rational<unsigned>`		
`rational<short int>`		
`rational<long int>`		
`rational<std::string>`		
`rational<double>`		

Question n°6.3 :

Nous ajoutons aux opérateurs '*' déjà définis l'opérateur suivant :

```
class rational
{
...
public:
    rational();
    explicit rational(int);
    rational(int aNumerator, int aDenominator);

    operator double() const { return numerator / denominator; }

    template<class integerT>
    rational operator *(integerT anIntegerValue)
    {
        return rational(numerator * anIntegerValue,
            denominator);
    }

...
}
```

Question n°6.3.1

Expliquer le comportement de l'opérateur `template<class integerT> rational operator *(integerT anIntegerValue).`

Question n°6.3.2

Sachant que nous avons deux opérateurs '*' ,

```
rational operator *( const rational<T>& anIntegerValue) {...}

template<class integerT>
rational operator *(integerT anIntegerValue)
{
    return rational(numerator * anIntegerValue,
        denominator);
}
```

Indiquez pour chacun des appels suivants lequel des deux opérateurs qui sera appelé :

```
rational<int> half(1, 2);
rational<int> athird(1, 3);
short value = 2;

auto a = athird * half;
auto b = half * value;
```

Question n°6.3.3

Est-ce que l'opération suivante est supportée :

```
rational<int> half(1, 2);
rational<short> athird(1, 3);
auto a = athird * half;
```

Transformer la definition de :

```
rational operator *( const rational<T>& anIntegerValue) {...}
```

pour supporter cette operation.

Question n°7 : Opérateurs surchargés & Flux

Nous souhaitons écrire sur un flux de type. Pour ce faire, nous envisageons de définir un opérateur << qui a la signature suivante :

```
template<class charT, class traits, class T>
std::basic_ostream<charT, traits>& operator << (std::basic_ostream<charT, traits>& aStream,
rational<T> theRational);
```

Question n°7.1 :

Pourquoi cet opérateur ne peut pas être défini comme les opérateurs précédents dans la classe rational<T> ?

Question n°7.2 :

Proposer une implantation de l'opérateur

```
template<class charT, class traits, class T>
std::basic_ostream<charT, traits>& operator << (std::basic_ostream<charT, traits>& aStream,
rational<T> theRational);
```

qui pour

- rational<T> (1, 2) retourne l'affichage suivant : 1/2
- rational<T> (0, 2) retourne l'affichage suivant : 0
- rational<T> (4, 1) retourne l'affichage suivant : 1
- rational<T> (1, 0) retourne l'affichage suivant : inf