2019

上級プログラミング

第二回レポート課題

2019/05/19

US162039　梶田悠

目次

レポートの課題内容

ソースコード 1 に示す main 関数で分数についての各種演算ができるように，以下で述べるような分数のクラス Fraction を作成し，作成したクラスが正しく動作しているかを， 図 1， 図 2 及び図 3 の実行例を参考に確認せよ．ただし，実行例の図中の下線はキーボードからの入力を示す．

■Fraction クラスの要件 Fraction クラスはデータメンバとして 2 つの整数型変数を持ち，それぞれの変数は分子と分母を表す．コンストラクタでは，分数の初期化を行う．分数が負の場合は分子を負数にする．分母が 0 の場合はエラーを表示し，プログラムを終了する．コンストラクタに引数が与えられなかった場合，分母が 1，分子が 0 となるように実装する．Fraction クラスは自分自身を約分する reduce 関数をメンバに持つ．また，中置演算子（+,-,*,/,>,<）を用いての四則演算や大小比較も可能である．加えて，演算子“<<”及び“>>”をオーバロードしているため，分数の標準入出力も可能である．Fraction クラスを出力する際，一つの分数を括弧で括り，(1/2) や(-2/3) のように表示する．但し，分母が 1 の場合は分数の形式ではなく，通常の整数のように，2 や 3 のように表示する．

補足

コードは github にアップロードしてあるので、レポートのコードが読みづらい場合は適宜参照して下さい。

分数の演算において最大公約数、最小公倍数を求める必要があった為、std::gcd()関数、std::lcm()関数を利用した。これらは C++17 以降に実装されているためプログラムを実行し挙動を確認する場合は C++17 に準拠した環境で行う必要があるので注意して下さい。

コード

以下にソースコードを掲載する。main()関数を含むファイルのコードは割愛する。

コード 1　Fraction.h

```
#pragma once
#include <iostream>
#include <cstdlib>
#include <numeric> //for std::gcd() std::lcm()
#include <cmath>   //for signbit()
using namespace std;

class Fraction {
    //numrator    : 分子
    //denominator : 分母
private:
    int numrator, denominator;
    //符号整理
    void adjustSign();
    void reduce();

public:
    Fraction(int numerator = 0, int denominator = 1);
    int getNumerator()   const { return numrator; }
    int getDenominator() const { return denominator; }
    //double型小数表現に変換した値を返す
    double getScaler() const { return (double)numrator / denominator; }
    bool isMinus() const;
    //四則演算
    void operator+=(const Fraction&);
    void operator-=(const Fraction&);
    void operator*=(const Fraction&);
    void operator/=(const Fraction&);
    bool operator< (const Fraction&) const;
    bool operator> (const Fraction&) const;
    bool operator==(const Fraction&) const;
    friend istream& operator>>(istream&, Fraction&);
};

//四則演算
Fraction operator+(const Fraction&, const Fraction&);
Fraction operator-(const Fraction&, const Fraction&);
Fraction operator*(const Fraction&, const Fraction&);
Fraction operator/(const Fraction&, const Fraction&);
```

```
//入出力（入力のみfriend）
istream& operator>>(istream&, Fraction&);
ostream& operator<<(ostream&, const Fraction&);
```

コード 2　Fraction.cpp

```
#include "Fraction.h"
//コンストラクタ
//default : numrator = 0, denominator = 1
Fraction::Fraction(int numrator, int denominator) {
    if (denominator == 0) {
        cerr << "分母に0が設定されています" << endl;
        exit(EXIT_FAILURE);
    }
    this->numrator = numrator;
    this->denominator = denominator;
    reduce();
    adjustSign();
}

//四則演算子
void Fraction::operator+=(const Fraction& frac) {
    //分母の最小公倍数を求めて分母を揃えて加算
    //※c++17 or later
    int LCM = lcm(this->getDenominator(), frac.getDenominator());
    int mul0 = LCM / this->getDenominator();
    int mul1 = LCM / frac.getDenominator();

    int numrator = mul0 * this->getNumerator() + mul1 * frac.getNumerator();
    int denominator = LCM;

    this->numrator = numrator;
    this->denominator = denominator;
    reduce();
    adjustSign();
}
void Fraction::operator-=(const Fraction& frac) {
    //分母の最小公倍数を求めて分母を揃えて減算
    //※c++17 or later
    int LCM = lcm(this->getDenominator(), frac.getDenominator());
    int mul0 = LCM / this->getDenominator();
    int mul1 = LCM / frac.getDenominator();

    int numrator = mul0 * this->getNumerator() - mul1 * frac.getNumerator();
    int denominator = LCM;

    this->numrator = numrator;
```

```
    this->denominator = denominator;
    reduce();
    adjustSign();
}
void Fraction::operator*=(const Fraction& frac) {
    this->numrator *= frac.getNumerator();
    this->denominator *= frac.getDenominator();
    reduce();
    adjustSign();
}
void Fraction::operator/=(const Fraction& frac) {
    this->numrator *= frac.getDenominator();
    this->denominator *= frac.getNumerator();
    reduce();
    adjustSign();
}

Fraction operator+(const Fraction& f0, const Fraction& f1) {
    Fraction f = f0;
    f += f1;
    return f;
}
Fraction operator-(const Fraction& f0, const Fraction& f1) {
    Fraction f = f0;
    f -= f1;
    return f;
}
Fraction operator*(const Fraction& f0, const Fraction& f1) {
    Fraction f = f0;
    f *= f1;
    return f;
}
Fraction operator/(const Fraction& f0, const Fraction& f1) {
    Fraction f = f0;
    f /= f1;
    return f;
}

//比較演算子
bool Fraction::operator<(const Fraction& frac) const {
    return this->getScaler() < frac.getScaler();
}
bool Fraction::operator>(const Fraction& frac) const {
    return this->getScaler() > frac.getScaler();
}
bool Fraction::operator==(const Fraction& frac) const {
    return this->getScaler() == frac.getScaler();
}
```

```
//入出力（入力のみfriend）
istream& operator>>(istream& stream, Fraction& frac) {
    int numerator, denominator;
    stream >> numerator >> denominator;
    if (denominator == 0) {
        cerr << "分母に0が設定されています" << endl;
        exit(EXIT_FAILURE);
    }
    frac.numrator = numerator;
    frac.denominator = denominator;
    frac.reduce();
    frac.adjustSign();
    return stream;
}
ostream& operator<<(ostream& stream, const Fraction& frac) {
    if (frac.getDenominator() == 1 || frac.getNumerator() == 0) stream <<
frac.getNumerator();
    else stream << "(" << frac.getNumerator() << "/" << frac.getDenominator() <<
")";
    return stream;
}

//約分
void Fraction::reduce() {
    //※c++17 or later
    int GCD = gcd(this->getNumerator(), this->getDenominator());
    if (GCD == 0) return;
    this->numrator /= GCD;
    this->denominator /= GCD;
}

bool Fraction::isMinus() const {
    bool isMinus = signbit(getScaler()) != 0;
    return isMinus;
}

//符号整理
void Fraction::adjustSign() {
    if (this->isMinus()) {
        this->denominator = abs(this->getDenominator());
        this->numrator = abs(this->getNumerator()) * -1;
    }
    else {
        this->denominator = abs(this->getDenominator());
        this->numrator = abs(this->getNumerator());
    }
}
```

考察

入出力演算子

入出力演算子のオーバーロードは以下のように宣言した。

```
class Fraction {
public:
    friend istream& operator>>(istream&, Fraction&);
};

//入出力（入力のみfriend）
istream& operator>>(istream&, Fraction&);
ostream& operator<<(ostream&, const Fraction&);
```

出力は分子、分母の値をゲッタで取得し出力するだけなので private な変数にアクセスする必要がないため Fraction クラスの friend 関数にしていないが、入力の場合は分子、分母を表す numrator 変数、denominator 変数の値を更新する必要がある。この 2 変数は private な変数なので friend 関数にする必要がある為上記のように記述した。

四則演算子

四則演算子のオーバーロードは以下のように宣言した。

```
class Fraction {
public:
    //四則演算
    void operator+=(const Fraction&);
    void operator-=(const Fraction&);
    void operator*=(const Fraction&);
```

```
    void operator/=(const Fraction&);
};

//四則演算
Fraction operator+(const Fraction&, const Fraction&);
Fraction operator-(const Fraction&, const Fraction&);
Fraction operator*(const Fraction&, const Fraction&);
Fraction operator/(const Fraction&, const Fraction&);
```

+=,-=,*=,/=演算子のオーバーロードについては、これらは private な変数の値を書き換える必要があるので、グローバルな関数として記述する場合は friend 関数にする必要がある。同じ関数についてのコードが離れた位置に記述されるのはコードの確認や編集をする際に確認漏れや修正漏れが発生する可能性が高くなるので、メンバ関数として実装した。

+,-,*,/のオーバーロードを実装し始めた当初、まずメンバ関数として以下のようなオーバーロードの仕方を検討した。

```
class Fraction {
public:
    //四則演算
    void operator+(const Fraction&) const;
    void operator-(const Fraction&) const;
    void operator*(const Fraction&) const;
    void operator/(const Fraction&) const;
};
```

この実装だと左オペランドの型が Fraction でない場合に対応できない問題があり、メンバ関数ではなくグローバルな関数として例えば以下のようなオーバーロードを追加する必要がある。

```
void operator+(const Int, const Fraction&);
void operator-(const Int, const Fraction&);
void operator*(const Int, const Fraction&);
```

```
void operator/(const Int, const Fraction&);
```

上記のように書いた場合、同じ演算子のオーバーロードが複数必要になる事、それらが記述される位置が離れる事になり、これも前述した問題が発生する可能性が高くなる。なので、仮にグローバルな関数として以下のように定義しメンバ関数としてオーバーロードしないようにしたとする。

```
void operator+(const Fraction&, const Fraction&);
void operator-(const Fraction&, const Fraction&);
void operator*(const Fraction&, const Fraction&);
void operator/(const Fraction&, const Fraction&);
```

このように定義すると左オペランドが Fraction でない場合も暗黙的型変換によって対応される為、同じ演算子のオーバーロードが複数必要になる問題が解消され、それによってこれらの演算子の記述もこれのみで完結させることができるので最終的にこの定義の仕方を採用した。

const 修飾子

関数の const 修飾子の有無と、引数の const 修飾子の有無についての考察を以下に記述する。

両者ともなるべく const にし得る箇所は const に出来るようにした。単純にヒューマンエラーを回避するためである。その結果メンバ変数の変更をしない関数については const 修飾子を付け、そうでない関数は付けなかった。

引数の const 修飾子については、型を参照型にするかどうかの話を交えての考察を書く。参照型の引数は const にし得る場合はすべて const 修飾子を付けた。参照型の変数の値が関数内で書き換わり関数の呼び出し元の処理に影響を与える可能性がない事を明示し関数の利用者がその関数を使用するか否かを判断し易くなる為このようにした。

当初、参照型にする必要がない引数は参照型にせず引数を定義していた。この時、関数が呼び出されたタイミングで実引数のコピーが一時的に生成され関数の処理が済めば解放され、関数の呼び出し元の処理の流れに const の有無が影響を与えないので記述するメリットが薄いと判断して記述していなかった。ただし、参照型で引数を定義すれば変数の複製が発生しない分実行速度が速くなるメリットがある事も自覚していた。これを理由に参照型にする必要性がない場合にも参照型で定義する事も考えたが、参照型にする必要がある場合にのみ参照型で定義した方が関数の挙動を判断しやすい場合がありそうだと考え参照型にする必要がない場合は参照型にはしていなかった。どちらの選択をするかは決め手になる理由を見つけられず、C++を書いているエンジニアに一般的にこういった場合にはどう記述される文化があるかを聞いた所、「参照型に出来る箇所は全て参照型にし、const にできる箇所は全て const にするのが恐らく一般的」という回答を得たので最終的にはこれに従った。確かにこの記述は処

理のパフォーマンスにおいて最良であり且つヒューマンエラーを防止するので客観的には最善な記述ではあると考えた。

-