Labb 2 Rational

Uppgift: Implementera rationella tal som en generisk class Rational som tar heltalstypen som template parameter.

Denna laboration går ut på att exemplifiera
- Templates
- operator over loading
- typ konvertering (och de tvetydighets problem som kan uppstå)
- Traits (för VG)

vi kommer också att lära oss lite om de olika heltalen i C++

Målet är att t.ex. Rational<int> ska fungera så likt de inbyggda talen som möjligt (utan att vi arbetar ihjäl oss).

Rationella tal:

Rationell tal är bråk dvs. ett heltal dividerat med ett annat heltal. T.ex. 3/17 som är knappt 0,2.
Beräkningar sker utan avrundningar så 3/17+5/4 = (3*4+5*17)/(17*4) = 97/68.
Det är lämpligt att man alltid förenklar så mycket som möjligt. T.ex. så är 1/2 + 1/6 = (1*6+1*2)/(2*6) = 8/12 vilket kan förenklas till 2/3. För förenkling så används största gemensamma divisor algoritmen (GCD) som finns given i filen GCD.h, där även Reduce är definierad.
T.ex. så är GCD(8, 12) = 4. Dividera 8 resp 12 med 4 och vi får 2 resp. 3.
Vi kan använda vilken heltalstyp som helst som grund för rationella tal och vi gör därför de rationella talen som en template typ med heltalstypen som template parameter.
Skiss på början av Rational klassen:

```
#include <iostream>
#include "GCD.h"

template<typename Tint>
class Rational {
public:
	Tint P, Q;
	friend std::ostream& operator<< (std::ostream & cout, Rational<Tint> R){
		cout << R.P << '/' << R.Q;
		return cout;
	}

	Rational(): P(0), Q(1) {};
	Rational(Tint P):P(P), Q(1) {}
	Rational(Tint P, Tint Q):P(P), Q(Q) {
		Reduce(P, Q);
	}

	Rational operator+(const Rational rhs) const {
		Tint tempP=rhs.Q*P+Q*rhs.P;
		Tint tempQ=Q*rhs.Q;
		return Rational(tempP, tempQ);
	}
};
```

Hjälp:

Filen GCD.h innehåller GCD och Reduce.
Filen RelOps.h gör att det räcker att definiera operator== (resp. operator<).
Filen Main.h innehåller ett testprogram som jag själv använt. Observera att det kan finnas saker som ni kanske måste ändra för att få det att fungera med era tal.

Typkonverteringsproblemet:

Vi antar att vi kan konvertera från int till Rat - t.ex att vi har konstruktorn ”Rat(int i) {…”
Sen så har vi en funktion (eller operator) void Foo(Rat a, Rat b).
Om vi sedan anropar den med Foo(myRat, 1.0) så kommer 1.0 att konverteras till int och sedan till Rat och allt går bra.

Om vi nu lägger till en till konstruktor, t.ex ”Rat(short i) {…” och igen gör anropet Foo(myRat, 1.0) så får vi kompileringsfel, det går att konvertera 1.0 till antingen int eller short och kompilatorn vet inte vilket.
Observera att det fortfarande går bra att skriva Foo(myRat, 1) passar 1 perfekt med int.

Lösningen blir att vi antingen får nöja oss med att göra en konstruktor från ett av de inbyggda taltyperna eller göra konstruktorer för alla av de inbyggda taltyperna som vi vill vara kompatibla med.

Krav för G

Rational har alla de vanliga taltyperna som ”nära” klasser men de är väldigt många så vi nöjer oss med att försöka få Rational att fungera vettigt med

- short, int, long long som parameter (Rational<short> etc.)
- short, int, long long ska den kunna “samarbeta” med.

Precis som för de inbyggda operationerna så bryr vi oss inte om overflow och andra fel.

”Rtal” står för någon av typerna Rational<short>, Rational<int> eller Rational<long long>.
”rtal” står för något tal av typen ”Rtal”.
”Tal” står för någon av typerna Rational<short>, Rational<int>, Rational<long long>, short, int, eller long long. ”tal” står för något tal av typen ”Tal”.

Rational<Tint> ska kunna

- Konstrueras från ”Tal” dvs. `Rtal rtal(tal);`
- Jämföras med == dvs. `if (rtal == tal) …`
- Tilldelas (=) från ”Tal” dvs. `rtal=tal;`
- += med ”Tal” dvs. `rtal += tal;`
- + dvs. `(rtal + tal)`
- unärt ”–” dvs. `rtal1 = -rtal2;`
- båda ++ operatorerna, dvs. `++rtal; rtal++;`
- explicit konvertering till Tal. (Kräver VS2012 och kompilator CTP november 12).
- Overloading av `<<` och `>>` (ut och in matning): En bra designprincip är att om man läser in det man skrev ut så får man samma tal som resultat.

Kommentarer och tips:

- Skriv inte saker i onödan, t.ex. så kan ni fundera på om de inbyggda copy constructor och copy assignment operator inte duger som de är. Den inbyggda default constructorn

duger men går tyvärr inte att använda då den Visual Studio inte klarar att man skriver:

```
Rational() = default;  //C++11 konstruktion som inte finns i VS2012
```

- Se först till att det finns konstruktorer från alla ”Tal” till Rational (går att skriva med templates). Detta gör att man inte behöver skriva alla versioner av operatorerna utan att det räcker att göra t.ex. operator+ som en medlemsfunktion. Kompilatorn kommer vid behov att konvertera till rätt typ mha. de konstruktorer vi skrivit.
- För bara G så kan man nöja sig med att göra alla operatorer som medlemsfunktioner i klassen Rational.
- operator+ och operator += gör nästan samma sak. En allmän tumregel är att implementera += först och använda den för att göra +, det är dock inte alltid det bästa utan ibland så måste man skriva en separat + operator.
- ```
  friend std::ostream& operator<< (std::ostream & cout, Rational<Tint> R){
      . . . //Koden här
  }
  ```
  deklarerat inuti Rational är det enklaste sättet att gör friend funktioner.

Semantik för klassen.

Rational<int> är lika med ett bråk P/Q där P och Q är heltal.
Det är enklast om man ser till att Q alltid är positivt (P/-Q=-P/Q) och att man alltid har P och Q reducerade dvs de kan inte delas med samma tal: vi har aldrig 6/2 utan det blir 3/1.

(a/b) + (c/d) = (a*d + c*b) / (b*d)

(a/b) * (c/d) = (a*c)/b*d)

Kod för GCD (Greatest Common Divisor) och “Reduce” finns på Its ”GCD.h”
Kod som underlättar logiska operatorer finns i ”RelOps.h”

Krav för VG

När man beräknar t.ex. + så är det lätt att mellanresultaten blir för stora så att man får fel reultat, de är därför lämpligt att arbeta med större tal under beräkningarna. Man behöver då kunna få tag på vilken typ som är ”Nästa” i storleksordning. Används ”Traits” för att lösa detta – svårt hitta bra text om traits, jag hänvisar till min föreläsning.
Observera att man måste ta hänsyn till de exakta storlekarna på heltalen för att kunna göra detta.

Se till att operator+ fungerar även för ”int+Rational<long long>” etc. Den ska ge resultat som stämmer med den ”största” type:

- ”int+Rational<long long>” ska ge en ”Rational<long long>” som resultat.
- ”int+Rational<short>” ska ge en ”Rational<int>” som resultat.
- ”Rational<short> + Rational<long long>” ska ge en ”Rational<long long>” som resultat.

Även detta löses med Traits.

För VG krävs även att programmet inte är långsamt (t.ex. gör onödiga konverteringar och kopieringar), att det har const (och &) på rätt ställen samt är snyggt skrivet.
  ```