

## Labb 2: Rational Numbers

**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 överdriva).

### Rationella tal:

Rationella tal är bråk dvs. ett heltal dividerat med ett annat heltal. T.ex.  $3/17 \approx 0,2$ .

Beräkningar sker utan avrundningar:  $3/17 + 5/4 = (3*4 + 5*17)/(17*4) = 97/68$ .

Det är lämpligt att alltid förenkla 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  $\text{GCD}(8, 12) = 4$ . 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(): nom(0), denom(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: {
...
};
```

Observera att ni inte får använda STL eller decltyp eller liknande för att konstruera era klasser utan ni måste göra allt själva.

## Hjälp filer:

VG.h innehåller dels en define för VG men också en möjlighet att införa ett alias så ni kan använda testprogrammet även om ni döpt er klass till något annat än Rational.

GCD.h innehåller GCD och Reduce.

RelOps.h gör att det räcker att definiera operator== (resp. operator<).

RatTest.cpp är testprogrammet.

## Typkonverteringsproblem:

Det finns en risk för att vi kan konvertera saker på flera sätt. T.ex om vi har en funktion

```
void Foo(Rational<int> a, Rational<int> b);
```

och vi anropar den även med andra tal, t.ex:

```
Foo(Rational<int> R(1, 2), (short)3);
```

I C++ stil ska vi här konvertera 3 till ett rationellt så vi kan anropa funktionen. Problemet som ofta dyker upp är att det kan finnas flera sätt att göra denna omvandling och kompilatorn kan ha problem att hitta ett entydigt sätt att göra det. Om det till exempel finns konstruktorer som gör en Rational av både en short int och int, så kan kompilatorn få problem att hitta en bästa väg att konvertera från short int till Rational<int>.

## 1 Krav för G

Rational borde fungera tillsammans med alla tal på samma sätt som de inbyggda talen 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".

### 1.1 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.
- 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.

## 1.2 Kommentarer och tips:

- Skriv inte saker i onödan, t.ex. så kan ni fundera på om de automatgenererade konstruktörerna inte fungerar bra som de är.
- Se först till att det finns konstruktörer 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 (för betyg G).
- 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 göra friend funktion.

## 1.3 Semantik för klassen.

Rational<int> är lika med ett bråk P/Q där P och Q är heltal.

Det är enklast att alltid ha Q 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 ITSL "GCD.h"

Kod som underlättar logiska operatorer finns i "RelOps.h"

## 2 Krav för VG

### 2.1 Resultat typ av operator+

De inbyggda aritmetiska operatorerna ger alltid ett resultat som är av den "största" sorten t.ex. om man gör `LL + I` där LL är long long och I är int så kommer detta att resultera i en long long. Ert program ska fungera på samma sätt;

```
Rational<short> rs;
Rational<int> ri;
auto x = rs + ri; //x ska då få typen Rational<int>
```

även `rs+1` så ska det få typen Rational<int>

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

### 2.2 Typen som används för beräkning av mellanresultat.

När man beräknar t.ex. addition så är det lätt att mellanresultat blir för stora så att man får fel resultat, de är därför lämpligt att arbeta med större tal under beräkningarna. Därför måste du arbeta med större siffror under beräkningarna. Man behöver då kunna få tag på vilken typ som är "Nästa" i storleksordning. Använd "Traits" för att lösa detta – svårt hitta bra text om traits, jag hänvisar till min föreläsning.

### 2.3 Kompabilitet

Binära operatorer ska fungera även med ett heltal först. T.ex. `1==Rational<int>(3,2)` ska bli false.

## **2.4 Effektivitet**

Det ska inte ske onödiga konverteringar till Rational, t.ex. vid beräkningen av `1==Rational<int>(3, 2)`.

## **2.5 Snyggt program**

För VG krävs att programmet har ”const” och ”&” på rätt ställen samt är snyggt skrivet.