Contents

1 The RationalImpl<T> Class

The RationalImpl<T> Class is a rational (fractions) class.

1.1 Using RationalImpl<T>

Its very simple to use. Rational(1,2) will be a rational with data as the type std::ptrdiff_t. Using rational with a data type that cannot be used with the unary - operator is undefined.

Example 1: Creating organized rationals with RationalImpl<T>

```
1 cg::RationalImpl<int16_t> r(4, 8); //numerator and denominator are
      type int16_t.
2 cg::Rational r2(4, 8);//numerator and denominator are type std::
      ptrdiff_t.
3 r2.AutoSimp(true); //Will also simplify automatically.
```

Creating the RationalImpl<T> and using the r.AutoSimp(true) will cause it to attempt to simplify the object after each operation. It is recomended to use typedef or using declarations for this class (such as using BigRational = RationalImpl<SomeBigNumClass>;

Why?
The reason for this classes existance is that some developers may want floating point number that is 100% accurate and can be used without having to deal with the rounding of floats and double. Combined with a int64_t data type, The rational object can be a fractional value with 100% accuracy as far as the datatype can handle. **Bigger user defined datatypes may also be used as long as they have the operators for math defined.**

1.2 From double to RationalImpl<T>

Creating a RationalImpl<T> from a double value is easy.

Example 2: Creating organized rationals with RationalImpl<T>

```
1 auto r = cg::Rational::Make(0.444329, 9); //9 is the amount of
      decimals to keep. r == (444329000 / 1000000000)
2 r.AutoSimp(true); // autosimp is now on: r == (444329 / 1000000)
```

The simplification of the rational uses the euclidean GCD algorithm, and the LCM algorithm uses the GCD to speed things up, so both use euclidean algorithms.

1.3 Inverses of RationalImpl<T>

The RationalImpl<T> class has 4 functions dealing with inverses. AInverse(), Opposite() and MInverse(), Reciprocal().

Example 3: Creating organized rationals with RationalImpl<T>

```
cg::Rational r2(4, 8);
auto X = r2.Reciprocal(); //X now = rational of (8/4).
r2.MakeReciprocal();      //r2 now is (8/4).
auto Y = r2.Opposite();   //Y is now -(8/4).
r2.MakeOpposite();        //r2 is now -(8/4).
r2.Simplify();            //r2 is now -(2/1).
```

For the rational number, MInverse() is the same as Reciprocal() and AInverse() is the same as Opposite(). Each function has a version that is r.Make*** where the stars are Opposite, Reciprocal, AInverse, MInverse.

1.4 Scaling Up and Down

The rational can also be scaled, which is multiplying or dividing the numerator and denominator by the same value.

Example 4: Creating organized rationals with RationalImpl<T>

```
cg::Rational r2(4, 8);
r2.ScaleUp(3);          //r2 = (12, 24)
r2.ScaleDown(3);        //r2 = (4, 8) .... returns true
r2.ScaleDown(3);        //return false, nothing done, 3 is not
    evenly divisable by the numerator and denominator.
```

If the values are not divisible by the ScaleDown() argument, then nothing happens and false is returned. If they are divisible (both of them), they are scaled down and true is returned.

1.5 Math with RationalImpl<T>

All math and comparisons are overloaded.

Example 5: Creating organized rationals with RationalImpl<T>

```
cg::Rational r2(4, 8);
cg::Rational r3(4, 9);
auto X = r2 (+,-,*,/) r3; //Works as expected
auto Y = r2 (<,<=, ==, !=, >, >=) r3 //Works as expected.
r2 (+=, -=, *=, /=) r3;    //Works as expected.
r2 (+=, -=, *=, /=) 97;    //Works as expected.
++r2; r2++; --r2; r2--; -r2; //All works as expected.
```

All operators except the modulo operator will work properly.

1.6 Evaluation

The `RationalImpl<T>` class can be evaluated to another number, provided that the data type can take a constructor of type T, and T has the /= operator working properly.

Example 6: Creating organized rationals with `RationalImpl<T>`

```
1 cg::Rational r2(4, 8);
2 auto someBigDecimal = r2.Eval<SomeBigDecimalType>();
```

The accuracy of the Eval functions is only as good as the /= operator for the type given to it.