Programming in Modern C++

Tutorial T07: How to design a UDT like built-in types?: Part 1: Fraction UDT

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Tutorial Objectives

- Understand Building a data type: Fraction type

Tutorial Outline

Data types

Data types

Notion of Data types

- Data types in C++ are used to specify the type of the data we use in our programs.
- They are classified under three categories:
  - Built-In or Primitive Data types
  - Derived Data types
  - User-Defined Data type
- **Built-in data types:**
  - Built-in data types are the most basic data types in C++
  - They are predefined and can be used directly in a program
  - **Examples:** `char`, `int`, `float` and `double`
  - Apart from these, we also have `void` and `bool` data types
- **Derived Data types:**
  - Data types that are derived from the built-in types
  - **Examples:** arrays, functions, references and pointers
- **User Defined Type (UDT):**
  - Those are declared & defined by the user using basic data types before using it
  - **Examples:** structures, unions, enumerations and classes

User Defined Types

- Operator overloading helps us *build complete algebra* for UDT's much in the same line as is available for built-in types, called as, *Building data type*
  - **Complex type**: Add (+), Subtract (-), Multiply (*), Divide (/), Conjugate (!), Compare (==, !=, ...), etc.
  - **Fraction type**: Add (+), Subtract (-), Multiply (*), Divide (/), Reduce (unary *), Compare (==, !=, ...), etc.
  - **Matrix type**: Add (+), Subtract (-), Multiply (*), Divide (/), Invert (!), Compare (==, !=, ...), etc.
  - **Set type**: Union (+), Difference (-), Intersection (*), Subset (<, <=), Superset (>, >=), Compare (==, !=), etc.
  - **Direct IO**: read (>>) and write (<<) for all types

Fraction UDT

Design of Fraction UDT

- We intend to design a UDT `Fraction` which can behave like the build-in types like `int`
- The broad tasks involved include:
  - Make a clear statement of the concept of `Fraction`
  - Identify a representation for a `Fraction` object
  - Identify the properties and assertions applicable to all objects
  - Identify the operations for `Fraction` objects
    - Choose appropriate operators to overload for the operations
    - For example `operator+` to add two `Fraction` objects, or `operator<<` to stream a `Fraction` to `cout`
    - *Do not break the natural semantics for the operators*
- While it is possible to design and implement the UDT in one go (once you have acquired some expertise); it is better to go with iterative refinement. That is:
  - Make a design
  - Implement and Test
  - Refine and repeat

Notion of Fraction

- Intuitively fraction is a notation for numbers of the form $\frac{p}{q}$ where p and q are integers, like $\frac{2}{3}$, $\frac{4}{6}$, $\frac{3}{2}$ etc.
  - Fraction representation is *non-unique*: $\frac{2}{3} = \frac{4}{6} = \frac{8}{12} = \frac{-2}{-3}; \cdots, -\frac{2}{3} = \frac{-2}{3} = \frac{2}{-3}$
- For our UDT design, we need *uniqueness of representation*. So let us restrict with the following rules for a fraction $\frac{p}{q}$:
  - q must be *positive*: $q > 0$
  - p and q must be *mutually prime*: $gcd(p, \ q) = 1$

  Such fractions are known as *rational numbers* in mathematics
- Further a fraction $\frac{p}{q}$ is called *proper* if $|\frac{p}{q}| < 1$. It is *improper*, otherwise
  - An *improper fraction* can be written in *mixed fraction format* (assume $p > 0$) where we specify the maximum whole number in the fraction and the remaining proper fraction part:

$$\frac{p}{q} = (p \div q)\frac{p \ \% \ q}{q}$$

    For example, $\frac{17}{3} = 5\frac{2}{3}$

Definition of Fraction

Definition

$\frac{p}{q}$ is a fraction where p and q are integers, $q > 0$, and p and q are mutually prime, that is, $gcd(p, q) = 1$

That is, $p \in \mathcal{Z}, q \in \mathcal{N}$, $gcd(p, q) = 1$, where $\mathcal{Z}$ is the set of integers and $\mathcal{N}$ is the set of natural numbers

p is called the numerator and q is called the denominator

Definition

Any fraction $\frac{p}{q}$ where $gcd(p, q) > 1$, is irreduced and can be reduced to

$$\frac{p}{q} = \frac{p \div gcd(p, q)}{q \div gcd(p, q)}$$

Operations of Fraction

Definition

Reduction:

$$
\begin{aligned}
\frac{p}{q} &= \frac{p/gcd(p,q)}{q/gcd(p,q)}, if\ \ gcd(p,q) \neq 1 \\
&= \frac{-p}{-q}, if\ \ q < 0 \\
&= \frac{0}{1}, if\ \ p = 0 \\
&= undefined, if\ \ q = 0
\end{aligned}
$$

Addition: $(\frac{p}{q}) + (\frac{r}{s}) = \frac{p*(lcm(q,s)/q)+r*(lcm(q,s)/s)}{lcm(q,s)}$. Example 1: $\frac{5}{12} + \frac{7}{18} = \frac{5*3+7*2}{36} = \frac{29}{36}$

Subtraction: $(\frac{p}{q}) - (\frac{r}{s}) = (\frac{p}{q}) + (\frac{-r}{s})$. Example 2: $\frac{5}{12} - \frac{7}{18} = \frac{5*3+(-7)*2}{36} = \frac{1}{36}$

Multiplication: $(\frac{p}{q}) * (\frac{r}{s}) = \frac{p*r}{q*s}$. Example 3: $\frac{5}{12} * \frac{7}{18} = \frac{5*7}{12*18} = \frac{35}{216}$

Division: $(\frac{p}{q})\ /\ (\frac{r}{s}) = \frac{p*s}{q*r}$. Example 4: $\frac{5}{12} / \frac{7}{18} = \frac{5*18}{7*12} = \frac{15}{14}$

Modulus: $(\frac{p}{q})\ \%\ (\frac{r}{s}) = \frac{p}{q} - \lfloor(\frac{p}{q})\ /\ (\frac{r}{s})\rfloor * \frac{r}{s}$. Example 5: $\frac{5}{12} \% \frac{7}{18} = \frac{5}{12} - \lfloor\frac{15}{14}\rfloor * \frac{7}{18} = \frac{1}{36}$

where $lcm(q,s) = (q*s)/gcd(q,s)$

Rules of Fraction

Fractions obey fives rules of algebra as follows. For two fractions $\frac{p}{q}$ and $\frac{r}{s}$,

Definition

Rule of Invertendo: $\frac{p}{q} = \frac{r}{s} \Rightarrow \frac{q}{p} = \frac{s}{r}$. Use $!\frac{p}{q} = \frac{q}{p}$

Rule of Alternendo: $\frac{p}{q} = \frac{r}{s} \Rightarrow \frac{p}{r} = \frac{q}{s}$

Rule of Componendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p+q}{q} :: \frac{r+s}{s}$. Use $++\frac{p}{q} = \frac{p+q}{q} = \frac{p}{q} + 1$

Rule of Dividendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p-q}{q} :: \frac{r-s}{s}$. Use $--\frac{p}{q} = \frac{p-q}{q} = \frac{p}{q} - 1$

Rule of Componendo & Dividendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p+q}{p-q} :: \frac{r+s}{r-s}$

We define three operations on fractions: Invertendo (`operator!`), Componendo (`operator++`), and Dividendo (`operator--`) to facilitate fraction algebra expressions

Design of Fraction Class

- From the definition, the representation of a `Fraction` can simply be:

```
class Fraction { // Implicit assertion for proper fraction: gcd(|n_|, d_) = 1
    int n_;          // numerator. n_ belongs to Z
    unsigned int d_; // denominator. d_ belongs to N
}
```

- `Fraction` should support the following operation like `int`:
  - Construction, Destruction and Copy Operations
  - Unary Arithmetic Operations: Preserve (Sign), Negate, Componendo, and Dividendo
  - Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Mod
  - Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq
  - IO Operations: Read and Write
- `Fraction` should also support the following extended operation:
  - Invert
  - Convert to `double`
- `Fraction` also need to support the following utilities for convenience:
  - GCD and LCM
  - Reduction (of irreduced fraction to reduced fraction)

Design of Fraction: Interface: Version 1

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int = 1, int = 1);  // Three overloads including a default constructor
~Fraction();                          // No virtual destructor needed
Fraction(const Fraction&);            // Copy constructor
Fraction& operator=(const Fraction&); // Copy assignment operator
```

- IO Operations: Read and Write

```
static void Write(const Fraction&); // Outstreams a fraction to cout in n/d form
static void Read(Fraction&);        // Instreams n & d from cin to construct a fraction
```

- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
Fraction  Negate() const;   // Negate. p/q <-- -p/q
Fraction  Preserve() const; // Preserve. p/q <-- p/q
Fraction& Componendo();     // Componendo. p/q <-- p/q + 1
Fraction& Dividendo();      // Dividendo. p/q <-- p/q - 1
```

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction Add(const Fraction&) const;      // Generates a result fraction,
Fraction Subtract(const Fraction&) const; // Does not change the current object
Fraction Multiply(const Fraction&) const;
Fraction Divide(const Fraction&) const;
Fraction Modulus(const Fraction&) const;
```

Design of Fraction: Interface: Version 1

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool Eq(const Fraction&) const;     // Generates a comparison result
bool NotEq(const Fraction&) const; // Does not change the current object
bool Less(const Fraction&) const;
bool LessEq(const Fraction&) const;
bool More(const Fraction&) const;
bool MoreEq(const Fraction&) const;
```

- Extended Operations: Invert and Convert to `double`

```
Fraction Invert() const; // Inverts a fraction. !(p/q) = q/p
double Double();         // Converts a fraction to a double
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO;  // Defines 0/1
```

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int, int); // Finds the gcd for two +ve integers
static int lcm(int, int); // Finds the lcm for two +ve integers
Fraction& Reduce();        // Reduces a fraction
```

Implementation of Fraction: Version 1

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int n = 1, int d = 1): // Three overloads
    n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ is unsigned int. So no -ve value
{ Reduce(); } // Reduces the fraction
Fraction(const Fraction& f) : n_(f.n_), d_(f.d_) { } // Copy Constructor
~Fraction() { } // No virtual destructor needed
Fraction& operator=(const Fraction& f) { n_ = f.n_; d_ = f.d_; return *this; }
```

- IO Operations: Read and Write

```
static void Write(const Fraction& f) { cout << f.n_;
    if ((f.n_ != 0) && (f.d_ != 1)) cout << "/" << f.d_; // Suppress denominator
                                                         // if n_ == 0 or d_ == 1
}
static void Read(Fraction& f) { cin >> f.n_ >> f.d_; f.Reduce(); }
```

- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
Fraction  Negate() const   { return Fraction(-n_, d_); }
Fraction  Preserve() const { return *this; }
Fraction& Componendo()     { return *this = Add(Fraction::UNITY); }
Fraction& Dividendo()      { return *this = Subtract(Fraction::UNITY); }
```

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction Add(const Fraction& f2) const {
    unsigned int d = Fraction::lcm(d_, f2.d_);
    int n = n_*(d / d_) + f2.n_*(d / f2.d_);
    return Fraction(n, d);
}
Fraction Subtract(const Fraction& f2) const { return Add(f2.Negate()); }
Fraction Multiply(const Fraction& f2) const {
    return Fraction(n_*f2.n_, d_*f2.d_);
}
Fraction Divide(const Fraction& f2) const { return Multiply(f2.Invert()); }
Fraction Modulus(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    Fraction tf = Divide(f2);
    Fraction f = Subtract(Fraction(static_cast<int>(tf.n_ / tf.d_)).Multiply(f2));

    return f;
}
```

Implementation of Fraction: Version 1

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool Eq(const Fraction& f2) const     { return ((n_ == f2.n_) && (d_ == f2.d_)); }
bool NotEq(const Fraction& f2) const  { return !(Eq(f2)); }
bool Less(const Fraction& f2) const   { return Subtract(f2).n_ < 0; }
bool LessEq(const Fraction& f2) const { return !More(f2); }
bool More(const Fraction& f2) const   { return Subtract(f2).n_ > 0; }
bool MoreEq(const Fraction& f2) const { return !Less(f2); }
```

- Extended Operations: Invert and Convert to `double`

```
Fraction Invert() const { // Inverts a fraction. !(p/q) = q/p
    if (d_ == 0)  throw "Divide by 0 is undefined\n";
    return Fraction(d_, n_);
}
double Double() const { // Converts to a double
    return static_cast<double>(n_) / static_cast<double>(d_);
}
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO;  // Defines 0/1
```

Implementation of Fraction: Version 1

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int a, int b) { // Finds the gcd for two +ve integers
    while (a != b)
        if (a > b) a = a - b;
        else b = b - a;
    return a;
}
static int lcm(int a, int b) { // Finds the lcm for two +ve integers
    return (a / gcd(a, b))*b;
}
Fraction& Reduce() { // Reduces a fraction
    if (d_ == 0) { throw "Fraction with Denominator 0 is undefined"; }
    if (d_ < 0) { n_ = -n_;
        d_ = static_cast<unsigned int>(-static_cast<int>(d_));
        return *this;
    }
    if (n_ == 0) { d_ = 1; return *this; }
    unsigned int n = (n_ > 0) ? n_ : -n_, g = gcd(n, d_);
    n_ /= static_cast<int>(g); // as n_ is int and g is unsigned int the division may not work
    d_ /= g;
    return *this;
}
```

Testing Fraction: Version 1: Test Application


```
#include <iostream>
using namespace std;
#include "Fraction.h"

int main() {
    cout << "Construction, Copy Operations and Write Test" << endl; // Ctor, Copy & Write Test
    Fraction f1(5, 3); cout << "Fraction f1(5, 3) = "; Fraction::Write(f1); cout << endl;
    Fraction f2(7);    cout << "Fraction f2(7) = "; Fraction::Write(f2); cout << endl;
    Fraction f3;       cout << "Fraction f3 = "; Fraction::Write(f3); cout << endl;
    Fraction f4(f1);   cout << "Fraction f4(f1) = "; Fraction::Write(f4); cout << endl;
    Fraction f5(3, 6); cout << "Fraction f5(3, 6) = "; Fraction::Write(f5); cout << endl;
    Fraction f6(0, 4); cout << "Fraction f6(0, 4) = "; Fraction::Write(f6); cout << endl;
    cout << "Assignment: f2 = f1: f2 = "; Fraction::Write(f2 = f1); cout << endl << endl;

    cout << "Read Test" << endl; // Read Test
    cout << "Read f1 = "; Fraction::Read(f1); Fraction::Write(f1); cout << endl << endl;

    f1 = Fraction(2, 5); /* Using f1 for the following tests */ f2 = f1; // Copy to restore f1 later
    cout << "Unary Ops Test: Using f1 = ";
    Fraction::Write(f1); cout << " for all" << endl; // Unary Operations Test
    cout << "Negate: f1.Negate() = "; Fraction::Write(f1.Negate()); cout << endl;
    cout << "Preserve: f1.Preserve() = "; Fraction::Write(f1.Preserve()); cout << endl;
    cout << "Componendo: f1.Componendo() = "; Fraction::Write(f1.Componendo()); cout << endl; f1 = f2;
    cout << "Dividendo: f1.Dividendo() = "; Fraction::Write(f1.Dividendo()); cout << endl << endl;
```

Testing Fraction: Version 1: Test Application


```
f1 = Fraction(5, 12); f2 = Fraction(7, 18); // Using f1 and f2 for the following test
cout << "Binary Ops Test: Using f1 = "; // Binary Operations Test
Fraction::Write(f1); cout << ". f2 = "; Fraction::Write(f2); cout << " for all" << endl;
cout << "Binary Plus: f1.Add(f2) = "; Fraction::Write(f1.Add(f2));  cout << endl;
cout << "Binary Minus: f1.Subtract(f2) = "; Fraction::Write(f1.Subtract(f2));  cout << endl;
cout << "Binary Multiply: f1.Multiply(f2) = "; Fraction::Write(f1.Multiply(f2));  cout << endl;
cout << "Binary Divide: f1.Divide(f2) = "; Fraction::Write(f1.Divide(f2));  cout << endl;
cout << "Binary Residue: f1.Modulus(f2) = "; Fraction::Write(f1.Modulus(f2));  cout << endl << endl;

// Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
cout << "Logical Ops Test: Using f1 = "; // Logical Operations Test
Fraction::Write(f1); cout << ". f2 = "; Fraction::Write(f2); cout << " for all" << endl;
cout << "Equal: " << ((f1.Eq(f2)) ? "true" : "false") << endl;
cout << "Not Equal: " << ((f1.NotEq(f2)) ? "true" : "false") << endl;
cout << "Less: " << ((f1.Less(f2)) ? "true" : "false") << endl;
cout << "Less Equal: " << ((f1.LessEq(f2)) ? "true" : "false") << endl;
cout << "Greater: " << ((f1.More(f2)) ? "true" : "false") << endl;
cout << "Greater Equal: " << ((f1.MoreEq(f2)) ? "true" : "false") << endl << endl;

// Using f1 = Fraction(5, 12); for the following tests
cout << "Extended Ops Test: Using f1 = "; // Extended Operations Test
Fraction::Write(f1); cout << " for all" << endl;
cout << "Invert: f1.Invert() = "; Fraction::Write(f1.Invert()); cout << endl;
cout << "Double: f1.Double() = "; cout << f1.Double() << endl << endl;
```

Testing Fraction: Version 1: Test Application


```
    cout << "Static Constants Test" << endl; // Static Constants Test
    cout << "UNITY = "; Fraction::Write(Fraction::UNITY); cout << endl;
    cout << "ZERO = "; Fraction::Write(Fraction::ZERO); cout << endl << endl;
}
```

```
Construction, Copy Operations and Write Test
Fraction f1(5, 3) = 5/3
Fraction f2(7) = 7
Fraction f3 = 1
Fraction f4(f1) = 5/3
Fraction f5(3, 6) = 1/2
Fraction f6(0, 4) = 0
Assignment: f2 = f1: f2 = 5/3

Read Test
2 7
Read f1 = 2/7

Unary Ops Test: Using f1 = 2/5
Negate: f1.Negate() = -2/5
Preserve: f1.Preserve() = 2/5
Componendo: f1.Componendo() = 7/5
Dividendo: f1.Dividendo() = -3/5
```

All tests passed

```
Binary Ops Test: Using f1 = 5/12. f2 = 7/18 for all
Binary Plus: f1.Add(f2) = 29/36
Binary Minus: f1.Subtract(f2) = 1/36
Binary Multiply: f1.Multiply(f2) = 35/216
Binary Divide: f1.Divide(f2) = 15/14
Binary Residue: f1.Modulus(f2) = 1/36

Logical Ops Test: Using f1 = 5/12. f2 = 7/18 for all
Equal: false
Not Equal: true
Less: false
Less Equal: false
Greater: true
Greater Equal: true

Extended Ops Test: Using f1 = 5/12 for all
Invert: f1.Invert() = 12/5
Double: f1.Double() = 0.416667

Static Constants Test
UNITY = 1
ZERO = 0
```

Fraction: Version 1

- So now we have one design and implementation for `Fraction` objects that can be manipulated by various operation member functions
- However, it still leaves a lot more to be desired. Consider, that we want to evaluate the following fraction expression:

$$
\begin{aligned}
f1 &= \frac{2}{3} \\
f2 &= \frac{8}{1} \\
f3 &= \frac{5}{6} \\
f4 &= (f1 + f2)/(f1 - f2) + !f3 - f2 * f3 = -\frac{1097}{165}
\end{aligned}
$$

- Using Version 1:

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;

    f4 = f1.Add(f2).Divide(f1.Subtract(f2)).Add(f3.Invert()).Subtract(f2.Multiply(f3));
    Fraction::Write(f4); cout << endl;
}
```

- *Horrendously complicated and error-prone, to say the least*
- To simplify, we map the member functions to various overloaded operators in Version 2

Design of Fraction: Interface: Version 2

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int = 1, int = 1);  // Three overloads including a default constructor
~Fraction();                          // No virtual destructor needed
Fraction(const Fraction&);            // Copy constructor
Fraction& operator=(const Fraction&); // Copy assignment operator
```

- IO Operations: Read and Write (friend function needed for iostream support)

```
friend ostream& operator<<(ostream&, const Fraction&); // Write()
friend istream& operator>>(istream&, Fraction&);       // Read()
```

- Unary Arithmetic Operations: Preserve (Sign), Negate, Componendo, and Dividendo. Postfix operators are additions here

```
Fraction  operator+() const; // Preserve()
Fraction  operator-() const; // Negate()
Fraction& operator++();      // Pre-increment. Componendo(): p/q <-- p/q + 1
Fraction& operator--();      // Pre-decrement. Dividendo(): p/q <-- p/q - 1
Fraction  operator++(int);   // Post-increment.
                             // Lazy Componendo. p/q <-- p/q + 1. Returns old p/q
Fraction  operator--(int);   // Post-decrement.
                             // Lazy Dividendo. p/q <-- p/q - 1. Returns old p/q
```

Design of Fraction: Interface: Version 2

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction operator+(const Fraction&) const; // Add()
Fraction operator-(const Fraction&) const; // Subtract()
Fraction operator*(const Fraction&) const; // Multiply()
Fraction operator/(const Fraction&) const; // Divide()
Fraction operator%(const Fraction&) const; // Modulus()
```

Since the constructor of `Fraction` is `explicit`, an `int` cannot be implicitly converted to `Fraction`. So we do not expect an addition operation like `i + f` where `int i;` and `Fraction`. Hence, member function operators are okay. Otherwise, we will need `friend` function operators:

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

Design of Fraction: Interface: Version 2

- Advanced Assignment Operators. These are additions here:

```
Fraction& operator+=(const Fraction&);
Fraction& operator-=(const Fraction&);
Fraction& operator*=(const Fraction&);
Fraction& operator/=(const Fraction&);
Fraction& operator%=(const Fraction&);
```

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool operator==(const Fraction&) const; // Eq()
bool operator!=(const Fraction&) const; // NotEq()
bool operator<(const Fraction&) const;  // Less()
bool operator<=(const Fraction&) const; // LessEq()
bool operator>(const Fraction&) const;  // More()
bool operator>=(const Fraction&) const; // MoreEq()
```

Design of Fraction: Interface: Version 2

- Extended Operations: Invert and Convert to `double`

```
Fraction operator!() const; // Invert()
operator double();          // Double()
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO;  // Defines 0/1
```

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int, int); // Finds the gcd for two +ve integers
static int lcm(int, int); // Finds the lcm for two +ve integers
Fraction& operator*();    // Reduce()
```

  Since reduction is not on the interface, we may not overload an operator for it - it will be fine to use the earlier `Reduce()` function

- *Note that Bit-wise operators, Shift operators etc. are not overloaded in* `Fraction` *since there is no semantic interpretation for them*

Implementation of Fraction: Version 2

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int n = 1, int d = 1): // Three overloads
    n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ is unsigned int. So no -ve value
{ *(*this); } // Reduces the fraction by operator*()
Fraction(const Fraction& f) : n_(f.n_), d_(f.d_) { } // Copy Constructor
~Fraction() { } // No virtual destructor needed
Fraction& operator=(const Fraction& f) { n_ = f.n_; d_ = f.d_; return *this; }
```

- IO Operations: Read and Write (`friend` function needed for `iostream` support)

```
friend ostream& operator<<(ostream& os, const Fraction& f) { os << f.n_;
    if ((f.n_ != 0) && (f.d_ != 1)) os << "/" << f.d_; // Suppress denominator
    return os;                                         // if n_ == 0 or d_ == 1
}
friend istream& operator>>(istream& is, Fraction& f) { is >> f.n_ >> f.d_;
    *f; /* Reduces the fraction by operator*() */ return is;
}
```

- Unary Arithmetic Operations: Preserve, Negate, Componendo, Dividendo, & Postfix operators:

```
    Fraction operator-() const { return Fraction(-n_, d_); }
    Fraction operator+() const { return *this; }
    Fraction& operator++()     { return *this += Fraction::UNITY; }
    Fraction& operator--()     { return *this -= Fraction::UNITY; }
    Fraction operator++(int)   { Fraction f = *this; ++*this; return f; }
    Fraction operator--(int)   { Fraction f = *this; --*this; return f; }
```

Implementation of Fraction: Version 2

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulo:

```
Fraction operator+(const Fraction& f2) const {
    unsigned int d = lcm(d_, f2.d_);
    int n = n_*(d / d_) + f2.n_*(d / f2.d_);
    return Fraction(n, d);
}
Fraction operator-(const Fraction& f2) const { return *this + (-f2); }
Fraction operator*(const Fraction& f2) const { return Fraction(n_*f2.n_, d_*f2.d_); }
Fraction operator/(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    return Fraction(n_*f2.d_, d_*f2.n_);
}
Fraction operator%(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    Fraction tf = (*this) / f2;
    return (*this) - Fraction(tf - Fraction(tf.n_ % tf.d_, tf.d_))*f2;
    // return (*this) - Fraction(static_cast<int>(tf.n_ / tf.d_))*f2; // As in Ver 1
}
```

Implementation of Fraction: Version 2

- Advanced Assignment Operators. These are additions here:

```
Fraction& operator+=(const Fraction& f) { *this = *this + f; return *this; }
Fraction& operator-=(const Fraction& f) { *this = *this - f; return *this; }
Fraction& operator*=(const Fraction& f) { *this = *this * f; return *this; }
Fraction& operator/=(const Fraction& f) { *this = *this / f; return *this; }
Fraction& operator%=(const Fraction& f) { *this = *this % f; return *this; }
```

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool operator==(const Fraction& f2) const { return ((n_ == f2.n_) && (d_ == f2.d_)); }
bool operator!=(const Fraction& f2) const { return !(*this == f2); }
bool operator<(const Fraction& f2) const  { return (*this - f2).n_ < 0; }
bool operator<=(const Fraction& f2) const { return !(*this > f2); }
bool operator>(const Fraction& f2) const  { return (*this - f2).n_ > 0; }
bool operator>=(const Fraction& f2) const { return !(*this < f2); }
```

Implementation of Fraction: Version 2

- Extended Operations: Invert and Convert to `double`

```
Fraction operator!() const { // Inverts a fraction. !(p/q) = q/p
    if (d_ == 0) { throw "Divide by 0 is undefined\n"; }
    return Fraction(d_, n_);
}
operator double() const { return static_cast<double>(n_) / static_cast<double>(d_); }
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO;  // Defines 0/1s
```

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int a, int b);
static int lcm(int a, int b);

Fraction& operator*() { // Reduces a fraction
    if (d_ == 0) { throw "Fraction with Denominator 0 is undefined"; }
    if (d_ < 0) { n_ = -n_; d_ = static_cast<unsigned int>(-static_cast<int>(d_)); return *this; }
    if (n_ == 0) { d_ = 1; return *this; }
    unsigned int n = (n_ > 0) ? n_ : -n_, g = gcd(n, d_);
    n_ /= static_cast<int>(g); // as n_ is int and g is unsigned int the division may not work
    d_ /= g;
    return *this;
}
```

Testing Fraction: Version 2: Pass Test Application

```
#include <iostream>
using namespace std;
#include "Fraction.h"
int main() {
    cout << "Construction, Copy Operations and Write Test" << endl; // Ctor, Copy & and Write Test
    Fraction f1(5, 3); cout << "Fraction f1(5, 3) = " << f1 << endl;
    Fraction f2(7);    cout << "Fraction f2(7) = " << f2 << endl;
    Fraction f3;       cout << "Fraction f3 = " << f3 << endl;
    Fraction f4(f1);   cout << "Fraction f4(f1) = " << f4 << endl;
    Fraction f5(3, 6); cout << "Fraction f5(3, 6) = " << f5 << endl;
    Fraction f6(0, 4); cout << "Fraction f6(0, 4) = " << f6 << endl;
    cout << "Assignment: f2 = f1: f2 = " << (f2 = f1) << endl << endl;

    cout << "Read Test" << endl; // Read Test
    cin >> f1; cout << "Read f1 = " << f1 << endl << endl;

    f1 = Fraction(2, 5); /* Using f1 for the following tests */ f2 = f1; // Copy to restore f1 later
    cout << "Unary Ops Test: Using f1 = " << f1 << " for all" << endl; // Unary Operations Test
    cout << "Negate: -f1 = " << -f1 << endl;
    cout << "Preserve: +f1 = " << +f1 << endl;
    cout << "Componendo: ++f1 = " << ++f1 << endl; f1 = f2;
    cout << "Dividendo: --f1 = " << --f1 << endl; f1 = f2;
    cout << "Lazy Componendo: f1++ = " << f1++ << " Lazy f1 = " << f1 << endl; f1 = f2;
    cout << "Lazy Dividendo: f1-- = " << f1-- << " Lazy f1 = " << f1 << endl << endl;
```

Testing Fraction: Version 2: Pass Test Application

```
f1 = Fraction(5, 12); f2 = Fraction(7, 18); // Using f1 and f2 for the following test

// Binary Operations Test
cout << "Binary Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
cout << "Binary Plus: f1 + f2 = " << (f1 + f2) << endl;
cout << "Binary Minus: f1 - f2 = " << (f1 - f2) << endl;
cout << "Binary Multiply: f1 * f2 = " << (f1 * f2) << endl;
cout << "Binary Divide: f1 / f2 = " << (f1 / f2) << endl;
cout << "Binary Residue: f1 % f2 = " << (f1 % f2) << endl << endl;

// Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
f3 = f1; // Copy to restore f1 later
// Binary Assignment Operations Test
cout << "Binary Assignment Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
cout << "Plus Assign: f1 += f2: f1 = " << (f1 += f2) << endl; f1 = f3;
cout << "Minus Assign: f1 -= f2: f1 = " << (f1 -= f2) << endl; f1 = f3;
cout << "Multiply Assign: f1 *= f2: f1 = " << (f1 *= f2) << endl; f1 = f3;
cout << "Divide Assign: f1 /= f2: f1 = " << (f1 /= f2) << endl; f1 = f3;
cout << "Residue Assign: f1 %= f2: f1 = " << (f1 %= f2) << endl << endl; f1 = f3;
```

Testing Fraction: Version 2: Pass Test Application


```
    // Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
    // Logical Operations Test
    cout << "Logical Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
    cout << "Equal: " << ((f1 == f2) ? "true" : "false") << endl;
    cout << "Not Equal: " << ((f1 != f2) ? "true" : "false") << endl;
    cout << "Less: " << ((f1 < f2) ? "true" : "false") << endl;
    cout << "Less Equal: " << ((f1 <= f2) ? "true" : "false") << endl;
    cout << "Greater: " << ((f1 > f2) ? "true" : "false") << endl;
    cout << "Greater Equal: " << ((f1 >= f2) ? "true" : "false") << endl << endl;

    // Extended Operations Test
    // Using f1 = Fraction(5, 12); for the following tests
    cout << "Extended Ops Test: Using f1 = " << f1 << " for all" << endl;
    cout << "Invert: !f1 = " << !f1 << endl;
    cout << "Double: (double)f1 = "; cout << static_cast<double>(f1) << endl << endl;

    // Static Constants Test
    cout << "Static Constants Test" << endl;
    cout << "UNITY = " << Fraction::UNITY << endl;
    cout << "ZERO = " << Fraction::ZERO << endl << endl;
}
```

Testing Fraction: Version 2: Pass Test Application


```
Construction, Copy Operations and Write Test
Fraction f1(5, 3) = 5/3
Fraction f2(7) = 7
Fraction f3 = 1
Fraction f4(f1) = 5/3
Fraction f5(3, 6) = 1/2
Fraction f6(0, 4) = 0
Assignment: f2 = f1: f2 = 5/3

Read Test
2 7
Read f1 = 2/7

Unary Ops Test: Using f1 = 2/5 for all
Negate: -f1 = -2/5
Preserve: +f1 = 2/5
Componendo: ++f1 = 7/5
Dividendo: --f1 = -3/5
Lazy Componendo: f1++ = 2/5 Lazy f1 = 7/5
Lazy Dividendo: f1-- = 2/5 Lazy f1 = -3/5

Binary Ops Test: Using f1 = 5/12. f2 = 7/18
Binary Plus: f1 + f2 = 29/36
Binary Minus: f1 - f2 = 1/36
```

All tests passed

```
Binary Multiply: f1 * f2 = 35/216
Binary Divide: f1 / f2 = 15/14
Binary Residue: f1 % f2 = 1/36

Binary Assignment Ops Test: Using f1 = 5/12. f2 = 7/18
Plus Assign: f1 += f2: f1 = 29/36
Minus Assign: f1 -= f2: f1 = 1/36
Multiply Assign: f1 *= f2: f1 = 35/216
Divide Assign: f1 /= f2: f1 = 15/14
Residue Assign: f1 %= f2: f1 = 1/36

Logical Ops Test: Using f1 = 5/12. f2 = 7/18 for all
Equal: false
Not Equal: true
Less: false
Less Equal: false
Greater: true
Greater Equal: true

Extended Ops Test: Using f1 = 5/12 for all
Invert: !f1 = 12/5
Double: (double)f1 = 0.416667

Static Constants Test
UNITY = 1
ZERO = 0
```

Testing Fraction: Version 2: Fail Test Application


```
int main() {
    try { cout << "Construct Fraction (1, 0): ";
        Fraction f1(1, 0); // Construct Fraction (1, 0): Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; } cout << endl;
    Fraction f1;
    try { cout << "Read f1 = "; // Read f1 = 1 0
        cin >> f1; cout << f1 << endl; // Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; } cout << endl;
    f1 = Fraction(5, 12); Fraction f2 = Fraction::ZERO, f3;
    try { cout << "Binary Divide: f3 = " << f1 << " / " << f2 << ": ";
        f3 = f1 / f2; cout << f3 << endl; // Binary Divide: f3 = 5/12 / 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Binary Residue: f3 = " << f1 << " % " << f2 << ": ";
        f3 = f1 % f2; cout << f3 << endl; // Binary Residue: f3 = 5/12 % 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Divide Assign: f1 = " << f1 << " /= " << f2 << ": ";
        f1 /= f2; cout << f1 << endl; // Divide Assign: f1 = 5/12 /= 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Residue Assign: f1 = " << f1 << " %= " << f2 << ": ";
        f1 %= f2; cout << f1 << endl; // Residue Assign: f1 = 5/12 %= 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Invert: f1 = " << " ! " << f2 << ": ";
        f1 = !f2; cout << f1 << endl; // Invert: f1 =  ! 0: Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; }
}
```

Cross Version Test

- To assess Verion 2 against Version 1, again consider the following fraction expression:

$$
\begin{aligned}
f1 &= \frac{2}{3} \\
f2 &= \frac{8}{1} \\
f3 &= \frac{5}{6} \\
f4 &= (f1 + f2)/(f1 - f2) + !f3 - f2 * f3 = -\frac{1097}{165}
\end{aligned}
$$

- *Using* ***Version 1****: Very easy to get confused in the chain of calls and parentheses*

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;

    f4 = f1.Add(f2).Divide(f1.Subtract(f2)).Add(f3.Invert()).Subtract(f2.Multiply(f3));
    Fraction::Write(f4); cout << endl;
}
```

- *Using* ***Version 2****: Just as we write the algebra*

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;

    f4 = (f1 + f2) / (f1 - f2) +!f3 - f2 * f3;
    cout << f4 << endl;
}
```

Tutorial Summary

- Analysed the difference between Built-in & UDT
- Discussed the meaning of Building a data type
- Understood the necessity of Building a data type
- Built a Fraction data type by iterative refinement