Programming in Modern C++

Tutorial T09: How to design a UDT like built-in types?: Part 3: Updates and Mixes of UDTs

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 Recap

- Presented the design, implementation and test for `Int<N>` and `Poly<T>` types
- Showed how `Poly<int>` as well as `Poly<Fraction>` works
- Outlined several practice UDTs for homework

Tutorial Objectives

- To update UDTs: `Fraction`, `Int<N>` and `Poly<T>`
- To test mix of UDTs

Tutorial Outline

Update UDTs

Update UDTs

- `Fraction`
  - Change binary arithmetic and comparison operators to `friend` functions from non-static member functions
  - Parameterize `Fraction` with type `T = int`
  - Provide mixed format support
- `Int<N>`
  - In the constructor of `Int<N>`, allow out-of-range values to wrap around instead of `assert`
  - Implement `operator*()`, `operator/()`, and `operator%()`
- Mixed UDT Apps
  - Test `Fraction<int>`
  - Test `Fraction<Int<4> >`
  - Test `Poly<Int<4> >`
  - Test `Poly<Fraction<Int<4> > >`

Fraction UDT: Update

Fraction UDT: Update

Fraction UDT: Update: Agenda

- We have the following update agenda for `Fraction`
  - Change binary arithmetic and comparison operators to `friend` functions from non-static member functions
  - Parameterize `Fraction` with type `T = int`
  - Provide mixed format support

Fraction: friend Operators

- To facilitate the power of `friend` operators, we make the constructor non-`explicit`

```
/* explicit */ Fraction(int n = 1, int d = 1): // Three overloads
    n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ cannot be -ve
{ *(*this); } // Reduces the fraction by operator*()
```

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

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

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

```
friend bool operator==(const Fraction& f1, const Fraction& f2); // Eq()
friend bool operator!=(const Fraction& f1, const Fraction& f2); // NotEq()
friend bool operator<(const Fraction& f1, const Fraction& f2);  // Less()
friend bool operator<=(const Fraction& f1, const Fraction& f2); // LessEq()
friend bool operator>(const Fraction& f1, const Fraction& f2);  // More()
friend bool operator>=(const Fraction& f1, const Fraction& f2); // MoreEq()
```

Fraction: Template

- To provide an underlying type for `Fraction`, we introduce type variable `T` with `int` as default
- `T` could be any integral type like `int`, `short`, `char`, `long`, or `Int<N>` etc.
- We also change the name of the type from `Fraction` to `Fraction_` not to clutter the user name space

```
template<typename T = int>
class Fraction_ { public:

// Change Fraction to Fraction_

// ...
private:
    T       n_;         // The Numerator. Earlier:int
    T       d_;         // The Denominator. Earlier: unsigned int
// ...
};
```

- In the application, add:

```
typedef Fraction_<int> Fraction; // Fraction is used in the application
```

Fraction: Template


```
template<typename T = int> class Fraction_ { public: // Parameterized Fraction_ with T = int
    Fraction_(T n = 1, T d = 1); Fraction_(const Fraction_& f);           // Ctor, C-Ctor, C=, Dtor
    ~Fraction_(); Fraction_& operator=(const Fraction_& f);
    friend ostream& operator<<(ostream& os, const Fraction_& f);         // Streaming ops
    friend istream& operator>>(istream& is, Fraction_& f);
    Fraction_ operator-() const; Fraction_ operator+() const;            // Unary arithmetic ops
    Fraction_& operator++(); Fraction_ operator++(int);
    Fraction_& operator--(); Fraction_ operator--(int);
    friend Fraction_ operator+(const Fraction_& f1, const Fraction_& f2); // Binary arithmetic ops
    friend Fraction_ operator-(const Fraction_& f1, const Fraction_& f2);
    friend Fraction_ operator*(const Fraction_& f1, const Fraction_& f2);
    friend Fraction_ operator/(const Fraction_& f1, const Fraction_& f2);
    friend Fraction_ operator%(const Fraction_& f1, const Fraction_& f2);
    friend bool operator==(const Fraction_& f1, const Fraction_& f2);    // Comparison ops
    friend bool operator!=(const Fraction_& f1, const Fraction_& f2);
    friend bool operator<(const Fraction_& f1, const Fraction_& f2);
    friend bool operator<=(const Fraction_& f1, const Fraction_& f2);
    friend bool operator>(const Fraction_& f1, const Fraction_& f2);
    friend bool operator>=(const Fraction_& f1, const Fraction_& f2);
    Fraction_& operator+=(const Fraction_& f);                           // Advanced assignment ops
    Fraction_& operator-=(const Fraction_& f); Fraction_& operator*=(const Fraction_& f);
    Fraction_& operator/=(const Fraction_& f); Fraction_& operator%=(const Fraction_& f);
    Fraction_ operator!() const; operator double() const;                // Special ops
private: static T gcd(T a, T b); static T lcm(T a, T b); Fraction_& operator*(); // Support functions
};
```

Fraction: Mixed Format Support

- Irrespective of whether a fraction is in *simple format* like $\frac{n}{d}$ ($\frac{2}{3}$ or $\frac{17}{5}$) or in *mixed format* like $w\frac{n}{d}$ ($\frac{2}{3}$ or $3\frac{2}{5}$), its *internal representation is always simple* $w\frac{n}{d} \equiv \frac{w*d+n}{d}$
- Hence, mixed format support is limited to:
  - Fraction construction

```
explicit Fraction_(T w, T n, T d) : // Mixed format fraction constructor
    n_(d < 0 ? w * -d - n : w * d + n), d_(d < 0 ? -d : d) // d must by non-negative
{ *(*this); } // Reduces the fraction
```

  - Fraction output operator

```
friend ostream& operator<<(ostream& os, const Fraction_& f);
```

  - Fraction input operator

```
friend istream& operator>>(istream& is, Fraction_& f);
```

- While the constructor can be distinguished by the distinct signature, the streaming operators have the same signature for *simple* as well as *mixed format*. Hence, we need a way to tell these operators about the format

Fraction: Mixed Format Support in Streaming Operators

- To design for the mixed format i/o, we recall the support for writing integers in multiple bases using <iomanip> component in standard library

```
#include <iostream>
#include <iomanip>
int main() { int i = 76;
    std::cout << std::oct << i << std::endl; // Set octal format. Prints 114
    std::cout << std::hex << i << std::endl; // Set hexadecimal format. Prints 4c
    std::cout << std::dec << i << std::endl; // Set decimal format. Prints 76
}
```

- Using <iomanip>, the format flag is set in ostream (cout). We cannot do that as ostream (or istream) cannot be changed. So, we need to keep the format option in the Fraction_ class
- We add a static bool bMixedFormat_ (true for mixed format, false for simple format)
- In the streaming operators, we can check this flag and adopt the appropriate formatting
- But how do we set / reset this flag? Using SetFormat(bool) spoils the built-in type-like syntax

Easy

```
Fraction f(17,5);
Fraction::SetFormat(false);
cout << f; // 17/5
Fraction::SetFormat(true);
cout << f; // 3+2/5
```

Desired

```
Fraction f(17,5);
cout << Fraction::simple;
cout << f; // 17/5
cout << Fraction::mixed;
cout << f; // 3+2/5
```

Fraction: Mixed Format Support in Streaming Operators

- For cout << Fraction::simple (cout << Fraction::mixed), we observe the following:
  - Fraction::simple (Fraction::mixed) needs to have an appropriate type, say Format, different from Fraction_, yet encapsulated by Fraction_. So we choose nested class Format in Fraction_
    ```
    class Format { bool bFormat_; public: Format(bool b): bFormat_(b) { } /* ... */ }; // Wraps bool
    ```
  - Fraction::simple and Fraction::mixed must be constants in Fraction_
    ```
    static const Format mixed;  // bMixedFormat_ = true
    static const Format simple; // bMixedFormat_ = false
    ```
  - Output streaming Fraction::simple (Fraction::mixed) in Fraction::Format should print nothing and set Fraction_::bMixedFormat_ appropriately
    ```
    friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
        // sets / resets mixed format flag
        bMixedFormat_ = m.bFormat_; // error: operator<< is friend of Format, not of Fraction_
        return os;
    }
    ```
  - So we use a wrapper in Format
    ```
    class Format { // ...
        void SetMixedFormat(bool b) const { bMixedFormat_ = b; } // access private member of Fraction_
        friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
            m.SetMixedFormat(m.bFormat_); // sets / resets mixed format flag
            return os;
        }
    };
    ```

Fraction: Mixed Format Support in Streaming Operators

- We use Fraction_::bMixedFormat_ to decide the format in the streaming operators:

```
friend ostream& operator<<(ostream& os, const Fraction_& f) {
    T w = 0, n = f.n_, d = f.d_;
    if (f.bMixedFormat_) { // Mixed format support
        w = n / d;         // Whole part = 3 in 17/5
        n %= d;            // Fraction part = 2/5 in 17/5
        if (n < 0) { --w; n += d; } // Negative: -17/5 = -4+3/5 = (-17/5 -1)+(-17%5+5)/5
        if (w) os << w << "+";      // w+ to be suppressed if 0
    }
    os << n; if ((n != 0) && (d != 1)) os << "/" << d; // To print the fraction part in both formats
    return os;
}

friend istream& operator>>(istream& is, Fraction_& f) {
    if (f.bMixedFormat_) { // Mixed format support - reads 3 numbers: w, n, d
        cout << "Input fraction in mixed Format" << endl;
        T w, n;
        is >> w >> n >> f.d_;
        f.n_ = w * f.d_ + n;
    }
    else // Simple format support - reads 2 numbers: n, d
        is >> f.n_ >> f.d_;
    *f; // Reduces the fraction
    return is;
}
```

Fraction: Mixed Format Support in Streaming Operators

- Finally, we put together Fraction_::Format class:

```
template<typename T = int> class Fraction_ { public: // ...
private: /* ... */ // Support for Mixed Format
    class Format { private: // Wraps bool so that special IO operators can be defined
        bool bFormat_; // Truthvalue for  Format object
        void SetMixedFormat(bool b) const; // Sets Fraction_::bMixedFormat_
        Format(bool b): bFormat_(b) { }    // Ctor is private - used only by friend class Fraction_

        friend ostream& operator<<(ostream& os, const Format& m); // Called to set bMixedFormat_
        friend istream& operator>>(istream& is, const Format& m); // Called to set bMixedFormat_

        friend class Fraction_; // Since ctor of Format is private, Fraction_ must be a friend
    };
public:
    // Format markers
    static const Format mixed;  // Denotes bMixedFormat_ = true
    static const Format simple; // Denotes bMixedFormat_ = false
// ...
};
```

- Instantiations of Format markers are:

```
const Fraction::Format Fraction::mixed(true);   // Denotes bMixedFormat_ = true
const Fraction::Format Fraction::simple(false); // Denotes bMixedFormat_ = false
```

Int<N> UDT: Update

Int<N> **UDT**: Update

Int<N> UDT: Update: Agenda

- We have the following update agenda for `Int<N>`
  - In the constructor of `Int<N>`, allow out-of-range values to wrap around instead of `assert`
  - Implement `operator*()`, `operator/()`, and `operator%()`

Int<N>: Constructor with Wraparound

- The constructor of Int<N> is:

```
template<typename T = int, unsigned int N = 4>
class Int_ { public: // ...
    explicit Int_<T, N>(int v = 1): v_(v) {     // Two overloads of Constructor
        assert(v_ <= static_cast<int>(MaxInt)); // assert will fire if the value
        assert(v_ >= static_cast<int>(MinInt)); // is out of limits
    } // ...
};
```

- For wraparound, we remove asserts and overload operator*()

```
template<typename T = int, unsigned int N = 4>
class Int_ { public: // ...
    explicit Int_<T, N>(int v = 1): v_(v) // Two overloads of Constructor
    { *(*this); }
    Int_<T, N> operator*() { // Wraparound operator
        v_ = v_ % TwoPowerN_T;
        if (v_ > MaxInt_T) v_ -= TwoPowerN_T;
        else if (v_ < MinInt_T) v_ += TwoPowerN_T;
        return *this;
    } // ...
};
```

- With this we get the following wraparound:

```
cout << Int_<>(5) << ' ' << Int_<>(77) << ' ' << Int_<>(-43) << endl; // 5 -3 5
```

Int<N>: Binary Operators

- With the wraparound, it becomes straightforward to implement binary operators with overflow

```
template<typename T = int, unsigned int N = 4>class Int_ { public: // ...
    friend Int_<T, N> operator+(const Int_<T, N>& i1, const Int_<T, N>& i2) {
        return Int_<T, N>(i1.v_ + i2.v_); }
    friend Int_<T, N> operator-(const Int_<T, N>& i1, const Int_<T, N>& i2) {
        return i1 + (-i2); } // return Int_<T, N>(i1.v_ - i2.v_); is also okay
    friend Int_<T, N> operator*(const Int_<T, N>& i1, const Int_<T, N>& i2) {
        return Int_<T, N>(i1.v_ * i2.v_); }
    friend Int_<T, N> operator/(const Int_<T, N>& i1, const Int_<T, N>& i2) {
        return Int_<T, N>(i1.v_ / i2.v_); }
    friend Int_<T, N> operator%(const Int_<T, N>& i1, const Int_<T, N>& i2) {
        return Int_<T, N>(i1.v_ % i2.v_); } // ...
};
```

- With this we get the following:

```
cout << "Binary Plus: Int(2) + Int(3) = " << (Int(2) + Int(3)) << endl;          // 5
cout << "Binary Plus: Int(-6) + Int(-7) = " << (Int(-6) + Int(-7)) << endl;      // 3
cout << "Binary Minus: Int(2) - Int(3) = " << (Int(2) - Int(3)) << endl;         // -1
cout << "Binary Minus: Int(-6) - Int(-7) = " << (Int(-6) - Int(-7)) << endl;     // 1
cout << "Binary Multiply: Int(3) * Int(2) = " << (Int(3) * Int(2)) << endl;      // 6
cout << "Binary Multiply: Int(7) * Int(5) = " << (Int(7) * Int(5)) << endl;      // 3
cout << "Binary Multiply: Int(-8) * Int(-8) = " << (Int(-8) * Int(-8)) << endl; // 0
cout << "Binary Divide: Int(3) / Int(2) = " << (Int(3) / Int(2)) << endl;        // 1
cout << "Binary Divide: Int(7) / Int(-5) = " << (Int(7) / Int(-5)) << endl;      // -1
cout << "Binary Residue: Int(3) % Int(2) = " << (Int(3) % Int(2)) << endl;       // 1
cout << "Binary Residue: Int(-6) % Int(2) = " << (Int(-6) % Int(2)) << endl;     // 0
```

Int<N>: Binary Operators: Properties

- Try to prove the usual arithmetic properties of the `Int<N>` binary operators for addition, subtraction, multiplication, and division under wraparound:
  - Are all operators *Associative*? For example,
    - `a + b + c = (a + b) + c = a + (b + c)`
  - Are addition and multiplication *Commutative*? For example,
    - `a + b = b + a`
  - Do multiplication and division *Distribute* over addition and subtraction?For example,
    - `a * (b + c) = a * b + a * c`
- Especially, check for the boundary conditions under wraparound:
  - `MaxInt + 1` = `MinInt`
  - `MinInt - 1` = `MaxInt`
  - `- MinInt` = `MinInt`
- Consider exception and / or assert support in the constructors and / or operators if some specific values can break the properties

Mixed UDT Apps

Mixed UDT Apps: Agenda

- We have the following agenda for Mixed UDT Apps
  - Test `Fraction<int>`
  - Test `Fraction<Int<4> >`
  - Test `Poly<int>`: Done in **Tutorial 08**
  - Test `Poly<Fraction<int > >`: Done in **Tutorial 08** - actually using the non-template version of `Fraction`
  - Test `Poly<Int<4> >`
  - Test `Poly<Fraction<Int<4> > >`

Fraction<int>: Application

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

typedef Fraction_<int> Fraction;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
bool  Fraction::bMixedFormat_ = false;
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);

int main() {
    Fraction fa(5, 3);
    cout << "Fraction fa(5, 3) = " << Fraction::mixed << fa << " = " << Fraction::simple << fa;
    Fraction fb(7, 9);
    cout << "Fraction fb(7, 9) = " << Fraction::mixed << fb << " = " << Fraction::simple << fb;
    cout << "fa + fb = " << Fraction::mixed << (fa + fb) << " = " << Fraction::simple << (fa + fb);
    cout << "fa - fb = " << Fraction::mixed << (fa - fb) << " = " << Fraction::simple << (fa - fb);
    cout << "fa * fb = " << Fraction::mixed << (fa * fb) << " = " << Fraction::simple << (fa * fb);
    cout << "fa / fb = " << Fraction::mixed << (fa / fb) << " = " << Fraction::simple << (fa / fb);
}
```

```
Fraction fa(5, 3) = 1+2/3 = 5/3
Fraction fb(7, 9) = 7/9 = 7/9
fa + fb = 2+4/9 = 22/9
fa - fb = 8/9 = 8/9
fa * fb = 1+8/27 = 35/27
fa / fb = 2+1/7 = 15/7
```

Fraction<Int<4> >: Application

```
#include <iostream>
using namespace std;
#include "Frac.h"
#include "../Int/Int.h"
typedef Int_<int, 4> Int; typedef Fraction_<Int> Fraction;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
bool  Fraction::bMixedFormat_ = false;
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);

int main() {
    Fraction fa(5, 3);
    cout << "Fraction fa(5, 3) = " << Fraction::mixed << fa << " = " << Fraction::simple << fa;
    Fraction fb(7, 10);
    cout << "Fraction fb(7, 10) = " << Fraction::mixed << fb << " = " << Fraction::simple << fb;
    cout << "fa + fb = " << Fraction::mixed << (fa + fb) << " = " << Fraction::simple << (fa + fb);
    cout << "fa - fb = " << Fraction::mixed << (fa - fb) << " = " << Fraction::simple << (fa - fb);
    cout << "fa * fb = " << Fraction::mixed << (fa * fb) << " = " << Fraction::simple << (fa * fb);
    cout << "fa / fb = " << Fraction::mixed << (fa / fb) << " = " << Fraction::simple << (fa / fb);
}
```

```
Fraction fa(5, 3) = 1+2/3 = 5/3
Fraction fb(7, 10) = -2+5/6 = -7/6
fa + fb = 1/2 = 1/2
fa - fb = 1/6 = 1/6
fa * fb = -2+1/2 = -3/2
fa / fb = 2/5 = 2/5
```

Poly<Int<4> >: Application


```
#include <iostream>
using namespace std;
#include "../Int/Int.h"
#include "Poly.h"
typedef Int_<int, 4> Int;
const int Int::TwoPowerN_T = 1 << N;          // 2^N
const int Int::MaxInt_T = (1 << (N-1))-1; /* 2^(N-1)-1 */ Int::MinInt_T = -(1 << (N-1)); // -2^(N-1)
const Int Int::MaxInt = Int(Int::pow() - 1), Int::MinInt = Int(-Int::pow());

void main() { vector<Int> vf = { 2, 15, 7 };
    Poly<Int> pf1(vf); cout << "pf1(x): " << pf1 << " pf1(2) = " << pf1(2) << endl;
    Poly<Int> pf2; cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
    cin >> pf2; /* 3 9 7 2 -11 */ cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
    Poly<Int> pf3 = pf1 + pf2; cout << "pf3(x): " << pf3 << " pf3(2) = " << pf3(2) << endl;
    Poly<Int> pf4 = pf1 - pf2; cout << "pf4(x): " << pf4 << " pf4(2) = " << pf4(2) << endl << endl;
}
```

```
pf1(x): 7x^2 + -1x^1 + 2. pf1(2) = -4 // 2 = 2, 15 = -1, 7 = 7. pf1(2)= 7*4 + -1*2 + 2 = 28-2+2 = 28 = -4
pf2(x): 1. pf2(2) = 1
Enter degree of the polynomial 3
Enter all the coefficients like a0+a1*x+a2*x^2+....an*x^n
9 7 2 -11 // -7 7 2 5
pf2(x): 5x^3 + 2x^2 + 7x^1 + -7. pf2(2) = 7 // pf2(2)= 5*8 + 2*4 + 7*2 + -7 = 56+8+14-7 = 71 = 71-64 = 7
pf3(x): 5x^3 + -7x^2 + 6x^1 + -5. pf3(2) = 3
pf4(x): -5x^3 + 5x^2 + -8x^1 + -7. pf4(2) = 5
```

Poly<Fraction<Int<4> > >: Application


```
#include <iostream>
using namespace std;
#include "../Fraction/Frac.h"
#include "../Int/Int.h"
#include "Poly.h"
const int N = 4; typedef Int_<int, 4> Int; const int Int::TwoPowerN_T = 1 << N; // 2^N
const int Int::MaxInt_T = (1 << (N-1))-1; /* 2^(N-1)-1 */ Int::MinInt_T = -(1 << (N-1)); // -2^(N-1)
const Int Int::MaxInt = Int(Int::pow() - 1), Int::MinInt = Int(-Int::pow());
typedef Fraction_<Int> Fraction; bool Fraction::bMixedFormat_ = false;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);
void main() {
    vector<Fraction> vf1 = { Fraction(1, 2), Fraction(-3, 5), Fraction(2, 4) };
    Poly<Fraction> pf1(vf1); cout << "pf1(x): "<< pf1 << " pf1(Fraction(2)) = "<< pf1(Fraction(2))<< endl;
    vector<Fraction> vf2 = { Fraction(1, 2), Fraction(2, 3) };
    Poly<Fraction> pf2(vf2); cout << "pf2(x): "<< pf2 << " pf2(Fraction(2)) = "<< pf2(Fraction(2))<< endl;
    Poly<Fraction> pf3 = pf1 + pf2;
    cout << "pf3(x): " << pf3 << " pf3(Fraction(2)) = " << pf3(Fraction(2)) << endl;
    Poly<Fraction> pf4 = pf1 - pf2;
    cout << "pf4(x): " << pf4 << " pf4(Fraction(2)) = " << pf4(Fraction(2)) << endl << endl;
}
pf1(x): 1/2x^2 + -3/5x^1 + 1/2. pf1(Fraction(2)) = 7/6 // pf1(2/1)= 1/2*4 -3/5*2 + 1/2 = 7/6
pf2(x): 2/3x^1 + 1/2. pf2(Fraction(2)) = -5/6 // pf2(2/1) = 2/3*2/1 + 1/2 = 4/3 + 1/2 = 11/6 = -5/6
pf3(x): 1/2x^2 + 1. pf3(Fraction(2)) = 3 // pf4(2/1) = 1/2*4/1 + 1 = 3
pf4(x): 1/2x^2. pf4(Fraction(2)) = 2 // pf4(2/1) = 1/2*4/1 = 2
```

Caveat: Mixes may fail

Caveat: Mixes may fail

Caveat in mixing UDTs

- While `Fraction<int>`, `Poly<int>`, `Int<N>`, or `Poly<Fraction<int> >` work perfectly fine, `Fraction<Int<N> >` or `Poly<Fraction<Int<N> > >` may have some surprise
- This is due to the `T gcd(T, T)` algorithm in the context of `Int<N>`. Normally, we invoke `gcd()` for positive numbers only (that's how the Euler's Algorithm is designed to work)
- However, for `MinInt` in `Int<N>`, we have `-MinInt = MinInt`. Hence, if one of the `gcd()` parameters is `MinInt` we are perpetually in the realm of negative numbers. This leads to an *infinite loop* in the code below:

```
static T gcd(T a, T b) { // Finds the gcd for two +ve integers
    while (a != b) if (a > b) a = a - b; else b = b - a; // N = 4. (-8,3) => (-8,-5) => (-8,3) => ...
    return a;
}
```

- So we choose to `throw` (and eventually `assert` in the constructor) when one of the `gcd()` arguments is negative (eventually `MinInt`)

```
static T gcd(T a, T b) { // Finds the gcd for two +ve integers
    if (a < 0) throw "Negative first arg in gcd";
    if (b < 0) throw "Negative second arg in gcd";
    while (a != b) if (a > b) a = a - b; else b = b - a; // For N = 4, a = -8 is an infinite loop
    return a;
}
```

- How to fix?

Tutorial Summary

- UDTs Fraction, Int and Poly have been updated with various features
- Mixed applications involving multiple UDTs have been checked
- Caveat or loophole has been identified