Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Programming in Modern C++

Tutorial T08: How to design a UDT like built-in types?: Part 2: `Int` and Poly 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 T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Tutorial Recap

- 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

Tutorial Objectives

- To build more UDTs: `Int<N>` and `Poly<T>`
- To test mix of UDTs

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Tutorial Outline

1. Tutorial Recap
2. `Int<N>` UDT
    - Design
        - Operations
        - Class Design
    - Implementation
    - Test
3. Polynomial UDT
    - Design
    - Implementation
    - Test
4. Practice UDTs
5. Tutorial Summary

Tutorial T08

Partha Pratim Das

Tutorial Recap
Objective & Outline
Int<N> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary

Int<N> UDT

Understanding int

- The datatype we are most familiar with is `int` which is a signed integer
  - Represented in a given number of bits, typically, 8, 16, 32, 64, or 128
  - Hence, `int` can represent numbers from `INT_MAX` to `INT_MIN`
  - For example, for 32 bits, `INT_MAX` $= 2^{31} - 1 = 2147483647$ and `INT_MIN` $= -2^{31} = -2147483648$
  - Beyond the `INT_MAX` .. `INT_MIN` range we get integer overflow and numbers wraparound

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << INT_MAX   << endl; // 2147483647
    cout << INT_MIN   << endl; // -2147483648
    cout << INT_MAX+1 << endl; // -2147483648 integer overflow in expression of type 'int'
    cout << INT_MIN-1 << endl; // 2147483647  integer overflow in expression of type 'int'
    cout << -INT_MIN  << endl; // -2147483648 integer overflow in expression of type 'int'
}
```

Design of Int UDT

- To understand `int` better, we intend to design a UDT `Int<N>` which can behave like `int` albeit for a of size `N > 0` bits that can be specified
- The range of values will be:
  - $\texttt{MinInt} = -2^{N-1} \ldots \texttt{MaxInt} = 2^{N-1} - 1$
- The broad tasks involved include:
  - Make a clear statement of the concept of `Int`
  - Identify a representation for a `Int` object
  - Identify the properties and assertions applicable to all objects
  - Identify the operations for `Int` objects
    - Choose appropriate operators to overload for the operations
    - For example `operator+` to add two `Int` objects, or `operator<<` to stream a `Int` to `cout`
    - *Do not break the natural semantics for the operators*

Notion of Int

- Intuitively `Int<N>` is a notation for whole numbers of the form of `N` bits signed integers
- `MinInt` $= -2^{N-1} \ldots$ `MaxInt` $= 2^{N-1} - 1$
- Numbers in `Int<N>` bits within a range of values `MinInt` .. `MaxInt`. Beyond this range the numbers wrap around:
  - `MaxInt` $+ 1 =$ `MinInt`
  - `MinInt` - 1 $=$ `MaxInt`
- For example:
  - `N` $= 4 \Rightarrow$ Range: -8 .. 7
  - `MinInt` = -$2^3 = -8$ and `MaxInt` $= 2^3 - 1 = 7$
  - Except for *overflow*[1] as follows, all operations of `Int<N>` is same as `int`
    - ▷ $7 + 1 = -8$
    - ▷ $-8 - 1 = 7$
    - ▷ $-8 = -8$

[1] Some authors distinguish between overflow (being more than `MaxInt`) and underflow (being less than `MinInt`). However, we prefer to use the term overflow in both cases because actually representation *overflows* the bits in both cases

Operations of Int<N>

Definition

Limits:

$$\begin{aligned} \texttt{MaxInt} &= 2^{N-1} - 1 \\ \texttt{MinInt} &= -2^{N-1} \end{aligned}$$

Negation:

$$\begin{aligned} -a &= a, \textit{if } a == \texttt{MinInt} \\ &= -a, \textit{otherwise} \end{aligned}$$

Addition:

$$\begin{aligned} a + b &= a + b - 2^N, \textit{if } a + b > \texttt{MaxInt} \\ &= a + b + 2^N, \textit{if } a + b < \texttt{MinInt} \\ &= a + b, \textit{otherwise} \end{aligned}$$

Subtraction:

$$a - b = a + (-b)$$

Let $\texttt{N} = 4$

Example 1: $2 + 3 = 5$. $4 + 7 = -5$. $(-5) + 6 = 1$. $(-3) + (-2) = -5$. $(-6) + -7 = 3$

Example 2: $2 - 3 = -1$. $4 - 7 = -3$. $(-5) - 6 = 5$. $(-3) - (-2) = -1$. $(-6) - (-7) = 1$

Multiplication, Division, and Modulus: Left as exercise

Design of Int<N> Class

- Clearly, the representation of a Int<N> needs to be a template with an unsigned int param N
- For the implementation, the Int<N> needs to use an underlying type T where basic arithmetic operations are available. So T is a type parameter for Int<N>. By default this can be int
- It is important to note that N <= sizeof(T) * 8. Otherwise, our basic operations may overflow
- Hence, the Int<N> class would look like:

```
template<typename T = int, unsigned int N = 4>
class Int_ { // an N-bits integer class with underlying type T
    T v_;    // actual value in underlying type T
    // ... Rest of the class
}
```

- Note that we name the type as Int_ so that we can conveniently alias it in the user program as:

```
template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // T = int and N = 4
// Use as Int
```

- Int<N> should support the operation of int:
- Int<N> should also support conversion the underlying type T
- Int<N> may support the following constants for convenience of implementation:

```
static const Int_<T, N> MaxInt; // 2^(N-1)-1
static const Int_<T, N> MinInt; // -2^(N-1)
```

Design of Int<N> Interface and Implementation

```
template<typename T = int, unsigned int N = 4>
class Int_ { public:
    static const Int_<T, N> MaxInt; // 2^(N-1)-1
    static const Int_<T, N> MinInt; // -2^(N-1)

    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 is out of limits
        assert(v_ >= static_cast<int>(MinInt));
    }
    Int_<T, N>(const Int_<T, N>& i) : v_(i.v_) { } // Copy Constructor
    ~Int_<T, N>() { } // No virtual destructor needed
    Int_<T, N>& operator=(const Int_<T, N>& i) { v_ = i.v_; return *this; } // Assignment
    // Streaming operators for IO
    friend ostream& operator<<(ostream& os, const Int_<T, N>& i) { os << i.v_; return os; }
    friend istream& operator>>(istream& is, Int_<T, N>& i) {
        T v; is >> v; i = Int_<T, N>(v); // We deliberately construct to test that v is within limits
        return is;
    }
    // Unary arithmetic operators
    Int_<T, N> operator-() const { return Int_<T, N>(v_ == MinInt_T? v_: -v_); }
    Int_<T, N> operator+() const { return *this; }
    Int_<T, N>& operator++()     { *this = *this + Int_<T, N>(1); return *this; }
    Int_<T, N>& operator--()     { *this = *this - Int_<T, N>(1); return *this; }
    Int_<T, N> operator++(int)   { Int_<T, N> i = *this; ++*this; return i; }
    Int_<T, N> operator--(int)   { Int_<T, N> i = *this; --*this; return i; }
```


Tutorial T08
Partha Pratim Das
Tutorial Recap
Objective & Outline
Int<N> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary

Design of Int<N> Interface and Implementation


```
// Binary arithmetic operators
friend Int_<T, N> operator+(const Int_<T, N>& i1, const Int_<T, N>& i2) {
    T v = i1.v_ + i2.v_; // add values in underlying type T
    if (v > MaxInt_T) // MaxInt_T is MaxInt in type T
        return Int_<T, N>(v - TwoPowerN_T); // wrap around if the value is more than MaxInt
    else if (v < MinInt_T) // MinInt_T is MinInt in type T
        return Int_<T, N>(v + TwoPowerN_T); // wrap around if the value is less than MinInt
        else
            return Int_<T, N>(v); // value within limits - no action
}
friend Int_<T, N> operator-(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1 + (-i2); }

// NOT IMPLEMENTED - left as exercises
friend Int_<T, N> operator*(const Int_<T, N>& i1, const Int_<T, N>& i2);
friend Int_<T, N> operator/(const Int_<T, N>& i1, const Int_<T, N>& i2);
friend Int_<T, N> operator%(const Int_<T, N>& i1, const Int_<T, N>& i2);

// Logical comparison operators
friend bool operator==(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ == i2.v_; }
friend bool operator!=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ != i2.v_; }
friend bool operator<(const Int_<T, N>& i1, const Int_<T, N>& i2)  { return i1.v_ < i2.v_; }
friend bool operator<=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ <= i2.v_; }
friend bool operator>(const Int_<T, N>& i1, const Int_<T, N>& i2)  { return i1.v_ > i2.v_; }
friend bool operator>=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ >= i2.v_; }
```

Design of Int<N> Interface and Implementation


```
    // Advanced assignment operators: NOT IMPLEMENTED - left as exercises
    Int_<T, N>& operator+=(const Int_<T, N>& i);
    Int_<T, N>& operator-=(const Int_<T, N>& i);
    Int_<T, N>& operator*=(const Int_<T, N>& i);
    Int_<T, N>& operator/=(const Int_<T, N>& i);
    Int_<T, N>& operator%=(const Int_<T, N>& i);

    operator T() const { return v_; } // conversion to underlying type T

private: // data members
    T v_;                         // Value in underlying type T
    static const T MaxInt_T;      // MaxInt = 2^(N-1)-1 in underlying type T
    static const T MinInt_T;      // MinInt = -2^(N-1) in underlying type T
    static const T TwoPowerN_T; // 2^N in underlying type T

public: static int Int_<T, N>::pow() { return Int_<T, N - 1>::pow() * 2; }
};
template<typename T> class Int_<T, 1> { public: static int Int_<T, 1>::pow() { return 1; } };

// Instantiations of static const members
const Int Int::MaxInt = Int(Int::pow() - 1);
const Int Int::MinInt = Int(-Int::pow());
const int Int::MaxInt_T = static_cast<int>(Int::MaxInt); // 2^(N-1)-1
const int Int::MinInt_T = static_cast<int>(Int::MinInt); // -2^(N-1)
const int Int::TwoPowerN_T = (Int::MaxInt_T+1) << 1;      // 2^N
```

Testing Int<N>

```
#include <iostream>
using namespace std;

template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // N = 4

#include "Int.h"

void main() {
    cout << "Int::MaxInt = " << Int::MaxInt << endl; // Int::MaxInt = 7
    cout << "Int::MinInt = " << Int::MinInt << endl; // Int::MinInt = -8

    // Constructor, Copy Operations and Write Test
    Int f1(5);     cout << "Int f1(5) = " << f1 << endl;         // Int f1(5) = 5
    Int f2(0);     cout << "Int f2(0) = " << f2 << endl;         // Int f2(0) = 0
    Int f3;        cout << "Int f3 = " << f3 << endl;            // Int f3 = 1
    Int f4(f1);    cout << "Int f4(f1) = " << f4 << endl;        // Int f4(f1) = 5
    cout << "Assignment: f2 = f1: f2 = " << (f2 = f1) << endl; // Assignment: f2 = f1: f2 = 5

    // Read Test
    cin >> f1;                               // 3
    cout << "Read f1 = " << f1 << endl; // Read f1 = 3
```

Testing Int<N>

```
// Unary Operations Test
cout << "-Int(2) = " << -Int(2) << endl;       // -Int(2) = -2
cout << "-Int(-2) = " << -Int(-2) << endl;     // -Int(-2) = 2
cout << "-Int(-8) = " << -Int(-8) << endl;     // -Int(-8) = -8
cout << "-Int(7) = " << -Int(7) << endl;       // -Int(7) = -7
cout << "++Int(2) = " << ++Int(2) << endl;     // ++Int(2) = 3
cout << "++Int(7) = " << ++Int(7) << endl;     // ++Int(7) = -8
cout << "++Int(-8) = " << ++Int(-8) << endl; // ++Int(-8) = -7
cout << "--Int(0) = " << --Int(0) << endl;     // --Int(0) = -1
cout << "--Int(-7) = " << --Int(-7) << endl; // --Int(-7) = -8
cout << "--Int(-8) = " << --Int(-8) << endl; // --Int(-8) = 7

// Binary Operations Test
cout << "Int(2) + Int(3) = " << (Int(2) + Int(3)) << endl;       // Int(2) + Int(3) = 5
cout << "Int(4) + Int(7) = " << (Int(4) + Int(7)) << endl;       // Int(4) + Int(7) = -5
cout << "Int(-5) + Int(6) = " << (Int(-5) + Int(6)) << endl;     // Int(-5) + Int(6) = 1
cout << "Int(-3) + Int(-2) = " << (Int(-3) + Int(-2)) << endl; // Int(-3) + Int(-2) = -5
cout << "Int(-6) + Int(-7) = " << (Int(-6) + Int(-7)) << endl; // Int(-6) + Int(-7) = 3
```

Testing Int<N>


```
    // Binary Operations Test
    cout << "Int(2) - Int(3) = " << (Int(2) - Int(3)) << endl;     // Int(2) - Int(3) = -1
    cout << "Int(4) - Int(7) = " << (Int(4) - Int(7)) << endl;     // Int(4) - Int(7) = -3
    cout << "Int(-5) - Int(6) = " << (Int(-5) - Int(6)) << endl;   // Int(-5) - Int(6) = 5
    cout << "Int(-3) - Int(-2) = " << (Int(-3) - Int(-2)) << endl; // Int(-3) - Int(-2) = -1
    cout << "Int(-6) - Int(-7) = " << (Int(-6) - Int(-7)) << endl; // Int(-6) - Int(-7) = 1

    // Logical Operations Test
    cout << "Int(-6) == Int(-6): " << ((Int(-6) == Int(-6)) ? "T" : "F") << endl; // Int(-6) == Int(-6): T
    cout << "Int(4) != Int(3): " << ((Int(4) != Int(3)) ? "T" : "F") << endl;     // Int(4) != Int(3): T
    cout << "Int(-7) < Int(2): " << ((Int(-7) < Int(2)) ? "T" : "F") << endl;     // Int(-7) < Int(2): T
    cout << "Int(-7) <= Int(2): " << ((Int(-7) <= Int(2)) ? "T" : "F") << endl;   // Int(-7) <= Int(2): T
    cout << "Int(-5) > Int(-6): " << ((Int(-5) > Int(-6)) ? "T" : "F") << endl;   // Int(-5) > Int(-6): T
    cout << "Int(4) >= Int(4): " << ((Int(4) >= Int(4)) ? "T" : "F") << endl;     // Int(4) >= Int(4) = T
}
```

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT
- Design
  - Operations
  - Class Design
- Implementation
- Test

Polynomial UDT
- Design
- Implementation
- Test

Practice UDTs

Tutorial Summary

Polynomial UDT

Design of Polynomial UDT

- Polynomials $A(x)$ of x having degree $degree(A) = n$ and $n+1$ coefficients a_0, a_1, a_2, $\cdots$, a_n:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i$$

- The representation of a polynomial UDT `Poly` would need
  - a vector to keep the coefficients, and
  - a simple member to the degree (for null polynomials without coefficients)
- The types of coefficient and variable should be appropriate so that they can be multiplied and added. For simplicity, let us assume that they have the same type:

```
template<typename T = int> // Type of Coefficients and value
class Poly {               // a polynomial of type T
        vector<T>  coeff_; // coefficients
        size_t     deg_;   // deg_ = coeff_.size()-1
    // ... Rest of the class
}
```

Design of Polynomial UDT

- A polynomial $A(x)$ of degrees n may be negated to generate polynomial $R(x)$ of degrees n by flipping the sign of every coefficient. That is:

$$R(x) = -A(x) = -\sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{n} (-a_i) x^i$$

Hence,

$$r_i = -a_i, \ 0 \le i \le n$$

- Two polynomials $A(x)$ and $B(x)$ of degrees n and m respectively may be added to generate polynomial $R(x)$ of degree $max(n, m)$ by pairwise adding the coefficients of the same power. That is, for $n \ge m$

$$R(x) = A(x) + B(x) = \sum_{i=0}^{n} a_i x^i + \sum_{i=0}^{m} b_i x^i = \sum_{i=0}^{m} (a_i + b_i) x^i + \sum_{i=m+1}^{n} a_i x^i$$

Hence,

$$\begin{aligned} r_i &= a_i + b_i, \ 0 \le i \le m \\ &= a_i, \ m+1 \le i \le n \end{aligned}$$

Note: $A(x) - B(x) = A(x) + (-B(x))$

Design of Poly<T> Interface and Implementation

Tutorial T08
Partha Pratim Das
Tutorial Recap
Objective & Outline
Int<N> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary


```
template<typename T = int>        // Type of Coefficients and Value
class Poly { public:
    Poly(vector<T>& c = vector<T>( 1 )) : coeff_(c), deg_(c.size() - 1) { }
    Poly(size_t n) : coeff_(vector<T>(n+1)), deg_(n) { } // null polynomial
    Poly(const Poly& p) : coeff_(p.coeff_), deg_(p.deg_) { }
    ~Poly() { }          // No virtual destructor needed
    Poly& operator=(const Poly& p) { deg_ = p.deg_; coeff_ = p.coeff_; return *this; }

    Poly operator+() const { return *this; }
    Poly operator-() const { Poly r(deg_);
        for (unsigned int i = 0; i <= deg_; i++) r.coeff_[i] = -coeff_[i];
        return r;
    }
    Poly operator+(const Poly& p) const { Poly r(max(p.deg_, deg_)); // result
        vector<T> v;
        if (deg_ > p.deg_) { v = p.coeff_; r.coeff_ = coeff_; } // copy the longer (shorter)  vector
        else { v = coeff_; r.coeff_ = p.coeff_; }               // of coefficients to r.coeff_ (v)
        for (unsigned int i = 0; i <= min(p.deg_, deg_); i++) { // add the common coefficients
            r.coeff_[i] = t.coeff_[i] + v[i];
        }
        return r;
    }
    Poly operator-(const Poly&p) const { return *this + (-p); }
```

Design of Poly<T> Interface and Implementation

Tutorial T08
Partha Pratim Das
Tutorial Recap
Objective & Outline
Int<N> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary

```
    Poly& operator+=(const Poly& p) { *this = *this + p; return *this; }
    Poly& operator-=(const Poly&p) { *this = *this - p; return *this; }

    friend ostream& operator<<(ostream& os, const Poly& p) { int j;
        for (j = p.deg_; j >= 0; --j)
            { if (static_cast<T>(0) != p.coeff_[j]) break; } // first non-zero coeff.
        if (0 > j)
            os << 0;
        else
            if (0 == j) os << p.coeff_[j];
            else os << p.coeff_[j] << "x^" << j;
        for (int i = j-1; i >= 0; --i) {
            if (static_cast<T>(0) != p.coeff_[i]) {
                if (0 != i)
                    if (static_cast<T>(1) != p.coeff_[i])
                        os << " + " << p.coeff_[i] << "x^" << i;
                    else
                        os << " + " << "x^" << i;
                else
                    os << " + " << p.coeff_[i];
            }
        }
        os << ".";
        return os;
    }
```

Design of Poly<T> Interface and Implementation


```
    friend istream& operator >>(istream& is, Poly& p) {
        cout << "Enter degree of the polynomial ";
        is >> p.deg_;
        p.coeff_.resize(p.deg_ + 1);
        cout << "Enter all the coefficients like a0+a1*x+a2*x^2+....an*x^n" << endl;
        for (unsigned int i = 0; i <= p.deg_; i++)
            is >> p.coeff_[i];
        return is;
    }

    // Evaluates the polynomial - use Horner's Rule
    T operator()(const T& x) {
        T val = 0;
        for (int i = deg_; i >= 0; i--)
            val = val * x + coeff_[i];
        return val;
    }

private:
    vector<T>   coeff_;
    size_t      deg_;
    template<typename T> inline static T max(T a, T b) { return a > b ? a : b; }
    template<typename T> inline static T min(T a, T b) { return a < b ? a : b; }
};
```

Testing Poly<T>


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

void main() { vector<int> v = { 1, 2, 1 };
    Poly<int> p(v); cout << "p(x): " << p << " p(2) = " << p(2); // p(x): 1x^2 + 2x^1 + 1. p(2) = 9
    Poly<int> q(p); cout << "q(p): " << q << " q(2) = " << q(2); // q(p): 1x^2 + 2x^1 + 1. q(2) = 9
    Poly<int> s(vector<int>({ 0, 0, 1, 2, 1, 0, 2, 7, 0 }));
    cout << "s(x): " << s << " s(1) = " << s(1); // s(x): 7x^7 + 2x^6 + x^4 + 2x^3 + x^2. s(1) = 13
    Poly<int> r; cout << "r: " << r << " r(2) = " << r(2); // r: 1. r(2) = 1

    cin >> r; // 2 1 2 1
    cout << "r: " << r << " r(2) = " << r(2); // r: 1x^2 + 2x^1 + 1. r(2) = 9

    Poly<int> t(2); cout << "t(x): " << t << " t(2) = " << t(2); // t(x): 0. t(2) = 0

    r = p; cout << "r = p: " << r << " r(2) = " << r(2); // r = p: 1x^2 + 2x^1 + 1. r(2) = 9
    r = -p; cout << "r = -p: " << r << " r(2) = " << r(2); // r = -p: -1x^2 + -2x^1 + -1. r(2) = -9

    p = vector<int>({ 1, 5, 6 });
    cout << "p(x): " << p << " p(2) = " << p(2); // p(x): 6x^2 + 5x^1 + 1. p(2) = 35

    q = vector<int>({ 1, -2, 1 });
    cout << "q(x): " << q << " q(2) = " << q(2); // q(x): 1x^2 + -2x^1 + 1. q(2) = 1
```

Testing Poly<T>


```
    r = p + q; cout << "r = p + q: " << r << " r(2) = " << r(2); // r = p + q: 7x^2 + 3x^1 + 2. r(2) = 36
    r = p - q; cout << "r = p - q: " << r << " r(2) = " << r(2); // r = p - q: 5x^2 + 7x^1. r(2) = 34
    r = p; p += q; cout << "p += : " << p << " p(2) = " << p(2); // p += : 7x^2 + 3x^1 + 2. p(2) = 36
    p = r; p -= q; cout << "p -= : " << p << " p(2) = " << p(2); // p -= : 5x^2 + 7x^1. p(2) = 34

    vector<Fraction> vf = { Fraction(1, 2), Fraction(-3, 5), Fraction(2, 3) };
    Poly<Fraction> pf1(vf); cout << "pf1(x): " << pf1 << " pf1(2) = " << pf1(2) << endl;
    // pf1(x): 2/3x^2 + -3/5x^1 + 1/2. pf1(2) = 59/30

    Poly<Fraction> pf2; cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
    // pf2(x): 1. pf2(2) = 1

    cin >> pf2; // 1 2 3 1 2
    cout << "pf2: " << pf2 << " pf2(2) = " << pf2(2) << endl;
    // pf2(x): 1/2x^1 + 2/3. pf2(2) = 5/3

    Poly<Fraction> pf3 = pf1 + pf2;
    cout << "pf3(x): " << pf3 << " pf3(2) = " << pf3(2) << endl;
    // pf3(x): 2/3x^2 + -1/10x^1 + 7/6. pf3(2) = 109/30

    Poly<Fraction> pf4 = pf1 - pf2;
    cout << "pf4(x): " << pf4 << " pf4(2) = " << pf4(2) << endl;
    // pf4(x): 2/3x^2 + -11/10x^1 + -1/6. pf4(2) = 3/10
}
```

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT
Design
Operations
Class Design
Implementation
Test

Polynomial UDT
Design
Implementation
Test

Practice UDTs

Tutorial Summary

Practice UDTs

Practice UDTs

Tutorial T08
Partha Pratim Das
Tutorial Recap
Objective & Outline
Int<N> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary

- `Fraction`
  - Change binary arithmetic and comparison operators to `friend` functions from methods
  - Parameterize `Fraction` with type `T = int`
  - Provide mixed mode support
- `Int<N>`
  - Implement `operator*()`
  - Implement `operator/()`
  - Implement `operator%()`
- `Poly<T>`
  - Test `Poly<double>`
  - Use member function template for `operator()()` with another type parameter `U` for `x`
  - Analyze the compatibility issues between types `T` and `U`
- Mixed UDTs
  - Test `Fraction<Int<N> >`
  - Test `Poly<Int<N> >`
  - Test `Poly<Fraction<Int<N> > >`

Tutorial Recap

Objective & Outline

Int<N> UDT
Design
Operations
Class Design
Implementation
Test

Polynomial UDT
Design
Implementation
Test

Practice UDTs

Tutorial Summary

Tutorial Summary

- 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