Assignment SE-A-03

Partha Pratim Das

Objective & Outline

Data types

Fraction
Definition
Operations
Rules
Specs & Tasks
Sample Test

Int<N>
Definition
Operations
Specs & Tasks

Polynomial
Definition
Operations
Specs & Tasks

Mixed
Tasks

Software Engineering

Assignment 03 SE-A-03: Design UDTs like built-in types

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

Assign Date: *09-Mar-2022* **Submission Deadline**: *23:55, 15-Mar-2022*

Assignment Objectives

- Understand Building data types:
  - Fraction
  - Limited Size Integer
  - Polynomial

Assignment Outline

1. Data types
2. Fraction
   - Definition
   - Operations
   - Rules
   - Specs & Tasks
   - Sample Test
3. `Int<N>`
   - Definition
   - Operations
   - Specs & Tasks
4. Polynomial
   - Definition
   - Operations
   - Specs & Tasks
5. Mixed
   - Tasks

Assignment SE-A-03

Partha Pratim Das

Objective & Outline

Data types

Fraction
- Definition
- Operations
- Rules
- Specs & Tasks
- Sample Test

Int<N>
- Definition
- Operations
- Specs & Tasks

Polynomial
- Definition
- Operations
- Specs & Tasks

Mixed
- Tasks

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

Assignment SE-A-03
Partha Pratim Das
Objective & Outline
Data types
Fraction
Definition
Operations
Rules
Specs & Tasks
Sample Test
Int<N>
Definition
Operations
Specs & Tasks
Polynomial
Definition
Operations
Specs & Tasks
Mixed
Tasks

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

Assignment SE-A-03

Partha Pratim Das

Objective & Outline

Data types

Fraction
- Definition
- Operations
- Rules
- Specs & Tasks
- Sample Test

Int<N>
- Definition
- Operations
- Specs & Tasks

Polynomial
- Definition
- Operations
- Specs & Tasks

Mixed
- Tasks

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

Objective & Outline

Data types

Fraction
- Definition
- Operations
- Rules
- Specs & Tasks
- Sample Test

Int<N>
- Definition
- Operations
- Specs & Tasks

Polynomial
- Definition
- Operations
- Specs & Tasks

Mixed
- Tasks

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

Specifications of Fraction UDT

Assignment SE-A-03

Partha Pratim Das

Objective & Outline
Data types
Fraction
Definition
Operations
Rules
Specs & Tasks
Sample Test
Int<N>
Definition
Operations
Specs & Tasks
Polynomial
Definition
Operations
Specs & Tasks
Mixed
Tasks

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

```
template<typename T = int>
class Fraction { // Implicit assertion for proper fraction: gcd(|n_|, d_) = 1
    T n_; // numerator
    T d_; // denominator
}; // T is an integral type like int, char, short, long, etc.
```

- `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
  - Advanced Assignment 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)

Tasks for Fraction UDT

- Design an interface for `Fraction<T>` with appropriate overloads of operators wherever available
- Implement `class Fraction<T>`
- Test `class Fraction<int>` with applications for:
  - Pass Tests
  - Fail Tests
  - Complex Mixed Tests
- Use proper code organization in headers and source files
- Nicely comment the class definition, implementation and the test applications with your design choices, implementation considerations, possible errors, caveats etc.
- No separate documentation is needed

Sample Test Applications

```
#include <iostream>  // This code is unnaturally compacted to fit into one slide
using namespace std; // Do not follow this style. Format your code properly
#include "Fraction.h"
typedef Fraction_<int> Fraction; // template<typename T = int> class Fraction_;
void PassTest() { cout << ":::PASS TESTS:::" << endl << endl; // :::PASS TESTS:::
    Fraction fa(5, 3); cout << "Fraction fa(5, 3) = " << fa << endl; // Fraction fa(5, 3) = 5/3
    Fraction fb(7, 9); cout << "Fraction fb(7, 9) = " << fb << endl; // Fraction fb(7, 9) = 7/9
    cout << "fa + fb = " << (fa + fb) << endl;                       // fa + fb = 22/9
}
void FailTest() { cout << ":::FAIL TESTS:::" << endl << endl; // :::FAIL TESTS:::
    try { cout << "Fraction(1, 0): "; Fraction f1(1, 0); }
    catch (const char* s) { cout << s << endl; } // Fraction(1, 0): Fraction w/ Denominator 0 is undefined
    Fraction f1(5, 12), f2(0, 1), f3;
    try { cout << "Binary Divide: f3 = " << f1 << " / " << f2 << ": ";
        f3 = f1 / f2; cout << f3 << endl;
    }
    catch (const char* s) { cout << s << endl; } // Binary Divide: f3 = 5/12 / 0: Divide by 0 is undefined
}
void MixedTest() { cout << ":::MIXED TESTS:::" << endl << endl; // :::MIXED TESTS:::
    Fraction f1(2, 3), f2(8), f3(5, 6), f4;
    cout << "f1 = " << f1 << " f2 = " << f2 << " f3 = " << f3 << " f4 = " << f4 << endl;
    // f1 = 2/3 f2 = 8 f3 = 5/6 f4 = 1
    f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3;
    cout << "f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3 = " << f4 << endl;
    // f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3 = -1097/165
}
```

Objective & Outline

Data types

Fraction
Definition
Operations
Rules
Specs & Tasks
Sample Test

Int<N>
Definition
Operations
Specs & Tasks

Polynomial
Definition
Operations
Specs & Tasks

Mixed
Tasks

Int<N> UDT

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

Assignment SE-A-03
Partha Pratim Das
Objective & Outline
Data types
Fraction
Definition
Operations
Rules
Specs & Tasks
Sample Test
Int<N>
Definition
Operations
Specs & Tasks
Polynomial
Definition
Operations
Specs & Tasks
Mixed
Tasks

- 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` $= -2^{N-1}$ ... `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

Specifications of Int<N> UDT

- 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)`. 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>` 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)
```

Tasks for Int<N> UDT

Assignment SE-A-03

Partha Pratim Das

Objective & Outline

Data types

Fraction
- Definition
- Operations
- Rules
- Specs & Tasks
- Sample Test

Int<N>
- Definition
- Operations
- Specs & Tasks

Polynomial
- Definition
- Operations
- Specs & Tasks

Mixed
- Tasks

- Design an interface for `Int<N>` with appropriate overloads of operators wherever available
- Implement `class Int<N>`
- Test `class Int<4>` with applications for:
  - Pass Tests
  - Fail Tests
- Use proper code organization in headers and source files
- Nicely comment the class definition, implementation and the test applications with your design choices, implementation considerations, possible errors, caveats etc.
- No separate documentation is needed

Polynomial UDT

Polynomial UDT

Polynomials

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

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \sum_{i=0}^{n} a_ix^i$$

- 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
}
```

Operations of Polynomial

- 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,

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

- 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 $n \geq m$

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

  Hence,

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

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

Specifications of Polynomial UDT

Assignment SE-A-03

Partha Pratim Das

Objective & Outline

Data types

Fraction
- Definition
- Operations
- Rules
- Specs & Tasks
- Sample Test

Int<N>
- Definition
- Operations
- Specs & Tasks

Polynomial
- Definition
- Operations
- Specs & Tasks

Mixed
- Tasks

- `Poly<T>` should support the following operations:
  - Construction, Destruction and Copy Operations
  - Unary Arithmetic Operations: Preserve (Sign), Negate
  - Binary Arithmetic Operations: Add and Subtract
  - Advanced Assignment Operations: Add and Subtract
  - Evaluation Operation: `T x; Poly<T> p; p(x);`
  - IO Operations: Read and Write

Tasks for Polynomial UDT

- Design an interface for `Poly<N>` with appropriate overloads of operators wherever available
- Implement `class Poly<N>`
- Test `class Poly<int>` with applications for:
  - Pass Tests
  - Fail Tests
- Use proper code organization in headers and source files
- Nicely comment the class definition, implementation and the test applications with your design choices, implementation considerations, possible errors, caveats etc.
- No separate documentation is needed

Mixed UDT

Mixed UDT

Tasks for Mixed UDT

- Test the following applications:
  - Test `Fraction<Int<4> >`
  - Test `Poly<Int<4> >`
  - Test `Poly<Fraction<Int<4> > >`
- Use proper code organization in headers and source files
- Nicely comment the integration and the test applications with your design choices, implementation considerations, possible errors, caveats etc.
- No separate documentation is needed