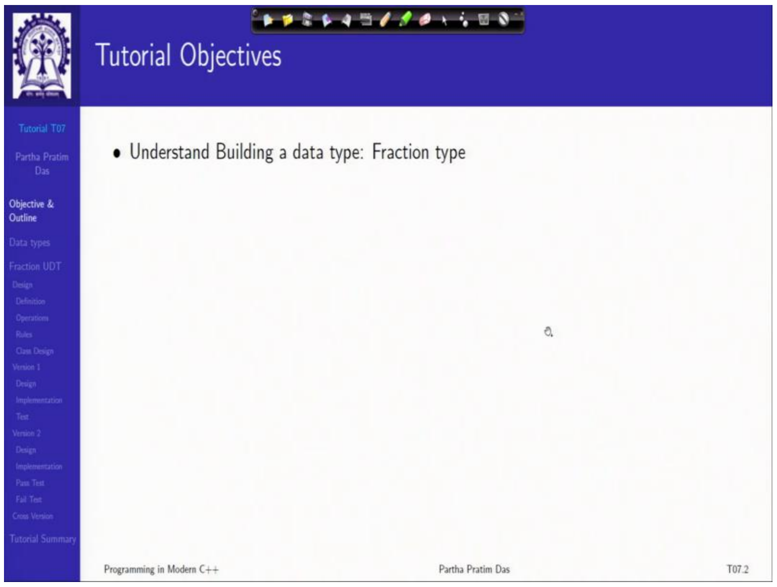
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 07 - Lecture 45
How to design a UDT like built-in types?
Part 1: Fraction UDT

(Refer Slide Time: 0:34)

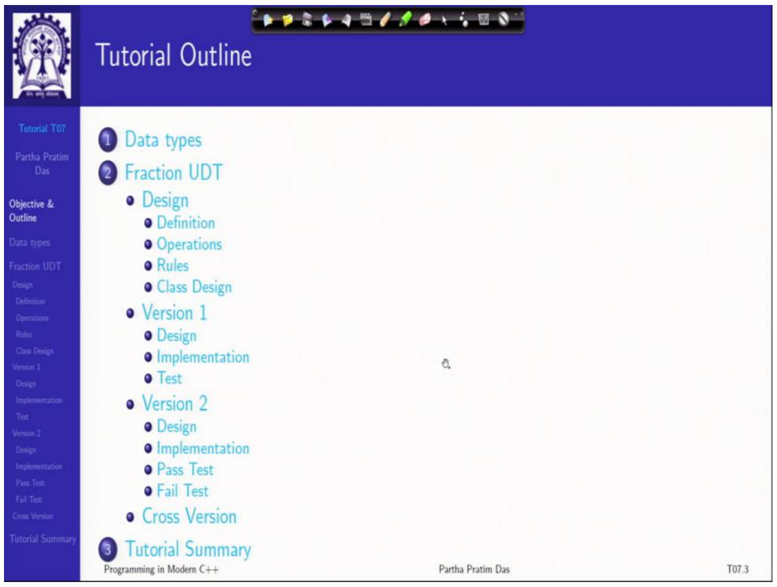


Tutorial Objectives

- Understand Building a data type: Fraction type

Tutorial T07 · Partha Pratim Das · Objective & Outline · Data types · Fraction UDT · Design · Definition · Operations · Rules · Class Design · Version 1 · Design · Implementation · Test · Version 2 · Design · Implementation · Pass Test · Fail Test · Cross Version · Tutorial Summary

Programming in Modern C++ · Partha Pratim Das · T07.2



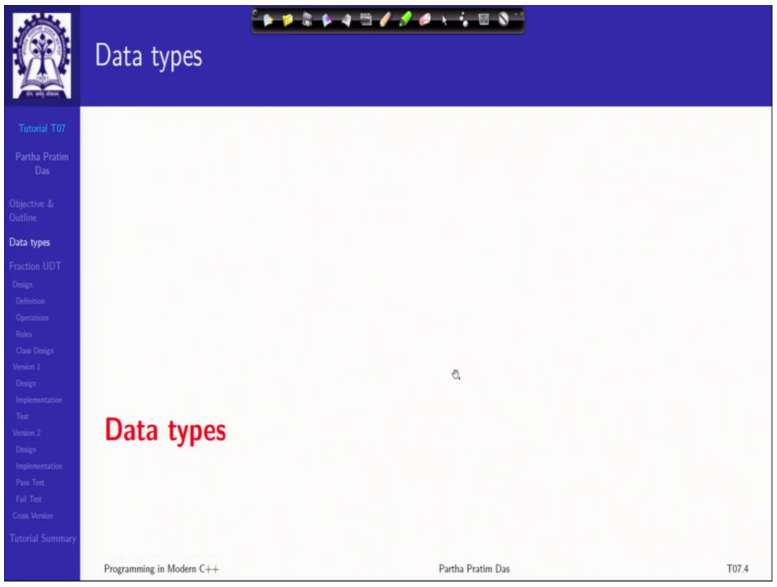
Tutorial Outline

1. Data types
2. Fraction UDT
   - Design
     - Definition
     - Operations
     - Rules
     - Class Design
   - Version 1
     - Design
     - Implementation
     - Test
   - Version 2
     - Design
     - Implementation
     - Pass Test
     - Fail Test
   - Cross Version
3. Tutorial Summary

Programming in Modern C++ · Partha Pratim Das · T07.3

Welcome to Programming in Modern C++, we are going to discuss tutorial 7. It will be on how to design a user defined type that behaves like a built-in type? This is the first part where we will discuss about a user defined type fraction. So, that is a basic objective. And this is the outline, which will be there on the left.

(Refer Slide Time: 0:60)



Data types

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

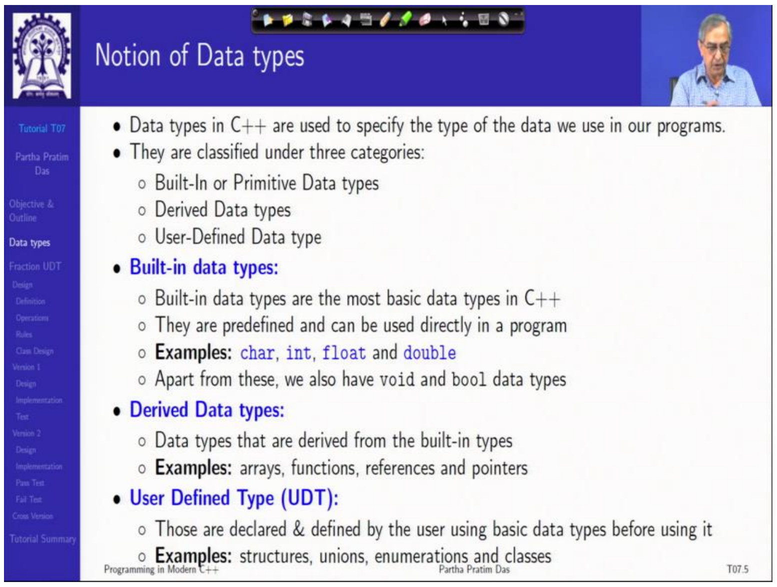
Fail Test

Cross Version

Tutorial Summary

Data types

Programming in Modern C++ Partha Pratim Das T07.4



Notion of Data types

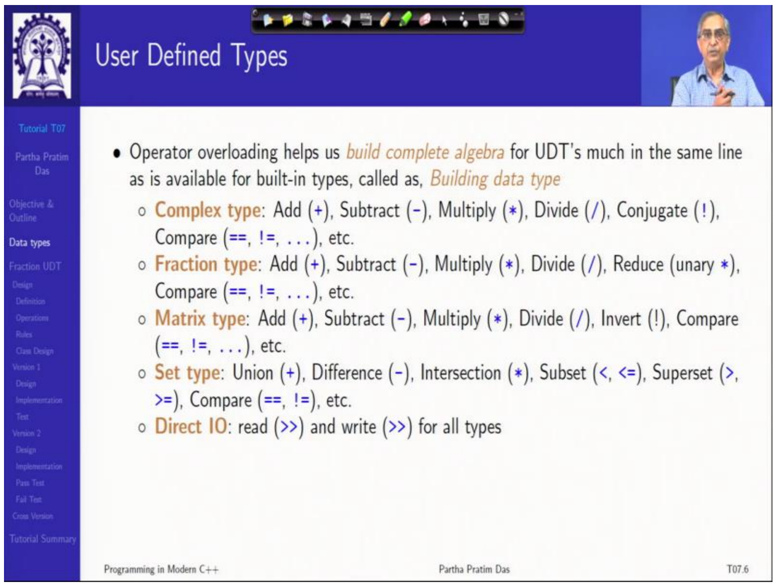
- Data types in C++ are used to specify the type of the data we use in our programs.
- They are classified under three categories:
  - Built-In or Primitive Data types
  - Derived Data types
  - User-Defined Data type
- **Built-in data types:**
  - Built-in data types are the most basic data types in 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

Programming in Modern C++ Partha Pratim Das T07.5

Now, before we get into this design, let me remind you of a few things that you have done in the modules so far, you have seen that there are several data types in C++, which are used to specify the type of data constants as well as data operations that we use in our program. And if I look into the different categories of data types that are available to us, certainly there are a number of primitive or built-in data types like char, int, available to us.

There are derived data types which are built on top of these, like arrays, structure and so on. And then there are user defined types, user defined types are the pure addition, and one of the strongest features of C++. So, in building types, as we have different kinds of char, int, float bool, even we have void. In derived types, arrays, functions, even references we have. In user defined types, the user has the liberty to define the set of nominal values of the data type as well as its operations and its properties actions.

(Refer Slide Time: 2:33)



User Defined Types

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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

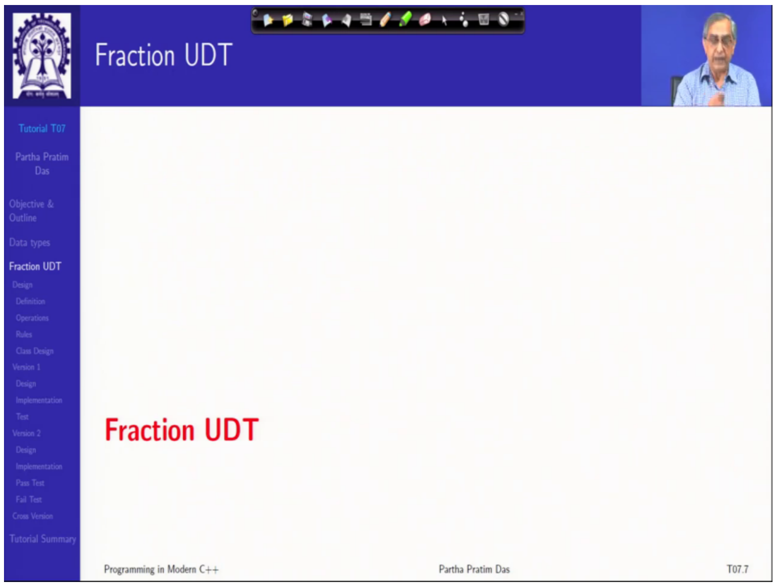
Programming in Modern C++ Partha Pratim Das T07.6

And this becomes really useful. Because now, you do not need to, you know, rely on a whole lot of functions that you had to write in C, to realize the functionality of the type of data that you are dealing with. But you can build for a particular concept of a data type, you can build it pretty much like the built-in types that we have. So, we have often taken example, in different places in the module of operations being done on the complex type, like add, subtract, multiply, divide, take conjugate off, and so on, so forth.

In fact, conjugate type is also where I mean, I am sorry, the complex type is also available in the standard library. In addition, we can ask for a Fraction type, which deals with fraction and does several algebraic operations, there could be matrix types, which deal with matrices, their addition, subtraction, inversion, squared matrices, vectors, and so on, so forth. There could be a set type, for union, intersection difference, and so on. There could be several other types like a polynomial type, there could be a rectangle type.

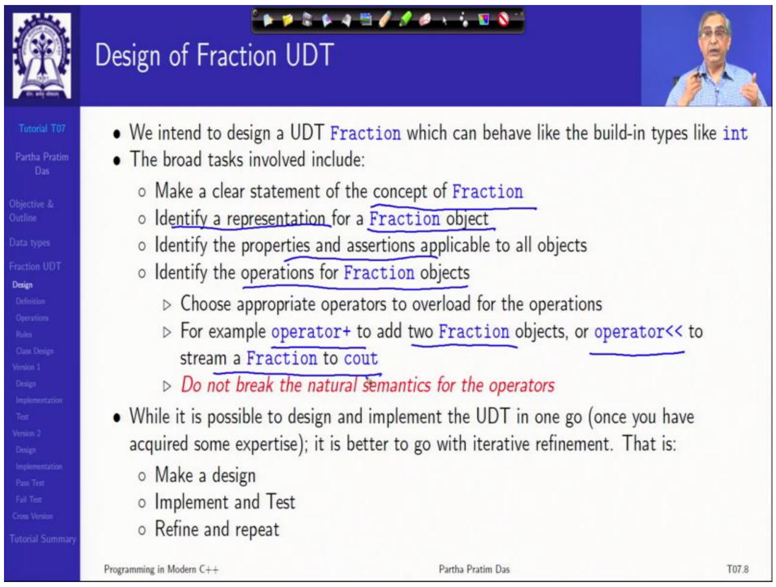
Now, the basic point here is that when you program with such entities in C, you necessarily write a bunch of functions. And you have to manage the data between these functions. But here, using the tricks of user defined type definition, and build up, you will be able to build a type, which behave almost exactly as your int type or your char type and so on. You can do direct IOs with them and so on.

(Refer Slide Time: 4:35)


Fraction UDT

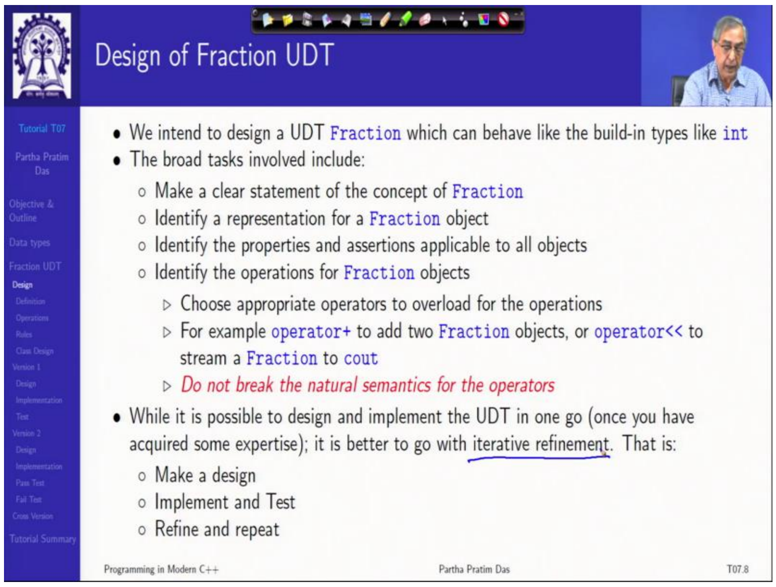
Fraction UDT

Programming in Modern C++ Partha Pratim Das T07.7


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

Programming in Modern C++ Partha Pratim Das T07.8



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

Programming in Modern C++ Partha Pratim Das T07.8

So, the main objective here is to illustrate the process of building such a user defined type, using a Fraction type, because that is known to all of us. So, what are the issues involved in this design? To design a utility fraction, which will behave like an int, the broad tasks, first task could involve that we need to have a clear idea about the concept of fraction what is mean by fraction? We must be clear about that. We must identify a proper representation of a Fraction object in the system in the memory. So, representation is important.

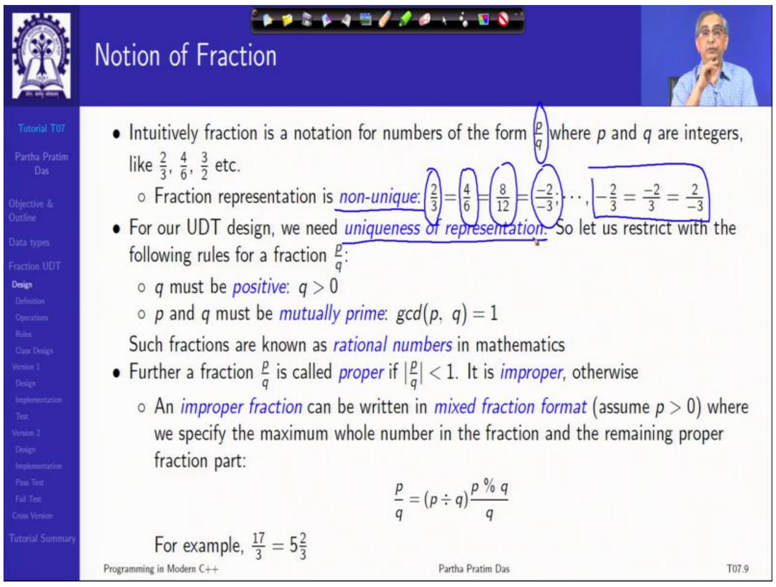
We have to identify the properties and assertions that are applicable for all objects, what is a valid object? What is not a valid object? And so on. And finally, we need to identify the operations for the Fraction objects. Once we do that, then we will be able to go towards the design of a Fraction UDT. Now, obviously, for the operations, you have to choose the appropriate operators to overload, you could just use you know member functions for doing different operations, but we will see why having overloaded operators will really help.

And when you make such a choice, you have to be careful for example, you can choose operator plus to add two fractions or operator output streaming to stream a Fraction object to cout but be careful not to break the natural semantics of the operator. What I mean by that, for example, you say operator +, but actually you are trying to do a multiplication of two fractions, do not do that, because that breaks the meaning and the overall value of making a UDT loses.

Because the whole idea is looking at the operator, user should be able to identify without any documentation or anything as to what is intended. So, when we talk about string type UDT, which is there in the standard library, we say that operator plus basically means concatenation and certainly some operators may not be defined for a certain UDT, like for a string type operator - probably does not make much sense.

The question is how do should you go about doing the design? Would you be able to conceptualize everything, workout every detail and put down a design in one go, yes, if you are an expert, you will be able to do that. But the whole purpose of this course, such tutorial is to create the expertise. So, what I am going to illustrate is not a one shot, design and go, but we will go by what is known as iterative refinement. That is, we will make a design, which is more or less okay. We will implement it, test it, identify the shortcomings and then refine and repeat this process. Here in this tutorial, I will show that at least in two stages.

(Refer Slide Time: 7:53)



Notion of Fraction

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

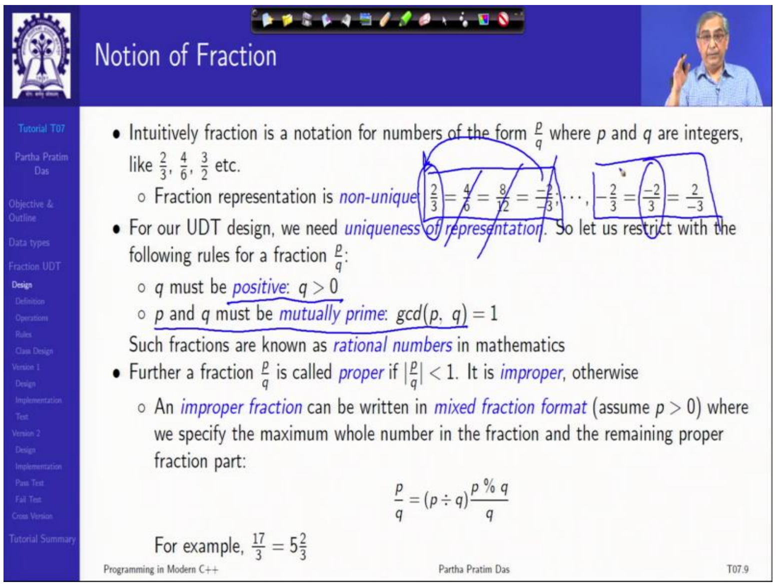
- Intuitively fraction is a notation for numbers of the form $\frac{p}{q}$ where p and q are integers, like $\frac{2}{3}, \frac{4}{6}, \frac{3}{2}$ etc.
  - Fraction representation is *non-unique*: $\frac{2}{3} = \frac{4}{6} = \frac{8}{12} = \frac{-2}{-3}; \cdots, -\frac{2}{3} = \frac{-2}{3} = \frac{2}{-3}$
- 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, $\frac{17}{3} = 5\frac{2}{3}$

Programming in Modern C++ Partha Pratim Das T07.9



Notion of Fraction

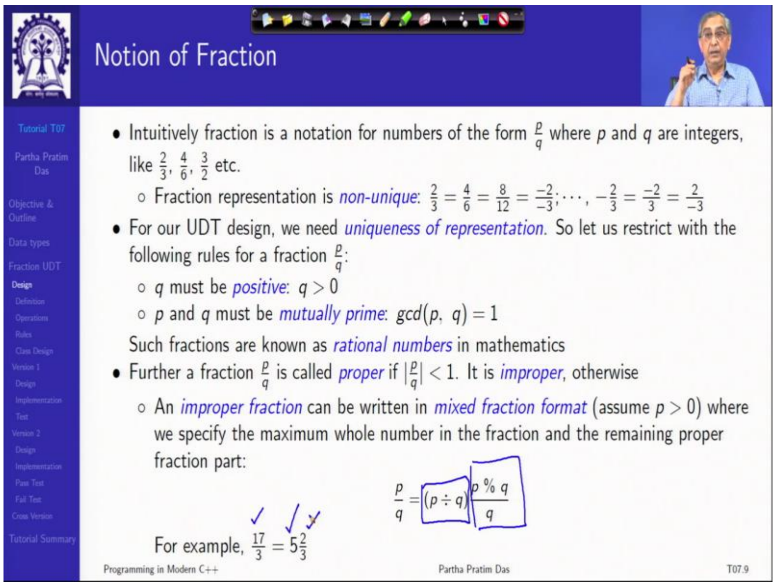
- Intuitively fraction is a notation for numbers of the form $\frac{p}{q}$ where p and q are integers, like $\frac{2}{3}, \frac{4}{6}, \frac{3}{2}$ etc.
  - Fraction representation is *non-unique*: $\frac{2}{3} = \frac{4}{6} = \frac{8}{12} = \frac{-2}{-3}; \cdots, -\frac{2}{3} = \frac{-2}{3} = \frac{2}{-3}$
- 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, $\frac{17}{3} = 5\frac{2}{3}$

Programming in Modern C++ Partha Pratim Das T07.9



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:

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

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

Programming in Modern C++ Partha Pratim Das T07.9

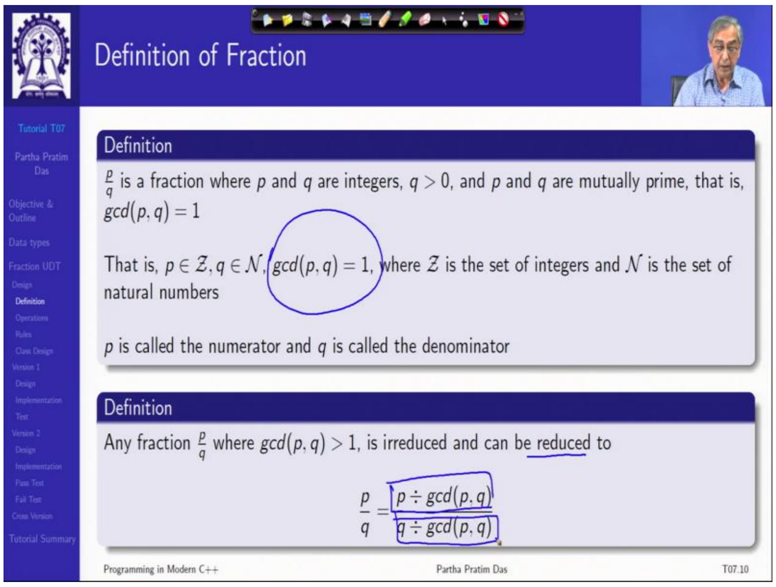
So, first a notion of the fraction every one of us understands what is the fraction which is a number of the form p/q, where p and q are integers. The important thing about fraction is, it is, it has a non-unique representation that is 2/3, 4/6, 8/12, -2/-3 all mean the same fraction, similarly, all of these. So, that is something which is going to create a lot of problem in terms of a UDT, because why how will it represent in the system? How will I represent it when I stream it to the output? And so on.

So, what we need is we need to uniquefy the representation, we need to make the representation unique. And for that we do two things one is out of p and q we make q necessarily positive, that is the denominator cannot be a negative number, this is the assumption we make, and we make that p and q must be mutually prime, that is the greatest common divisor gcd of p and q must be one.

So, that if you put that then out of all these by the first condition, this representation will go out, it is same as this. And by the second condition these two representations will also go out. So, you are left with only one unique representation for 2/3. Similarly, if you look at these three, the only unique representation is -2/3. So, this is also what we know by rational numbers in mathematics and that is what we are going to represent here in terms of the fraction.

Another point to be noted that a fraction is called proper if its absolute value is less than 1, otherwise it is called improper. So, there is some whole number part in that fraction, in the improper fraction. So, if I have an improper fraction then I can take out the whole number part and the remaining part I represent as a fraction, so, this is called mixed fraction format. So, 17/3 is 5(2/3), 5 is the whole part 5 * 3 is 15 + 2 is 17. So, this is the basic notion of the fraction that we deal with.

(Refer Slide Time: 10:20)



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

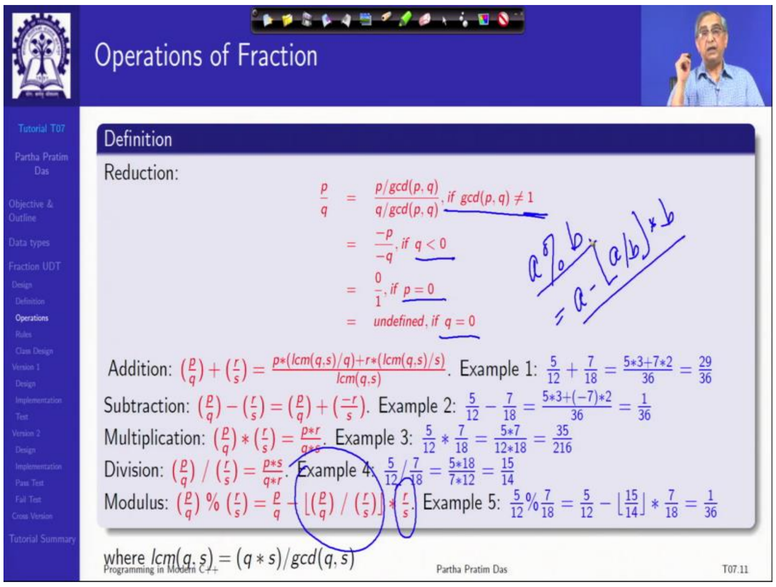
Definition

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

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

So, formally speaking, a fraction p/q for our purpose, where p and q are integer, q is greater than 0, p and q are mutually prime, p is called the numerator, q is called the denominator, and if a fraction does not satisfy this condition, then it is called an irreduced fraction. So, a irreduced fraction can be reduced by dividing the numerator and the denominator both by the gcd of the fraction.

(Refer Slide Time: 10:52)



Operations of Fraction

Definition

Reduction:

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

$a \% b = a - \lfloor a/b \rfloor * b$

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: $\frac{5}{12} / \frac{7}{18} = \frac{5*18}{7*12} = \frac{15}{14}$

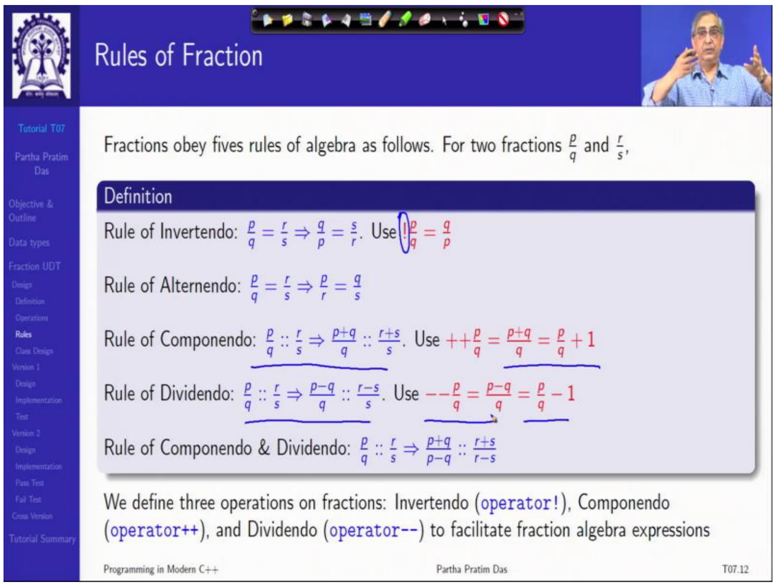
Modulus: $(\frac{p}{q}) \% (\frac{r}{s}) = \frac{p}{q} - \lfloor (\frac{p}{q}) / (\frac{r}{s}) \rfloor * \frac{r}{s}$. Example 5: $\frac{5}{12} \% \frac{7}{18} = \frac{5}{12} - \lfloor \frac{15}{14} \rfloor * \frac{7}{18} = \frac{1}{36}$

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

There are several operations available for example, reduction by itself is an operator if the gcd is not 1, then you can make the gcd 1 if q is not positive, then you can make q positive, if p is 0 q can be anything. So, you force q to be 1 and if p is q is 0, the denominator is 0 then the fraction is undefined. Given that we have a number of rules of fraction which you know from the arithmetic like addition, subtraction, multiplication division or even remainder.

So, by remainder what you mean is, you divide one fraction by the other first fraction by the second take the integer part of it multiplied with the second fraction and subtract from the first fraction. So, this basically is a % b is a minus read in a context of the fraction is what gives you the remainder here are examples given.

(Refer Slide Time: 11:59)



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 $\boxed{!}\frac{p}{q} = \frac{q}{p}$

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

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

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

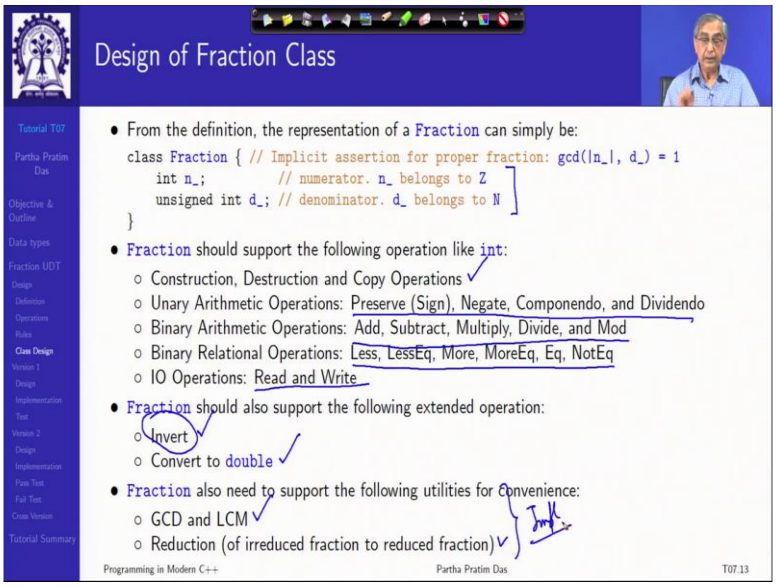
Rule of Componendo & Dividendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p+q}{p-q} :: \frac{r+s}{r-s}$

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

Programming in Modern C++ Partha Pratim Das T07.12

So, there are some rules of fraction also, like we know of invertendo, that if you flip the numerator and denominator of two fractions which are equal then they remain equal. So, we define that to be a flip operation or inversion operation. Componendo is adding 1 to a fraction, which is basically prefix or postfix operator increment operator, Dividend is subtracting 1, so, it is prefix or postfix subtraction operation. So, these are very naturally map on to the different operators that we have in our integer domain.

(Refer Slide Time: 12:45)



Design of Fraction Class

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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) ✓

Impl.

Programming in Modern C++ Partha Pratim Das T07.13

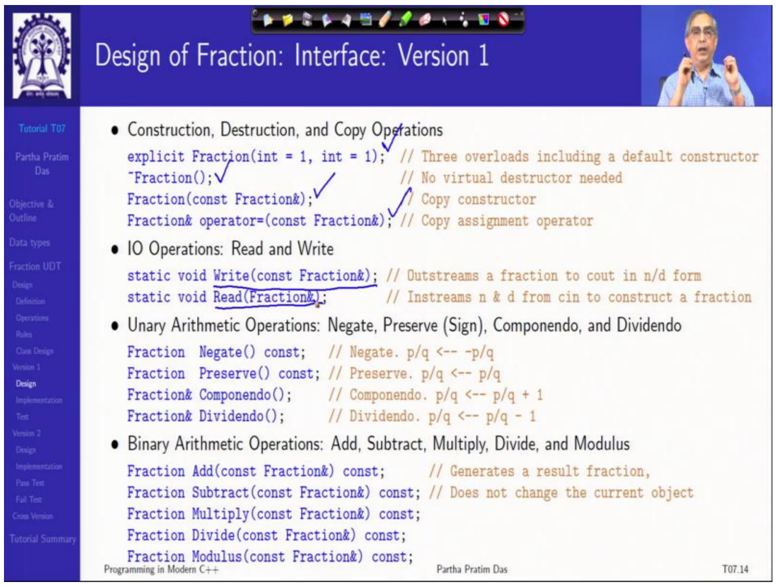
So, with that if we have to now talk about the design of the Fraction class, then certainly there will be two members. n_ for standing for numerator, which is of type int, because it can be signed and d_ which is the denominator, we take it as unsigned integer because we said that it has to be greater than 0 it cannot be negative. So, then what are the operations that we will have? Obviously, we will have the usual construction destruction copy, construction copy assignment and so on.

We can have different unary arithmetic operations like taking the negative of a fraction doing a componendo that is adding one subtracting one, we can do binary operations of add, subtract all that, we can do Boolean relational operations that is comparing two fractions less, less equal to and so on, certainly we can do read and write, we can invert a fraction, fraction that is flip it numerator and denominator, we can convert it to double and so on.

So, these are some of the very quickly identifiable operations of a Fraction type that we come to which have close simile, in terms of the interest not in every case, for example, it does not have an invert kind of functionality. But most others are basically borrowed from the int data type.

And along with that, we need some convenience functions like GCD and LCM, GCD because we need to check for the validity of whether a fraction is reduced. Otherwise, we will have to reduce it, LCM is required for adding subtracting functions and so on least common multiplier and we need a reduction operation. These are basically what your implementation is going to need. So, this is your basic design that you identify. So, that is the first step we have clarified the notion, we have decided on the representation, we have decided on the set of operations and their properties that we want to realize.

(Refer Slide Time: 15:03)



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
```

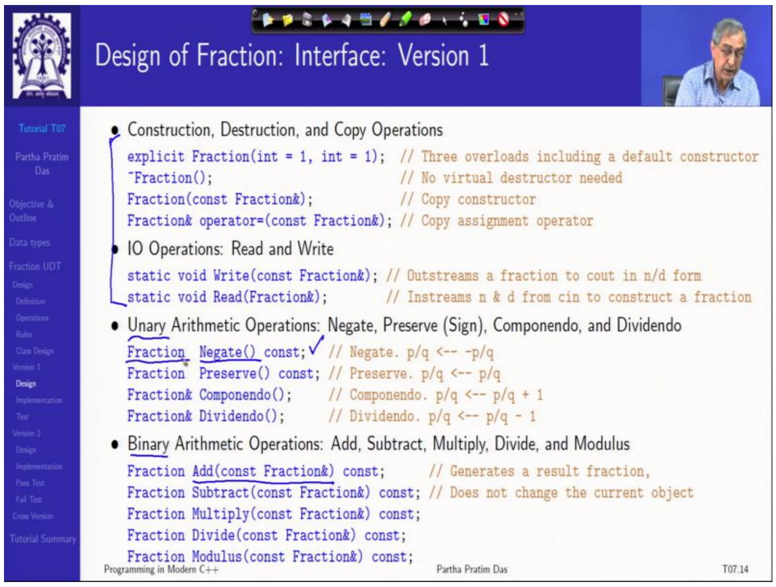
- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
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;
```

Programming in Modern C++ Partha Pratim Das T07.14



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
```

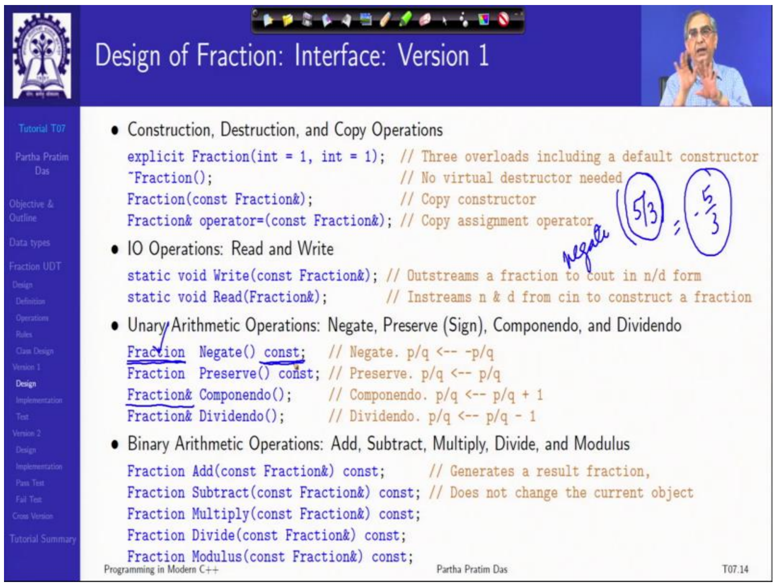
- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
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;
```

Programming in Modern C++ Partha Pratim Das T07.14


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;
```

negate $(5/3) ; -\frac{5}{3}$

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

Programming in Modern C++ Partha Pratim Das T07.14

So, with that, let us, let me do a first version of the interface design for the Fraction. So, I need a, we have a Fraction constructor, naturally, I need two numbers to construct a Fraction, so I have defaulted both of them to 1. So, that I may not give any parameter, in which case it will give me the unit fraction 1/1, which is the default constructor. Otherwise, I can give 1 the first parameter, or I can give both the parameters, there is a destructor, which does nothing copy constructor, copy assignment operators, these are routine.

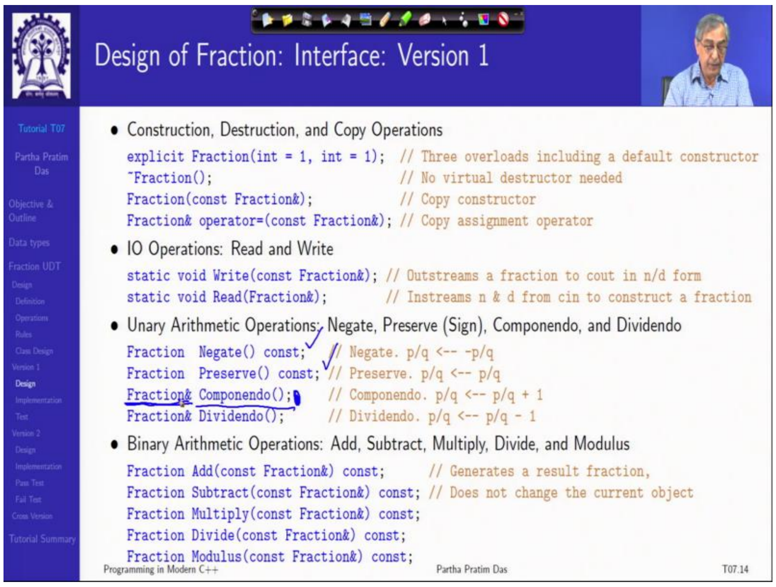
Naturally, I need a read write function, which will take a fraction and write it to the cout. Similarly, a read function which will read from the reader fraction from the cout as a pair of numbers. So, here you can see that, at the first go, I am not attempting to get into the different operators, I am just identified each and every operation, I am giving it a name for the member function and identifying what will be the signature.

Now, up to this point, these things are almost generic, and irrespective of whether it is a Fraction or a complex or any other we will have similar functionalities, but now we have say, I want to do a negate. So, when I decide on an interface member function, there are primarily based on the functionality, there are primarily three things to decide, one is what is the parameter? So, if it is a unary operation, then it does not show a parameter as a member function, because the object itself is the parameter on which it applies.

If it is a binary operation, then it takes only the second operand as a parameter, the first operand is the object itself on which this member function is invoked, you already know that. So, this is how you decide on the parameters, then you decide on the what is the return type? The return type typically, here is either you have returned by value, when will you have that when that operation computes something, which is a new object, and your current object is not changing, probably not changing or it may be changed, but probably it will the current object will not change and you are having a new object.

So, here, when I do a negate, say, I have a fraction 5/3, and I want to do a negate on that. So, I should get a fraction -5/3. So, this is a new fraction, which has to be created. So, I have a return by value and the original fraction should not be disturbed. So, I write a const. So, you will typically see that if it is not necessarily always but typically, when you have a return by value from such member functions, you will that member function should preferably be constant.

(Refer Slide Time: 18:20)



Design of Fraction: Interface: Version 1

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

- 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
```

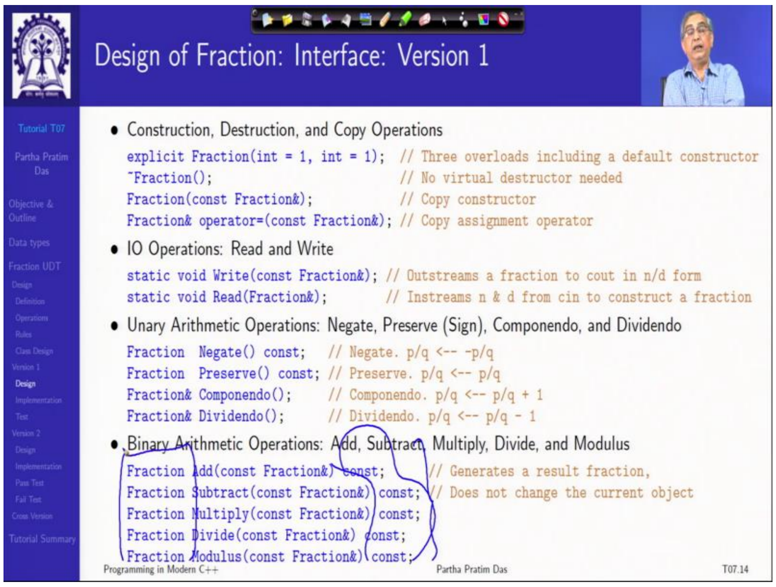
- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
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;
```

Programming in Modern C++ Partha Pratim Das T07.14



Design of Fraction: Interface: Version 1

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

- 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
```

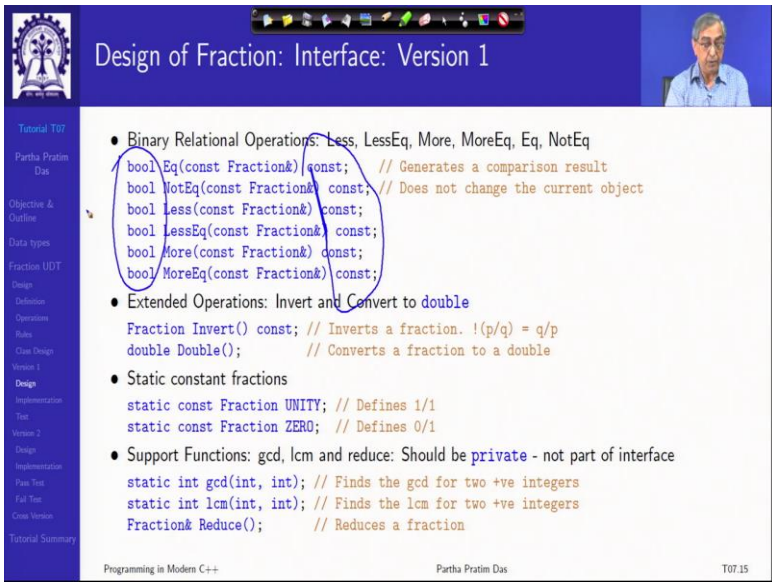
- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
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;
```

Programming in Modern C++ Partha Pratim Das T07.14

So, these are const. Whereas if I do componendo, I am adding 1 to the fraction itself. So, I want that fraction itself to change. Therefore, neither can it be const nor can it return a different fraction, it has to return itself. So, it has to return by reference. So, this is the basic consideration that you do for designing addition, subtraction, all of them, all of them are const member function, because you do not expect the parameter to change because of that, all of them return by value, because we expect the result to come as a returned object as a new object.

(Refer Slide Time: 19:06)



Design of Fraction: Interface: Version 1

Tutorial T07 · Partha Pratim Das · Objective & Outline · Data types · Fraction UDT · Design · Definition · Operations · Rules · Class Design · Version 1 · Design · Implementation · Test · Version 2 · Design · Implementation · Pass Test · Fail Test · Cross Version · Tutorial Summary

- 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
```

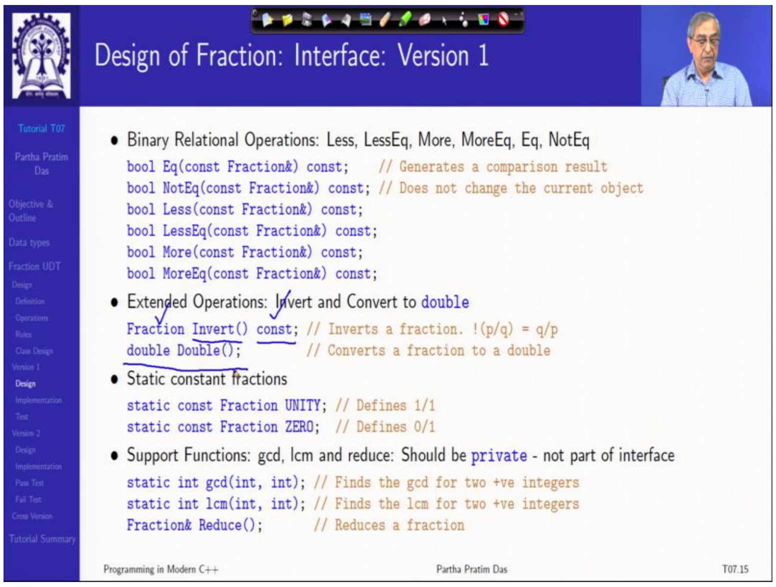
- Static constant fractions

```
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
```

Programming in Modern C++ · Partha Pratim Das · T07.15



Design of Fraction: Interface: Version 1

Tutorial T07 · Partha Pratim Das · Objective & Outline · Data types · Fraction UDT · Design · Definition · Operations · Rules · Class Design · Version 1 · Design · Implementation · Test · Version 2 · Design · Implementation · Pass Test · Fail Test · Cross Version · Tutorial Summary

- 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
```

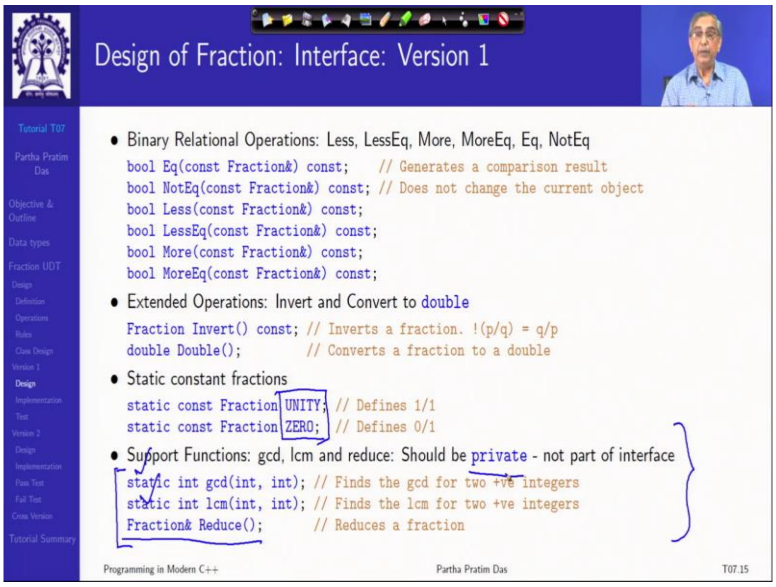
- Static constant fractions

```
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
```

Programming in Modern C++ · Partha Pratim Das · T07.15



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

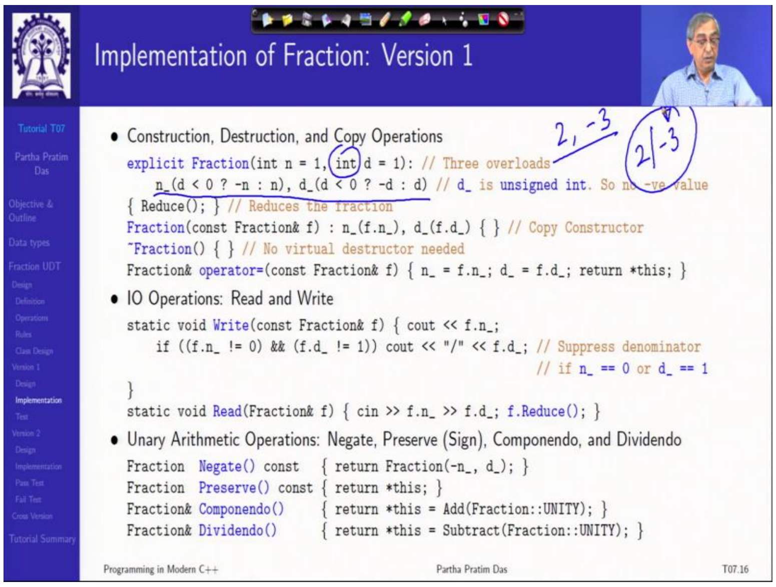
```
static int gcd(int, int); // Finds the gcd for two +ve integers
static int lcm(int, int); // Finds the lcm for two +ve integers
Fraction& Reduce();       // Reduces a fraction
```

Obviously, you can do equality inequality, all comparison checks, trivial to say that they are all constant member functions because they should not change the fractions that they are comparing. They will all return bool quite straightforward. If I invert a function, then, I am sorry, if I invert a fraction that take 1 by that fraction, then I will get a new fraction. So, it is written by value and it is a constant member function.

Double is a special member function I am saying which will take the current fraction and will return its value in as a double number. This is kind of conversion. In addition to that, we define some unity fraction and zero fraction, because they will come in ease for writing the code. And we have the support function for GCD, LCM. These are not dependent on your Fraction class. So, they are static members.

And we have a function which does reduction on the current Fraction object to bring it to the normalized, reduced form. Remember that these parts, these members are not a part of your interface, that is the user is not going to use this. So, you will typically have them as, as private and you may not mention that in the part of the interface just as together, I have written it here.

(Refer Slide Time: 20:51)



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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(); }
```

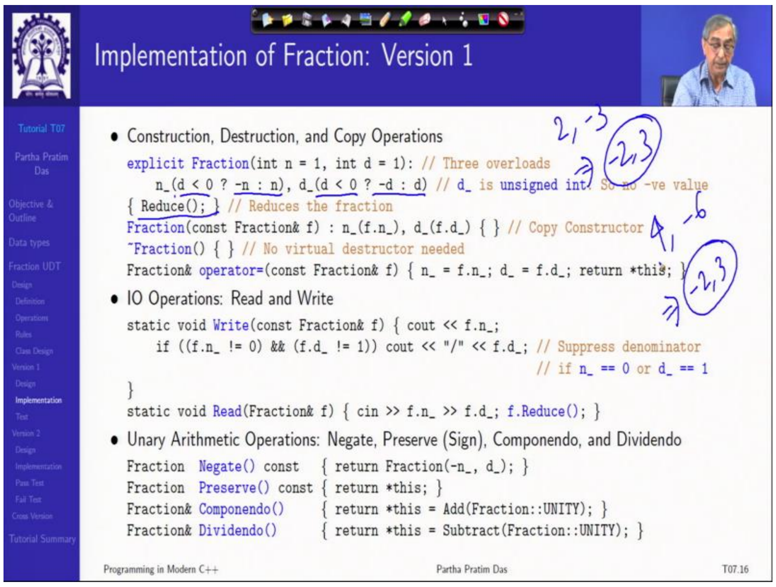
- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
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); }
```

2, -3

2|-3

Programming in Modern C++ Partha Pratim Das T07.16



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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); }
```

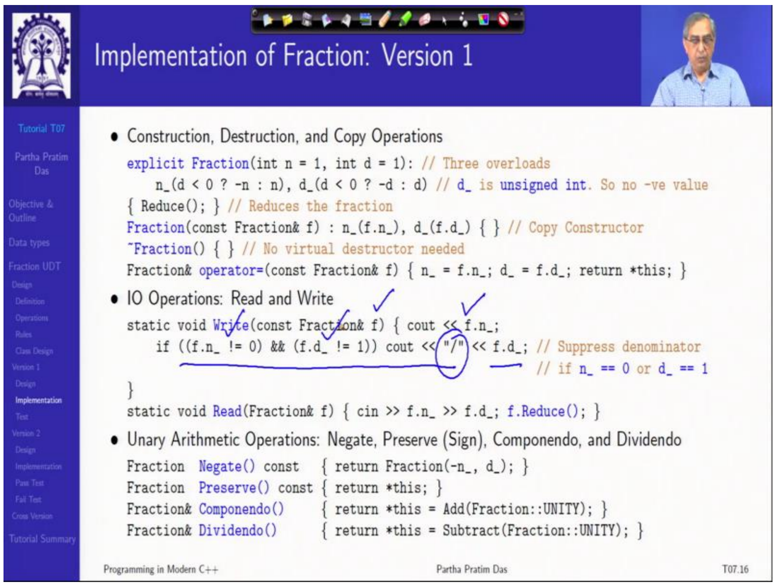
Programming in Modern C++ Partha Pratim Das T07.16

2, -3

$\Rightarrow$ -2, 3

4, -6

$\Rightarrow$ -2, 3



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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; }
```

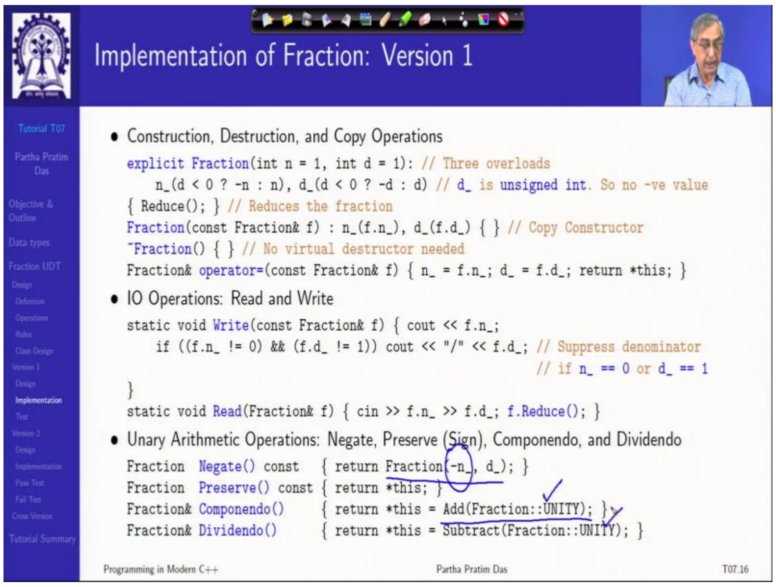
- IO Operations: Read and Write

```
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); }
```

Programming in Modern C++ Partha Pratim Das T07.16



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); }
```

Programming in Modern C++ Partha Pratim Das T07.16

So, having done that, the next is to, so I have a design now all the interface. So, now I will have to implement in terms of the two members data members n_ for the numerator and d_ for the denominator. So, these are very trivial implementations. For example, this is a constructor, which has to construct and do a reduce, you will see something interesting going on here, because when you call the constructor, the denominator is int.

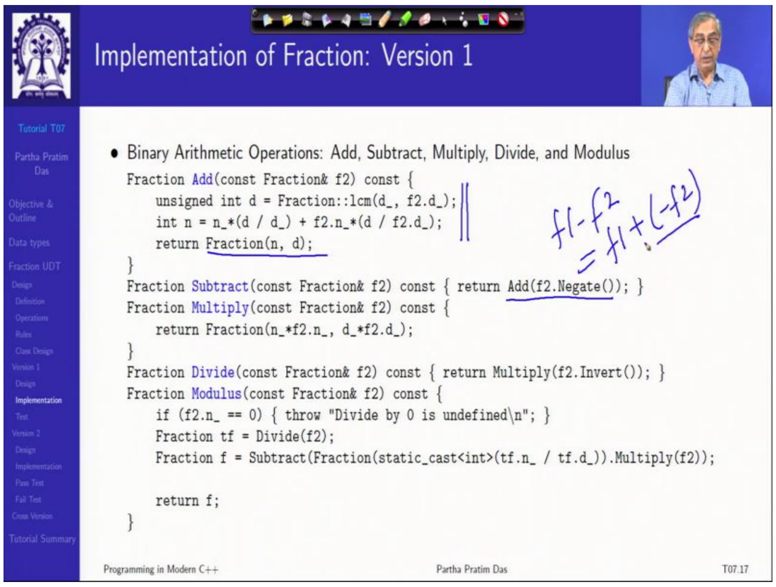
So, it is possible that I call the constructor with set 2 - 3, but I cannot have a fraction 2/-3 because the denominator is a signed number here, whereas in my representation is an unsigned integer, d_ is unsigned, so, I cannot take -3 and put it to d_. So, what I have to check before the data members are initialized that if d_ is negative, then I have to flip the sign. And if d_ is negative, then I have to flip the sign up. And also to keep, so, if it is given us 2 - 3, eventually I make it as - 2, 3 and construct and then do the reduction so that if I have 4 - 6, then I actually will get - 2, 3 in the representation.

There is nothing great in terms of the copy constructor because there are no resources destructor or copy assignment operator their routine, in terms of writing, you will write one component and then decide and write a slash to write the second component 2/3. But putting some conditions that if the numerator is 0, then do not write it as 0/1 or 0/5 it is better to write 0. Or if the denominator is 1, then it is a whole number. So, it is better to write just the whole number.

So, coming to negate, you just have to flip the sign of the numerator, in componendo you have to just add 1, you already now can see that why would we had this static constant

Fraction, I can add unity, I can subtract unity and I can very easily realize this different member functions.

(Refer Slide Time: 23:22)



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

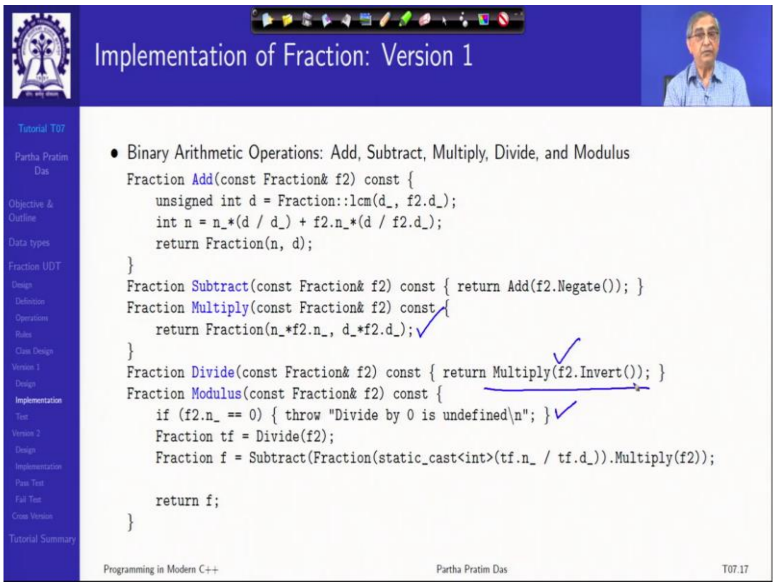
Cross Version

Tutorial Summary

- 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;
}
```

$f1 - f2$
$= f1 + (-f2)$



Implementation of Fraction: Version 1

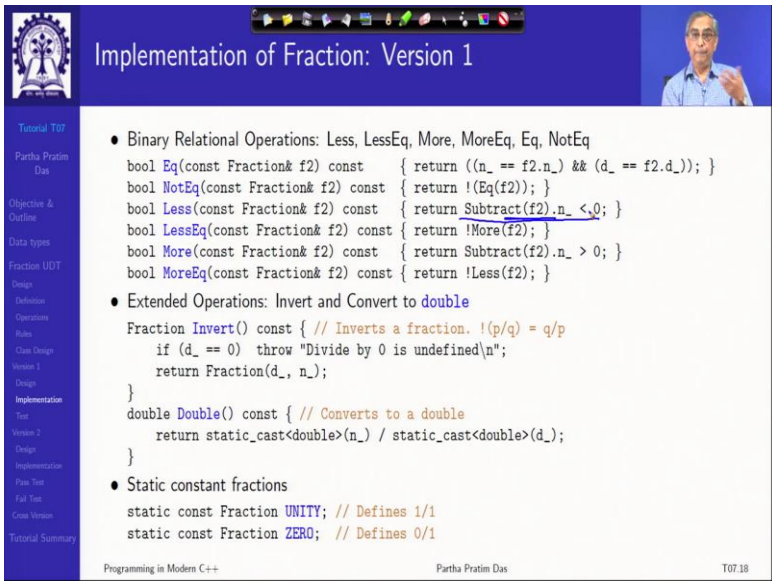
- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
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;
}
```

Programming in Modern C++ Partha Pratim Das T07.17

The Add function, you can go through carefully taking the LCM and so on. But it is exactly the formula as we have already given in the section of operations you can so that is just coded will give you this, you have a new Fraction created as a sum of the two Fractions and you return that. Subtract you can write in a similar way, but is a very compact way to write is you can negate f2, that is f1 - f2 is same as f1 + (-f2). So, you can just negate and use the add, which makes the code more compact and readable. Multiplication is simple. So, is division, I am again using invert to use multiplication for getting division. And this is the remainder you have to follow that formula for this that is nothing great in terms of C++ here.

(Refer Slide Time: 24:23)



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim Das

Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

- 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); }
```

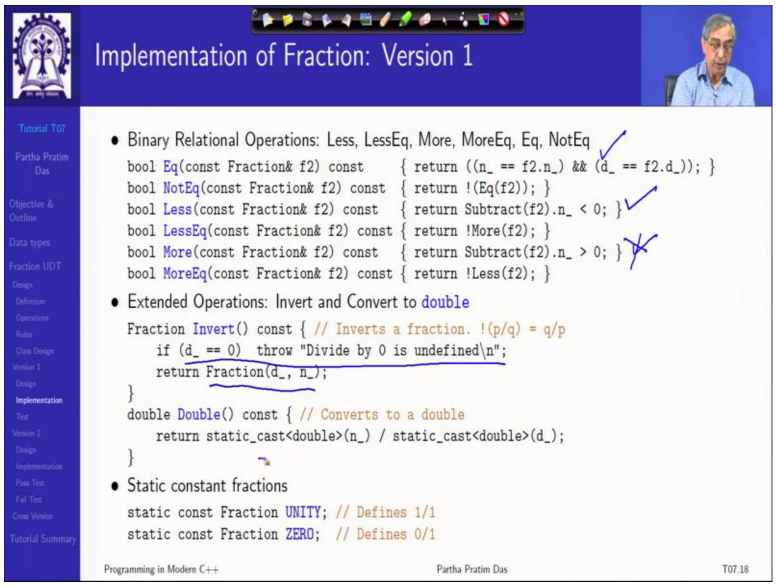
- Extended Operations: Invert and Convert to double

```
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
```

Programming in Modern C++ Partha Pratim Das T07.18



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_);
}
```

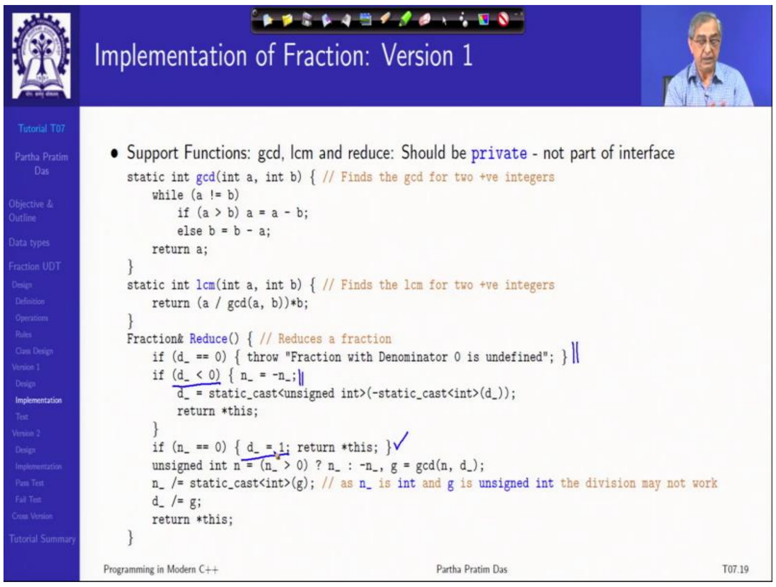
- Static constant fractions

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

Programming in Modern C++ Partha Pratim Das T07.18

Similarly, the comparison operators are very easy equality and inequality can be checked directly and for less or greater, you can use the subtract operation and see the sign of the subtraction. And you can see that one once you have some like this is one operation you implement, this is another operation you implement and this is a one the other three you can implement in terms of actually you can may not implement this also. If you implement two then everything else can be done as a combination of them. Similarly, for invert few have to remember that if d_ is 0, then you cannot invert a 0 fraction. So, you have to throw an exception, you can convert to a double and so on.

(Refer Slide Time: 25:16)



Implementation of Fraction: Version 1

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

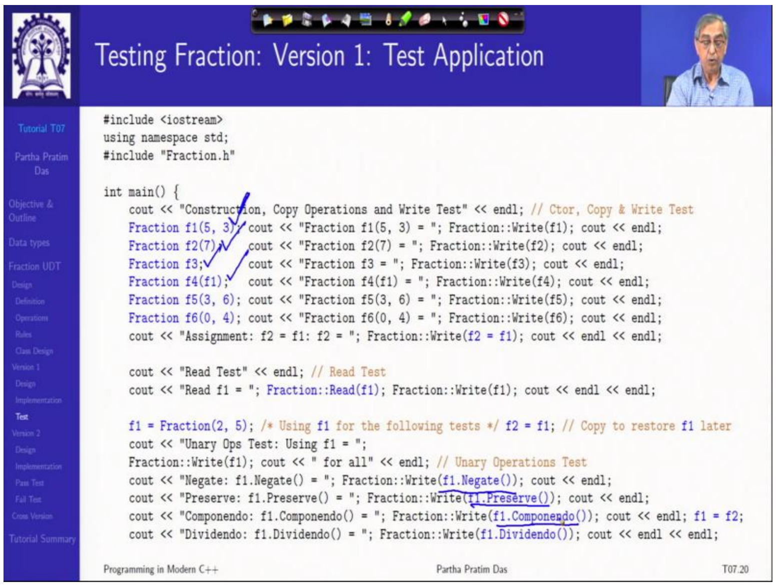
- Support Functions: gcd, lcm and reduce: Should be private - not part of interface

```
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;
}
```

Programming in Modern C++ Partha Pratim Das T07.19

So, this is a pretty straightforward implementation, this is just the code of GCD, LCM, which has routine code. And this is a reduce that the reduce reduction tells you that if, if d_ is 0, then it is an error. If d_ is negative, then you have to flip the numerator and as well as the denominator, if n_ is 0, that is numerator is 0, you have to force denominator to be 1 and if they are in reduced that is they have a GCD greater than 1 then have to take the GCD and divide both.

(Refer Slide Time: 26:06)



Testing Fraction: Version 1: Test Application

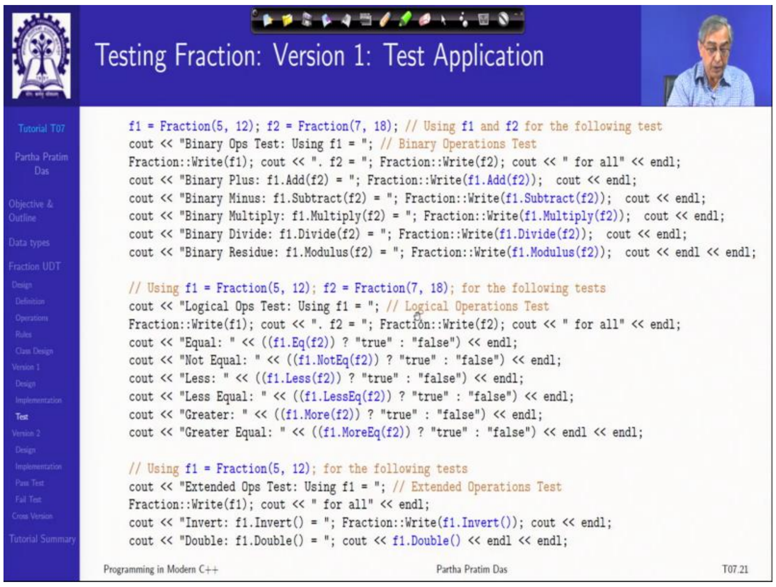
Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

```
#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;
```

Programming in Modern C++ Partha Pratim Das T07.20

So, your whole reduction process which is encoded in this. having done that the next thing you need to do for an UDT is to generate a test. So, in a test what you do? You write to every operation that you have written down, you write down sample cases for that. So, here is a sample case to construct a function with two parameters, fraction with two parameters with one parameter with no parameter a fraction by copying and so on, so forth. Similarly, you can you are trying out negate, preserve, componendo.

(Refer Slide Time: 26:43)

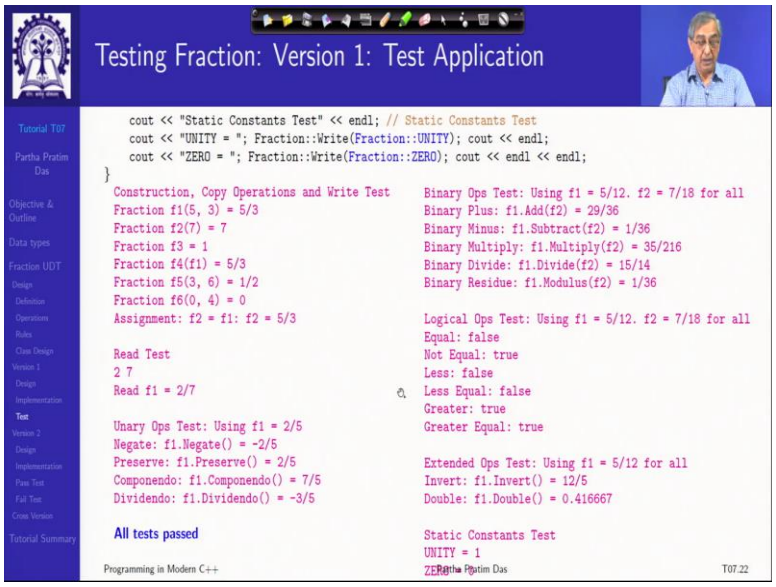


Testing Fraction: Version 1: Test Application

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

```
    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;
```

Programming in Modern C++ Partha Pratim Das T07.21



Testing Fraction: Version 1: Test Application

Tutorial T07
Partha Pratim Das
Objective & Outline
Data types
Fraction UDT
Design
Definition
Operations
Rules
Class Design
Version 1
Design
Implementation
Test
Version 2
Design
Implementation
Pass Test
Fail Test
Cross Version
Tutorial Summary

```
    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
```

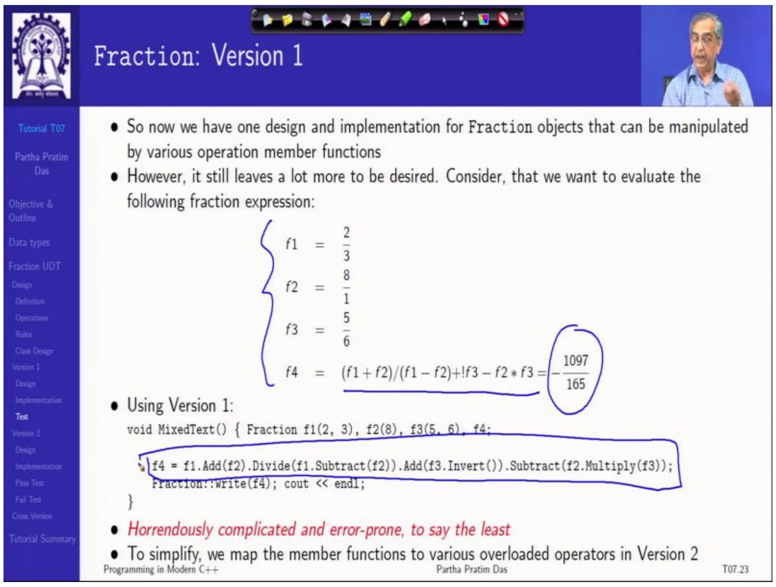
All tests passed

```
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
```

Programming in Modern C++ Partha Pratim Das T07.22

So, you just keep on doing each and every operation and write a code and generate output and see whether that is being satisfied, that is being printing correctly. So, these are all tests are written to actually be positive pass. And what you see here, you have to trace that program along with this output to be able to see that you are getting each and every output as you are expected. So, this is our version, first version of the design.

(Refer Slide Time: 27:11)



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:

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

- Using Version 1:

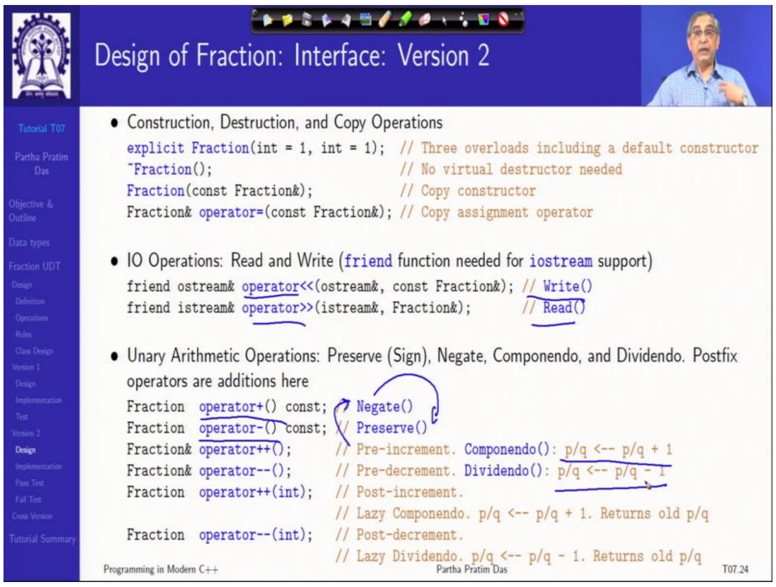
```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;

    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

Programming in Modern C++ Partha Pratim Das T07.23

And we had promised that we will refine it and use operators. So, let us see the context. Suppose with this version, I take an example of three fractions f1, f2, f3 that I have defined, and f4 is given by this expression, which actually turns out to be this value, how do you write it in this design, turns out that it is such an expression, horrendous looking, you can get lost any time in terms of using the operator name, parenthesis and so on. And it is very error prone.

(Refer Slide Time: 27:50)



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()
```

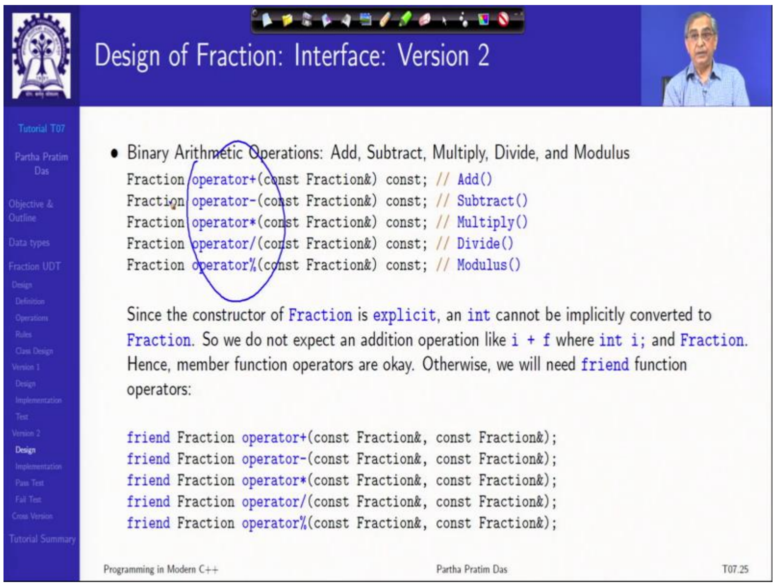
- Unary Arithmetic Operations: Preserve (Sign), Negate, Componendo, and Dividendo. Postfix operators are additions here

```
Fraction  operator+() const;  // Negate()
Fraction  operator-() const;  // Preserve()
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
```

Programming in Modern C++ Partha Pratim Das T07.24

And that is a motivation for you to jump to version 2, where the improvement that you do is for every member function that you have used, try to identify what is the appropriate operator that can be used. So, naturally for right and read, this is trivial. So, for. Sorry, this names are swapped, I will correct them. So, preserve is a positive, negate is a negative, then you have the pre-increment, which is componendo, then you have dividend. Incidentally, you also have post-increment, post-decrement. So, they are those can be called lazy componendo and lazy dividendo.

(Refer Slide Time: 28:40)



Design of Fraction: Interface: Version 2

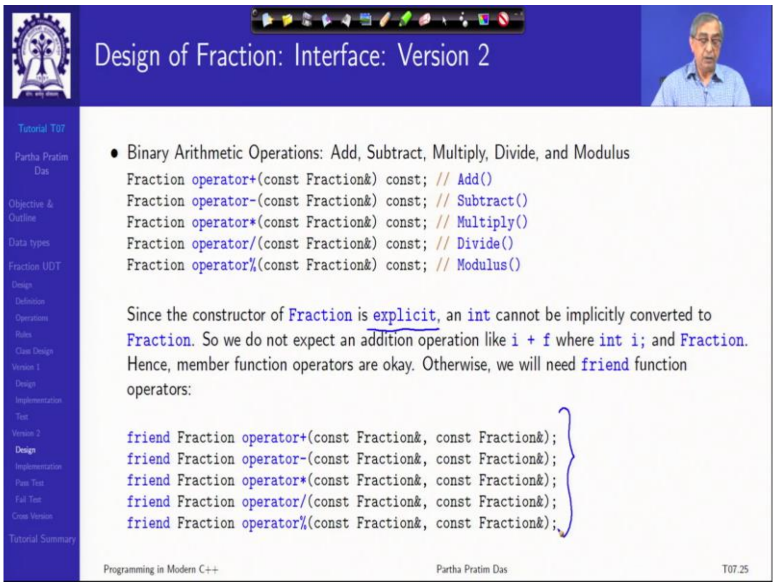
- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
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&);
```

Programming in Modern C++ Partha Pratim Das T07.25



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:

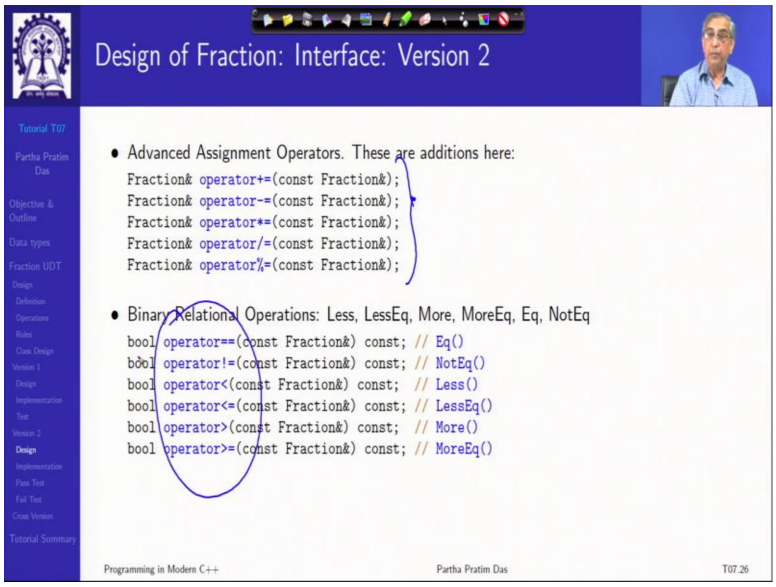
```
friend Fraction operator+(const Fraction&, const Fraction&);
friend Fraction operator-(const Fraction&, const Fraction&);
friend Fraction operator*(const Fraction&, const Fraction&);
friend Fraction operator/(const Fraction&, const Fraction&);
friend Fraction operator%(const Fraction&, const Fraction&);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

Programming in Modern C++ Partha Pratim Das T07.25

But what is more interesting is you can have all your addition, subtraction, all these all these operations, now written in terms of operators. And just to remind you on whether they should be member function or they should be friend function, just recall that the shortcoming of being a member function is that the first operand has to be a fraction type. Now, our constructor here is explicit, which means that we will not allow to take an integer and implicitly convert that to a fraction.

So, obviously the first member is a fraction can be a very good assumption and therefore, we do not provide the friend form of the operators which you may provide. There is no harm, but this is you can hear it just suffices to have member function operators overloaded for your addition, subtraction, binary operations.

(Refer Slide Time: 29:43)



Design of Fraction: Interface: Version 2

Tutorial T07

Partha Pratim Das

Objective & Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- 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&);
```

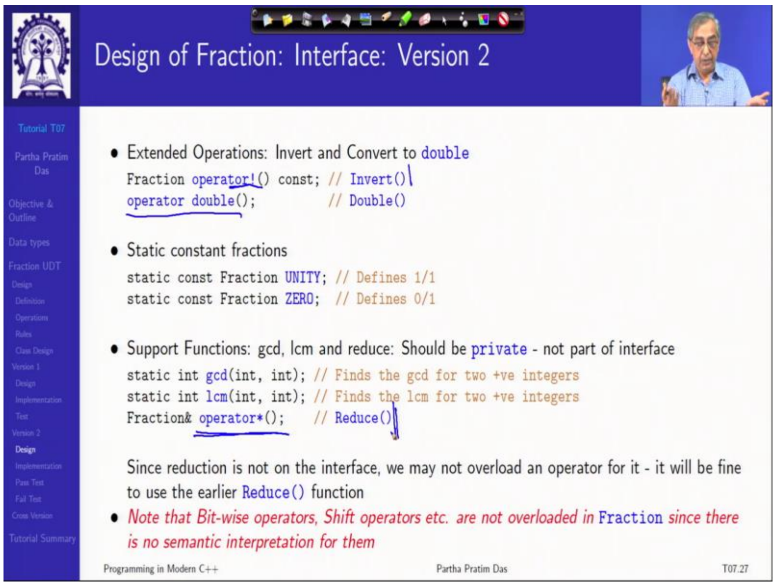
- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
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()
```

Programming in Modern C++ Partha Pratim Das T07.26

Then you do have your, you know advanced operators where you x assign x plus equal to y meaning x assigned x plus y and so on your relational operators and so on. So, all the time doing is I am just replacing the member functions that I have already designed by identifying the proper operator which can be used in this case.

(Refer Slide Time: 30:08)



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

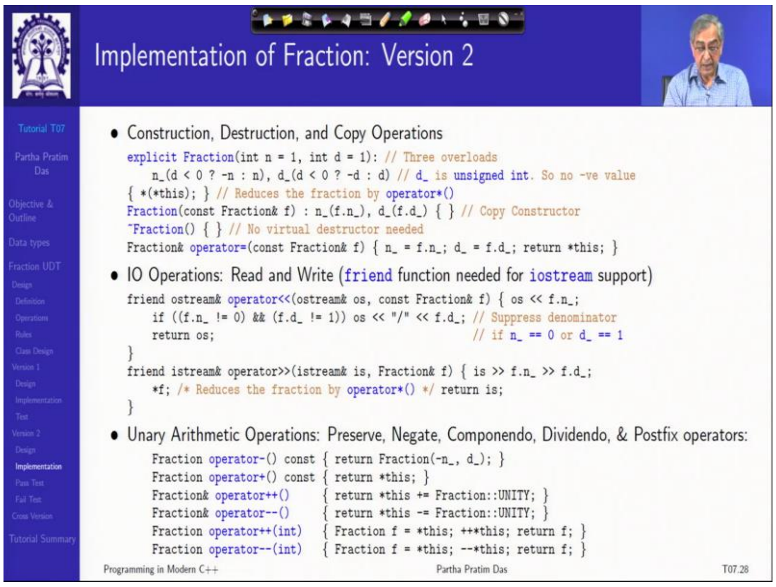
- *Note that Bit-wise operators, Shift operators etc. are not overloaded in Fraction since there is no semantic interpretation for them*

Programming in Modern C++ Partha Pratim Das T07.27

So, I have this exclamation for negate, I have operated double for the conversion and so on. And I have mapped reduce to this a little unnatural, because you do not have a notion of reducing int, but I have mapped a dereferencing operator because kind of it takes content, dereferencing operator takes content.

So, reduce is taking out the proper content so, on this but since this is not on the interface of your design, you may just retain the name reduce also because users will never use this. And there are several other operators which will not get mapped like bit-wise operators or shift operators and so on.

(Refer Slide Time: 30:53)



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; }
```

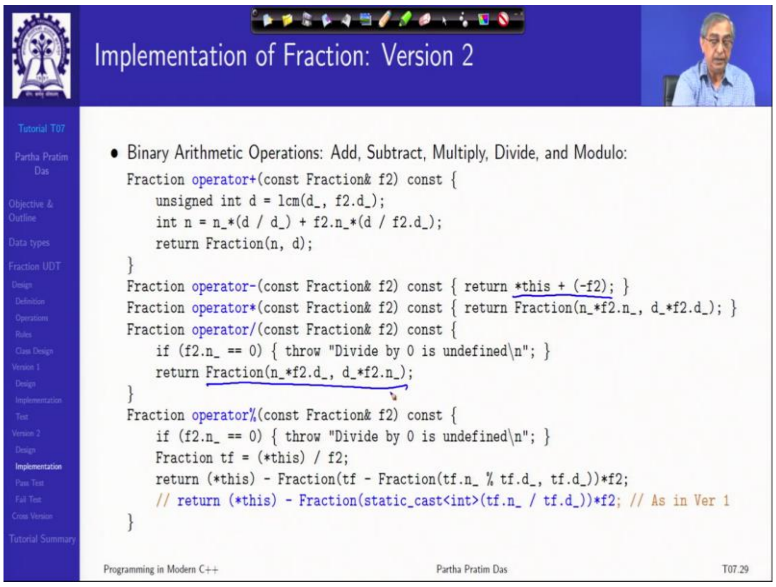
- IO Operations: Read and Write (friend function needed for iostream support)

```
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; }
```

Programming in Modern C++ Partha Pratim Das T07.28



Implementation of Fraction: Version 2

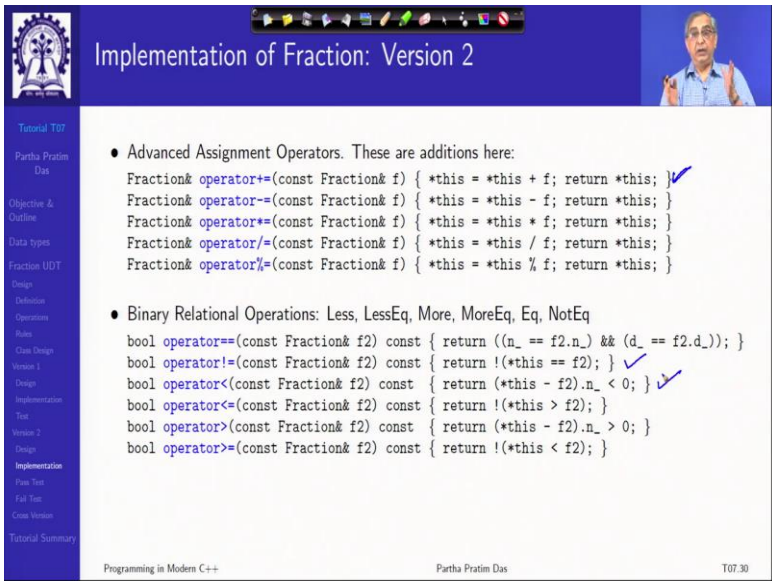
- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulo:

```
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
}
```

Programming in Modern C++ Partha Pratim Das T07.29

So, next are just replacing the implementations. And as you will have to go through this try this out thoroughly, we will see that several of them are just named replacement, but several of them particularly when it comes to say, writing this operator, operator subtraction, or writing, say, operator division and all that, you will find that you are even the expression of implementation would be much easier.

(Refer Slide Time: 31:33)



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; }
```

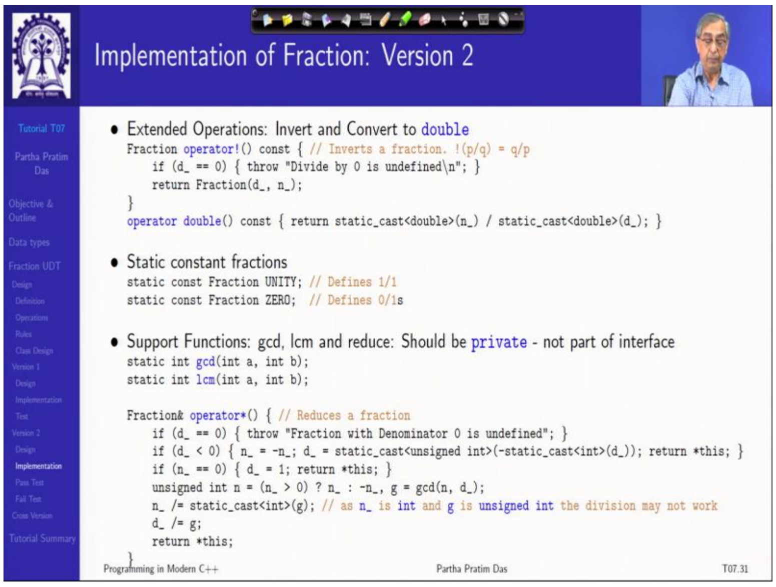
- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
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); }
```

Programming in Modern C++ Partha Pratim Das T07.30

Go through this and he will have a good sense of this. And you can see that how in single line in just two lines, you can write the advanced assignment operators, how quickly you can write the Boolean, the relational operators and so on, and everything is now in terms of operators as of your int type.

(Refer Slide Time: 31:53)



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_); }
```

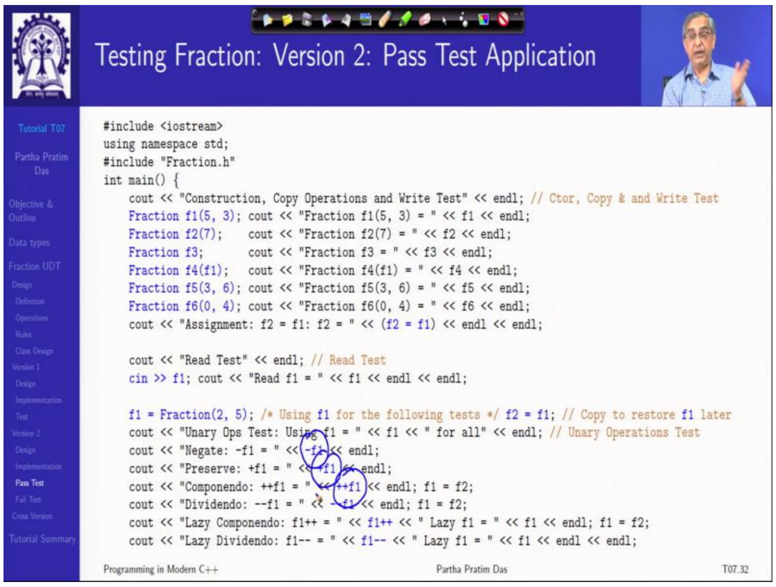
- Static constant fractions

```
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;
}
```

Programming in Modern C++ Partha Pratim Das T07.31



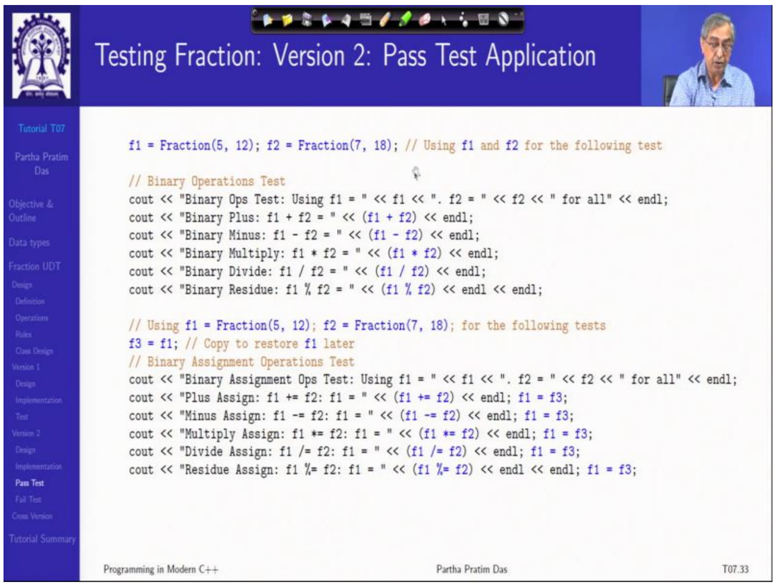
Testing Fraction: Version 2: Pass Test Application

```
#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;
```

Programming in Modern C++ Partha Pratim Das T07.32

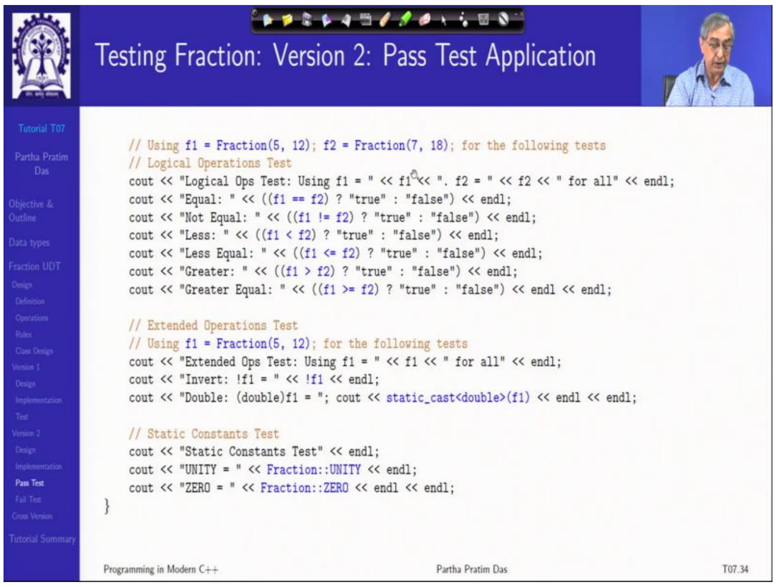
So, this is your complete implementation, go through that try this out. And again run this similar test utility to test out if things have been properly implemented. Of course, the expressions will now change for example, now, I earlier you were writing negate f1.negate, now you write -f1, you are writing say f1.componendo now you are just writing ++f1 and so on. So, things become more algebraic in nature. And you can have a complete test application based on that through which you test.

(Refer Slide Time: 32:27)



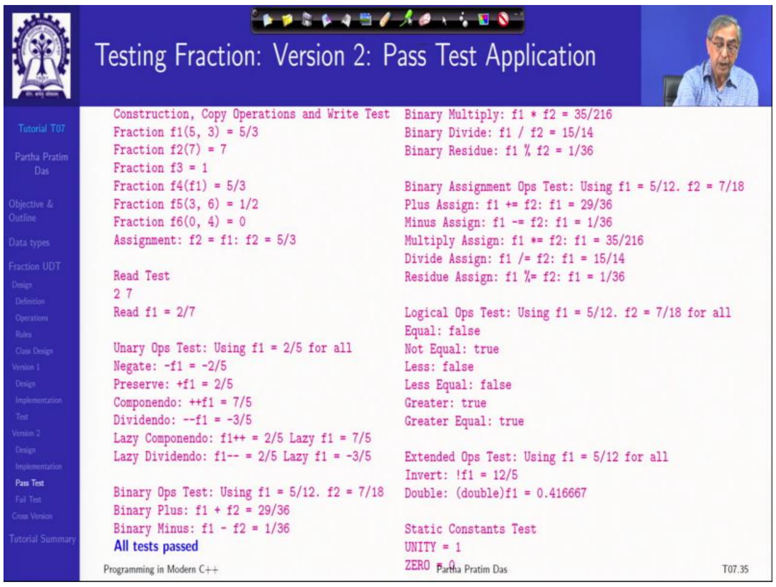
Testing Fraction: Version 2: Pass Test Application

```
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

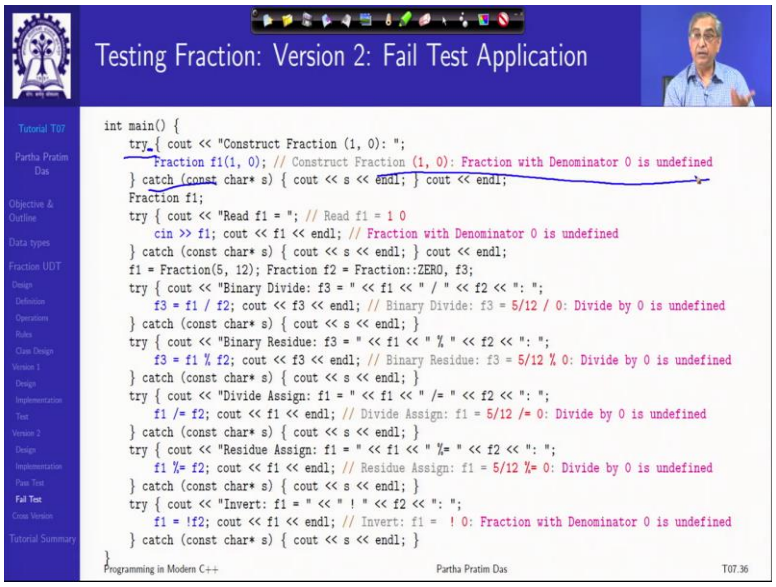
```
Construction, Copy Operations and Write Test
Fraction f1(5, 3) = 5/3
Fraction f2(7) = 7
Fraction f3 = 1
Fraction f4(f1) = 5/3
Fraction f5(3, 6) = 1/2
Fraction f6(0, 4) = 0
Assignment: f2 = f1: f2 = 5/3

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
```

And this is the total application I have given here. And as you test you will be able to get this output, try this out and you will see that very clearly.

(Refer Slide Time: 32:49)



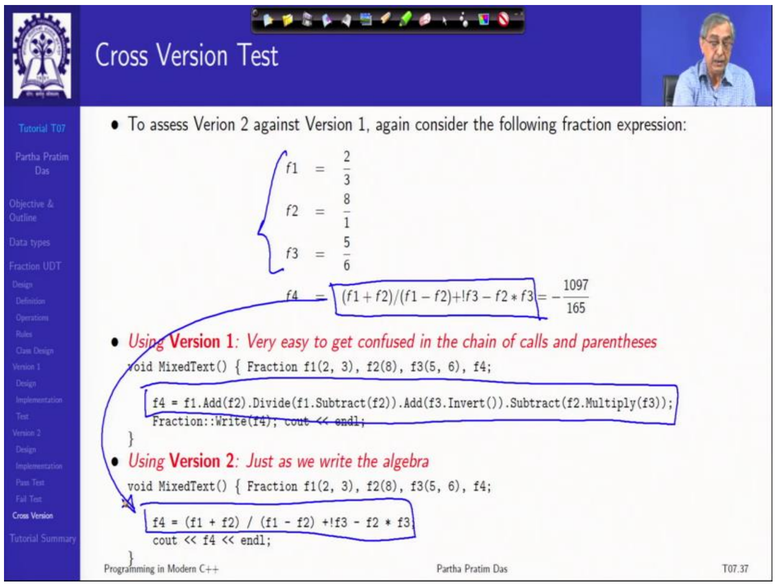
Testing Fraction: Version 2: Fail Test Application

```cpp
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; }
}
```

Programming in Modern C++ Partha Pratim Das T07.36

Now, there are one thing which you might want to also test is what happens if you have a failure? For example, what if somebody tries to construct a fraction with a denominator which is 0. So, naturally that those things will throw in your code. So, to try them you have to use the try, catch block and then see that actually, whether those messages are being got. So, here are examples of constructing with a denominator 0 or trying to divide by the 0 fraction or trying to take remainder with the 0 fraction and so on. And you will see that all of these will also pass because they will they are throwing the proper messages.

(Refer Slide Time: 33:33)



Cross Version Test

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

$$f1 = \frac{2}{3}, \quad f2 = \frac{8}{1}, \quad f3 = \frac{5}{6}$$

$$f4 = \boxed{(f1+f2)/(f1-f2)+!f3-f2*f3} = -\frac{1097}{165}$$

- **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*

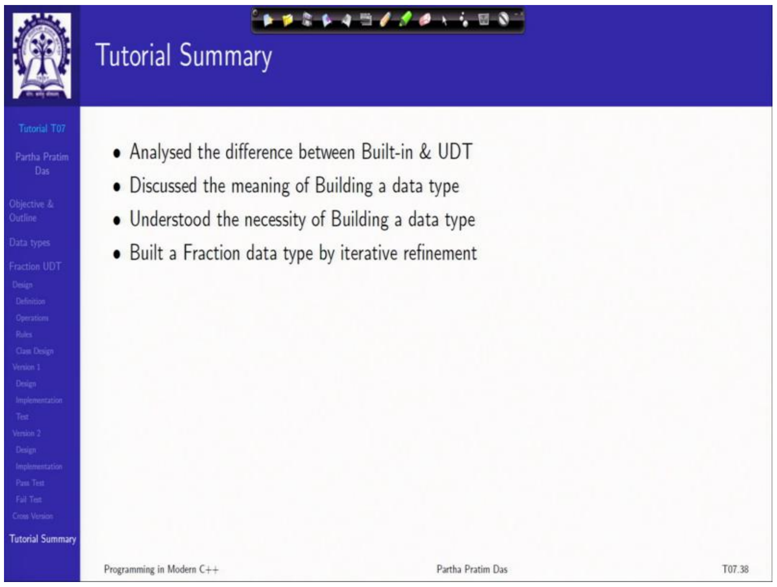
```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;
    f4 = (f1 + f2) / (f1 - f2) +!f3 - f2 * f3;
    cout << f4 << endl;
}
```

Tutorial T07 | Partha Pratim Das | Objective & Outline | Data types | Fraction UDT | Design | Definition | Operations | Rules | Class Design | Version 1 | Design | Implementation | Test | Version 2 | Design | Implementation | Pass Test | Fail Test | Cross Version | Tutorial Summary

Programming in Modern C++ Partha Pratim Das T07.37

Now, before we conclude let us get back to the example we took after version 1, that what if a high have three fractions and just an apparently not very complicated algebraic expression with these fractions and version 1 lead to such an expression, in version two, it reduces to exactly as you write the algebra that is the key advantage of being able to overload operators with proper meaning. And that is how your version 2 will turn out to be true UDT because it is now allowing you to use Fraction just like you used the int, float these kinds of types.

(Refer Slide Time: 34:25)



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

Tutorial T07 | Partha Pratim Das | Objective & Outline | Data types | Fraction UDT | Design | Definition | Operations | Rules | Class Design | Version 1 | Design | Implementation | Test | Version 2 | Design | Implementation | Pass Test | Fail Test | Cross Version | Tutorial Summary

Programming in Modern C++ Partha Pratim Das T07.38

So, in this part of the tutorial and we will continue in the next tutorial also show you with building of other UDTs, but we have discussed about the general issues in building a data type and why we need to build that and we have shown how to build a Fraction data type by iterative refinement. Thank you all very much for your attention. See you in the next module.