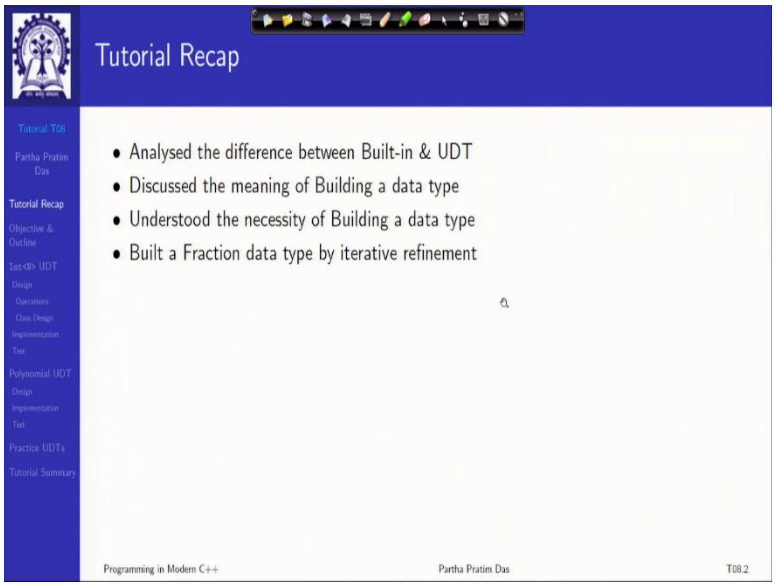
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 08
How to design a UDT like built-in types?: Part2: Int and Poly UDT

Welcome to Programming in Modern C++. We are going to discuss tutorial 08 on how to design a user defined type like the built-in type.

(Refer Slide Time: 00:43)



Tutorial Recap

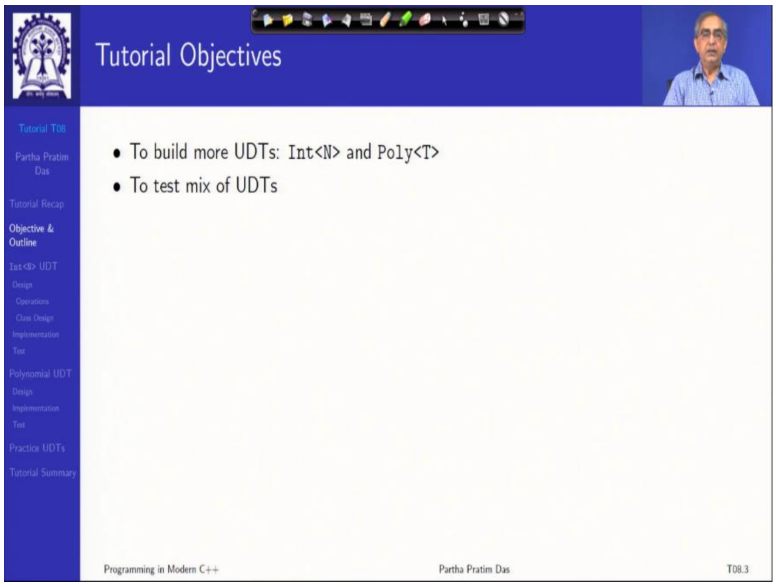
- Analysed the difference between Built-in & UDT
- Discussed the meaning of Building a data type
- Understood the necessity of Building a data type
- Built a Fraction data type by iterative refinement

Tutorial T08
Partha Pratim Das
Tutorial Recap
Objective & Outline
Int<8> UDT
Design
Operations
Class Design
Implementation
Test
Polynomial UDT
Design
Implementation
Test
Practice UDTs
Tutorial Summary

Programming in Modern C++ Partha Pratim Das T08.2

We had started discussing this in the last tutorial, in tutorial 07 where we analysed the difference between built-in types and UDTs and discussed what is the meaning of building a data type and understood the necessary build process and build a Fraction data type by iterative refinement.

(Refer Slide Time: 01:05)



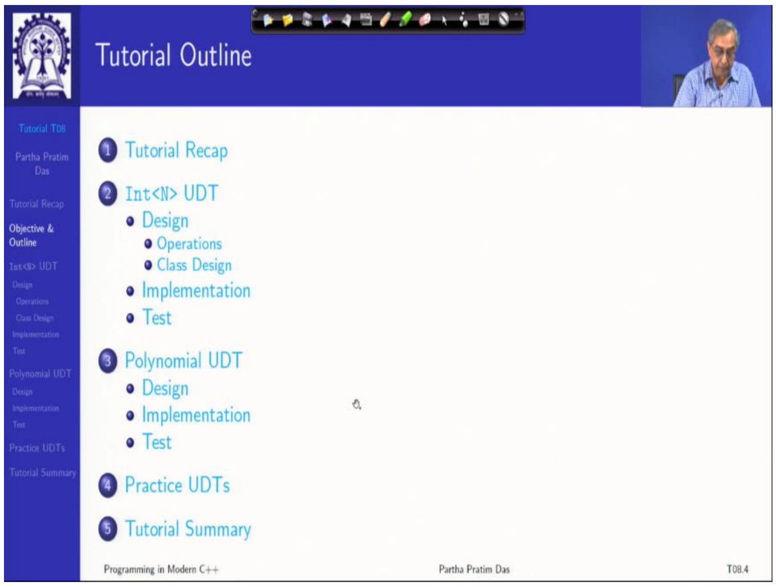
Tutorial Objectives

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

Programming in Modern C++ Partha Pratim Das T08.3

Continuing on that, we will build two more UDTs in this tutorial Int<N> and Poly<T> and we will try to now mix these UDTs that we have built.

(Refer Slide Time: 01:22)



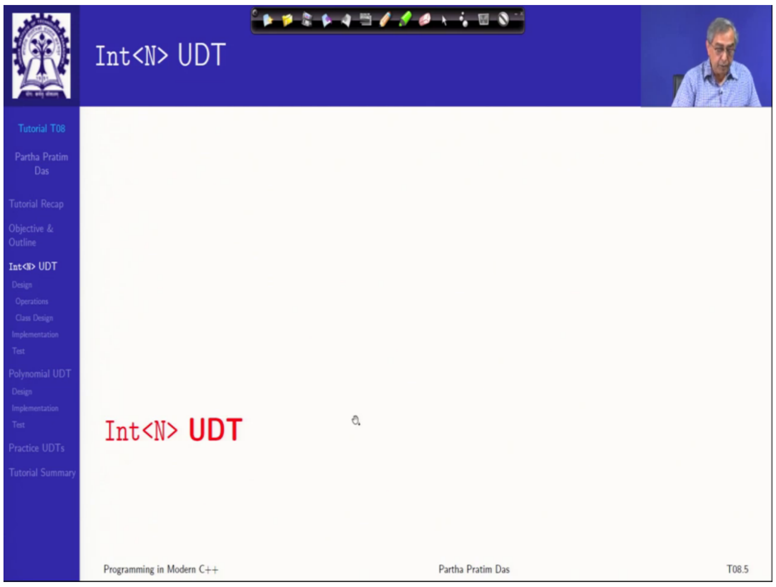
Tutorial Outline

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

Programming in Modern C++ Partha Pratim Das T08.4

So, this is the outline which will be there on the left all the time.

(Refer Slide Time: 01:28)


Int<N> UDT

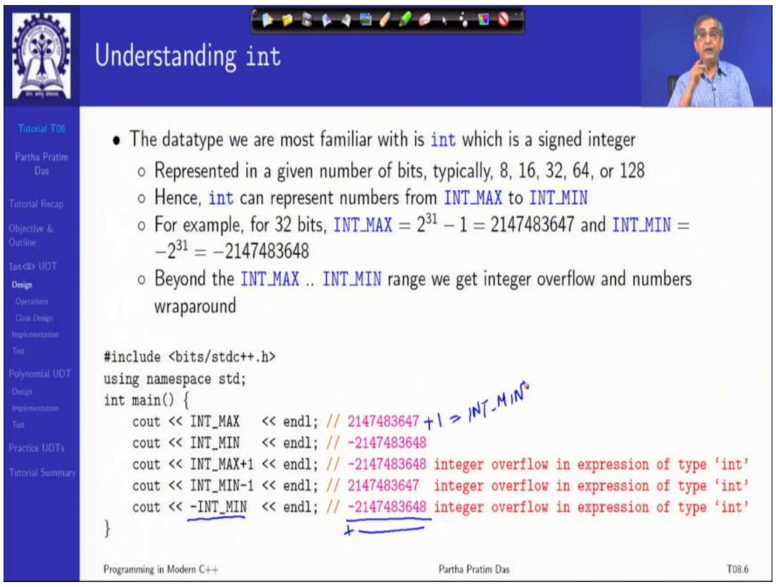
Tutorial T08

Partha Pratim Das

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

Int<N> UDT

Programming in Modern C++ Partha Pratim Das T08.5


Understanding int

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

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

Programming in Modern C++ Partha Pratim Das T08.6

Let us first talk about Int<N> UDT. Int<N> UDT has two purpose, one is to build a data type that is like the int data type in the built in int, integer data type of C++ and in the process understand some finer details regarding the int datatype itself. So, we are familiar with int as a signed integer, we know that it is represented in a given number of bits, typically 8-bit, 16-bit, 32-bit, 64 bit or even 128 bits. In our desktops and servers we mostly work with 32 and 64 bit but 8 and 16 bits are also popular in different embedded systems. 128 bit used in very, very high end applications which need really, really large integer numbers.

Now, we know that for the int type, there are two constants that are defined in the system from which you can really know what is the size of this integer and what is the range of values that the int is going to represent. For example, if you are using 32 bits, then

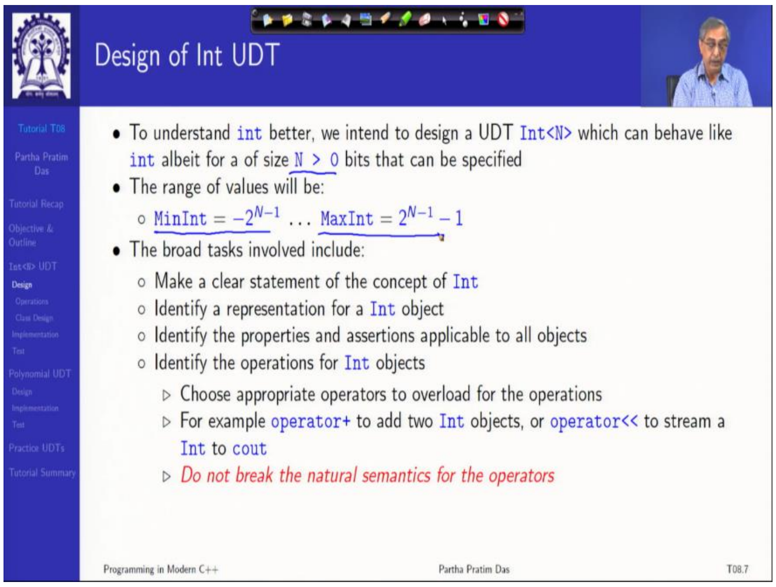
INT_MAX, the maximum integer that you can represent is naturally 2 to the power 31 minus 1 which is this number 2147483647. And int mean the minimum number or the maximum negative number that you can represent is minus 2 to the power 31 which is this number.

Naturally on the positive side, you have one number less because 0 is counted on the positive side. Beyond this INT_MAX, INT_MIN range, we get integer overflow and numbers wraparound. So, what does that mean is if you just write a very simple program to analyse this, you will see that you can print INT_MAX and INT_MIN which on a 32-bit system, I am showing this. It gives you these above values. But if you try to print INT_MAX plus 1, if you want to have one number more than that, then it actually wraps around the maximum and comes to the minimum. So, here you have INT_MAX and here you have INT_MIN mean.

So, this is 2 to the power 31 minus 1, this is -2 to the power 31. So, if you do plus 1, then instead of going here, instead of giving you 2 to the power 31, it wraps around and gives you minus 2 to the power 31. That is the basic wraparound process of finite bit representation that we have. I mean no matter how many bits you have, you will always have this property too.

Similarly, if you try to decrement subtract one from INT_MIN, the minimum negative number it will wrap around in the reverse way in the opposite way and give you the maximum positive number. And very interestingly, if you try to take the negative of minus INT_MIN you will get INT_MIN itself. You do not you will not get a positive number because the minus INT_MIN does not is has to be plus of this which does not have representation which is this plus 1 which basically as we have seen boils down to INT_MIN. So, these are the boundary values that you can see is where the numbers will behave differently from what you expect in the arithmetic to happen. I am sure you had known this, but I just wanted to highlight.

(Refer Slide Time: 05:29)



Design of Int UDT

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

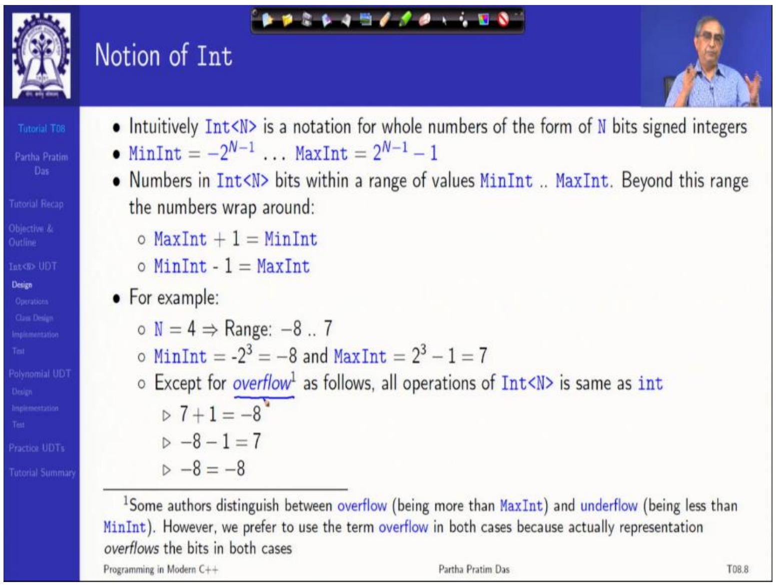
Tutorial T08 | Partha Pratim Das | Tutorial Recap | Objective & Outline | Int<N> UDT | Design | Operations | Class Design | Implementation | Test | Polynomial UDT | Design | Implementation | Test | Practice UDTs | Tutorial Summary

Programming in Modern C++ | Partha Pratim Das | T08.7

So, given this, if you want to build an integer type signed type, with a given number of bits that you want that I want this many bits and I want this integer property to happen within that bit. So, let us say that we want to build an integer having N number of bits, where N of course is a given constant. So naturally, we can you can very easily understand that MIN_INT which is the minimum integer you can represent will be -2 to the power (N–1) and MAX_INT will be 2 to the power (N–1) - 1.

So, with this understanding, we can now build a Int<N> type which behaves like the integer type except that it has the specified number of bits that you want and it need not be I mean normally, you will make it a power of 2 but it may not be a power of 2 it could be 7, it could be 35, anything that you really want it to be. So, even it could be much smaller.

(Refer Slide Time: 06:34)



Notion of Int

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

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

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

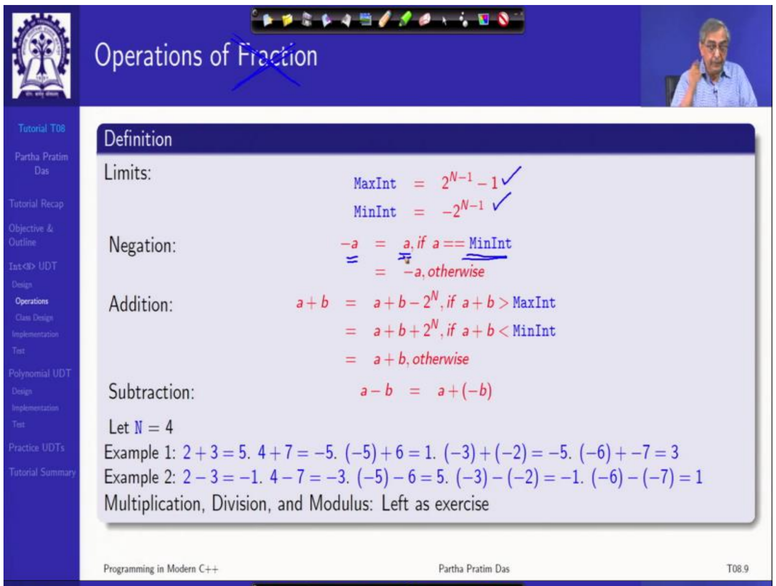
Programming in Modern C++ Partha Pratim Das T08.8

So, if we want to build Int<N> naturally, we want we would know that Int<N> will involve numbers from MIN_INT to MAX_INT. These are the wraparound properties that we have already seen. So, if we take a specific example, say if we take N to be 4 bits, I want a nibble integer, you know nibble sized integer half a bite size integer, then the range certainly would be -2 to the power 3 to 2 to the power 3 - 1 that is -8 to 7 this is your MIN_INT, this is your MAX_INT and you will have these properties that if you add 1 to 7, it will become -8.

If you subtract 1 from -8 (MIN_INT), it will become MAX_INT that is 7. And if you take the negation of the MIN_INT that is -8, it will be -8. So, these are the properties that you will have to ensure in and keep your all your computations bounded within that. Otherwise, so these are the overflow conditions you will have.

And just note on the below that in the literature, you will often find that cases like you know going beyond the MAX_INT often is referred to as overflow while getting lower than I mean attempt to get lower than MIN_INT is often called as an underflow but like many others, like the compilers, I prefer to call both of them as overflow because if the same phenomenon happened in both cases, that is the number of bits in the representation overflows, because you are going beyond that range. So, we will consistently call both of these situations as overflows.

(Refer Slide Time: 08:26)



Operations of Fraction

Tutorial T08

Partha Pratim Das

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

Definition

Limits:

$$\texttt{MaxInt} = 2^{N-1} - 1 \checkmark$$
$$\texttt{MinInt} = -2^{N-1} \checkmark$$

Negation:

$$-a = a, \text{ if } a == \texttt{MinInt}$$
$$= -a, \text{ otherwise}$$

Addition:

$$a + b = a + b - 2^N, \text{ if } a + b > \texttt{MaxInt}$$
$$= a + b + 2^N, \text{ if } a + b < \texttt{MinInt}$$
$$= a + b, \text{ otherwise}$$

Subtraction:

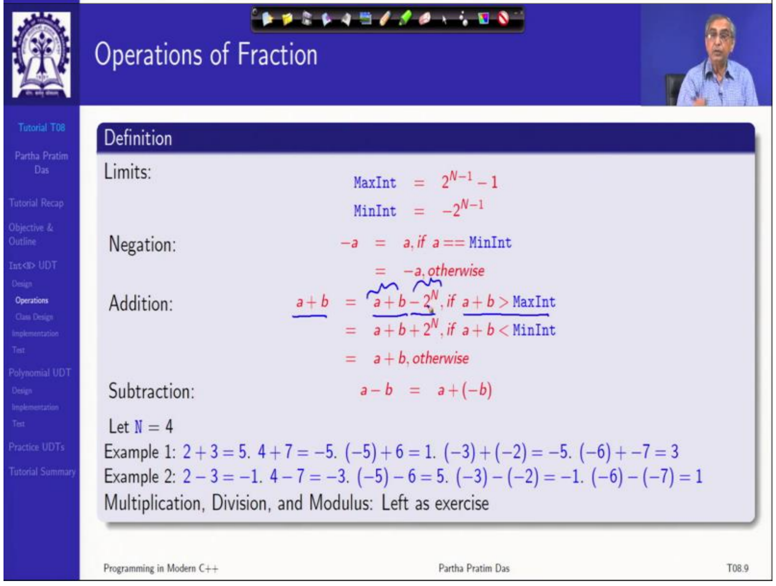
$$a - b = a + (-b)$$

Let N = 4

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

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

Multiplication, Division, and Modulus: Left as exercise



Operations of Fraction

Definition

Limits: $\texttt{MaxInt} = 2^{N-1} - 1$, $\texttt{MinInt} = -2^{N-1}$

Negation: $-a = a, \text{ if } a == \texttt{MinInt}$; $= -a, \text{ otherwise}$

Addition: $a + b = a + b - 2^N, \text{ if } a + b > \texttt{MaxInt}$; $= a + b + 2^N, \text{ if } a + b < \texttt{MinInt}$; $= a + b, \text{ otherwise}$

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

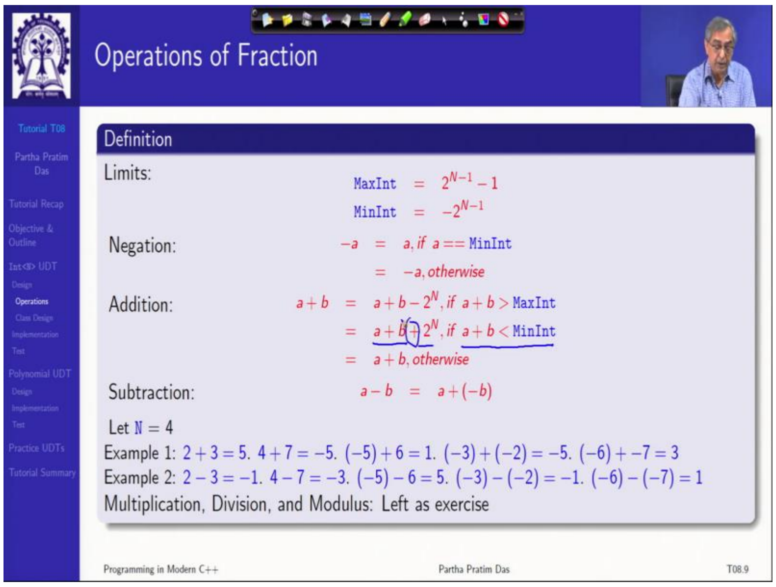
Let N = 4

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

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

Multiplication, Division, and Modulus: Left as exercise

Programming in Modern C++ Partha Pratim Das T08.9



Operations of Fraction

Definition

Limits: $\texttt{MaxInt} = 2^{N-1} - 1$, $\texttt{MinInt} = -2^{N-1}$

Negation: $-a = a, \text{ if } a == \texttt{MinInt}$; $= -a, \text{ otherwise}$

Addition: $a + b = a + b - 2^N, \text{ if } a + b > \texttt{MaxInt}$; $= a + b + 2^N, \text{ if } a + b < \texttt{MinInt}$; $= a + b, \text{ otherwise}$

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

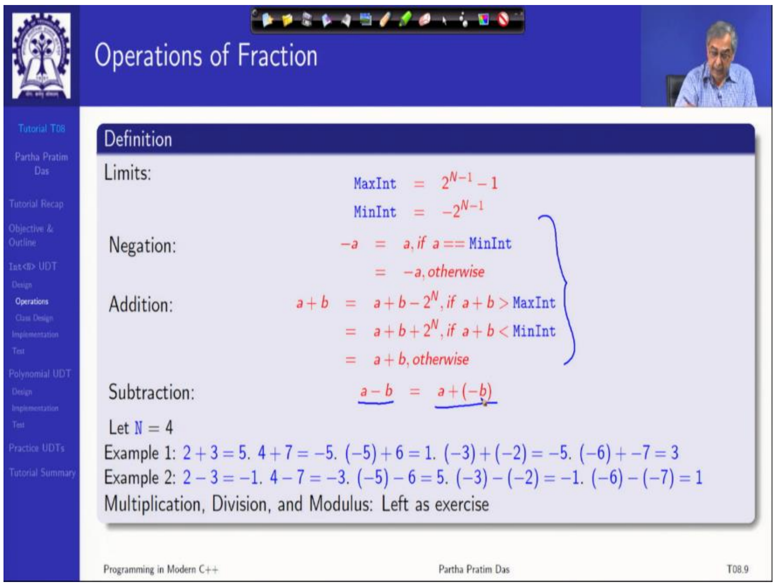
Let N = 4

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

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

Multiplication, Division, and Modulus: Left as exercise

Programming in Modern C++ Partha Pratim Das T08.9



Operations of Fraction

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Definition

Limits:

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

Negation:

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

Addition:

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

Subtraction:

$$\underline{a-b} = \underline{a+(-b)}$$

Let $\texttt{N} = 4$

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

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

Multiplication, Division, and Modulus: Left as exercise

Programming in Modern C++ Partha Pratim Das T08.9

So, in terms of this, I am sorry, this needs to be corrected, I will correct that this is operations of Int<N>. So, these are the limits. Naturally, if you do negation, then given that a is a proper Int<N> that is an N bit integer. Naturally, if it is within the range, then its negation would be the negative of that number except if it is MIN_INT. In that case, it will be the MIN_INT itself.

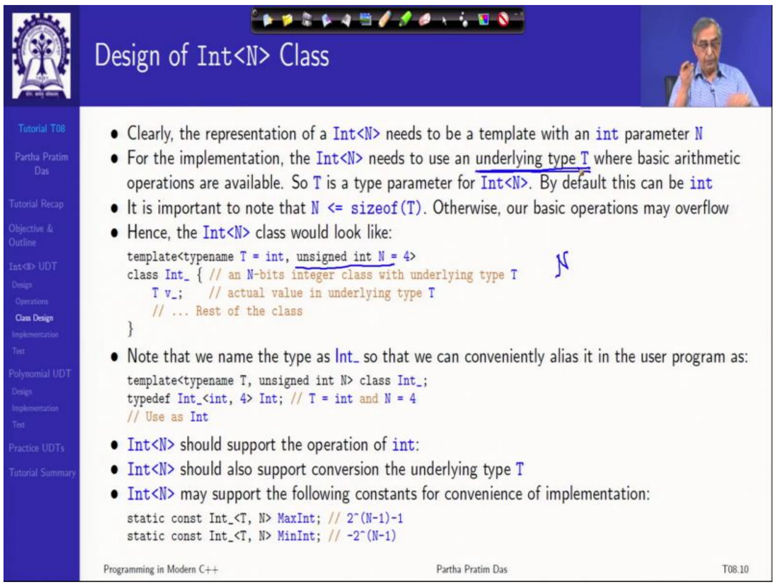
Similarly, if you want to do addition, then if the added result is more than MAX_INT you have overflown MAX_INT then your actual result will have to be obtained by adding these two and then subtracting 2 to the power N from this because you are wrapping around. So, as you wrap around, you have a jump of minus because our entire range is 2 to the power N. So, as you wrap around, you are coming down from the maximum to the minimum. So, you have a jump of -2 to the power N. So, you have to subtract -2 to the power you have to subtract 2 to the power N.

Similarly, if you add and find that you are because a and b could be negative also, so if your addition gives you a number which is less than MIN_INT in the actual algebra, then you will have to add 2 to the power N why? Because if you go below MIN_INT, you will have to wrap around and go up to MAX_INT then keep coming down. So, you will have to jump up a gap of 2 to the power N. So, you will have to add 2 to the power N. That is a simple logic and when the addition result is within the range, then you will not have to do any changes. So, this is these are the specific rules that of the operation Int<N> that you will have to follow.

Subtraction can be easily done by adding the negation of the second operant to the first operant. Here are some examples with and without overflow that you can use for following.

In the same way you can define multiplication division modulus but I leave them as exercise for you, I mean certainly not everything I should solve and present to you.

(Refer Slide Time: 10:57)



Design of Int<N> Class

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

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

N

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

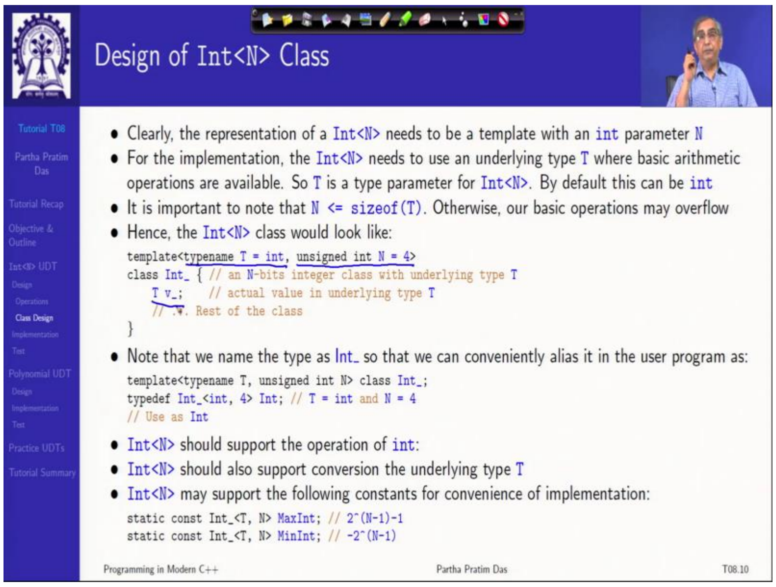
```
template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // T = int and N = 4
// Use as Int
```

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

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

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

Programming in Modern C++ Partha Pratim Das T08.10



Design of Int<N> Class

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

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

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

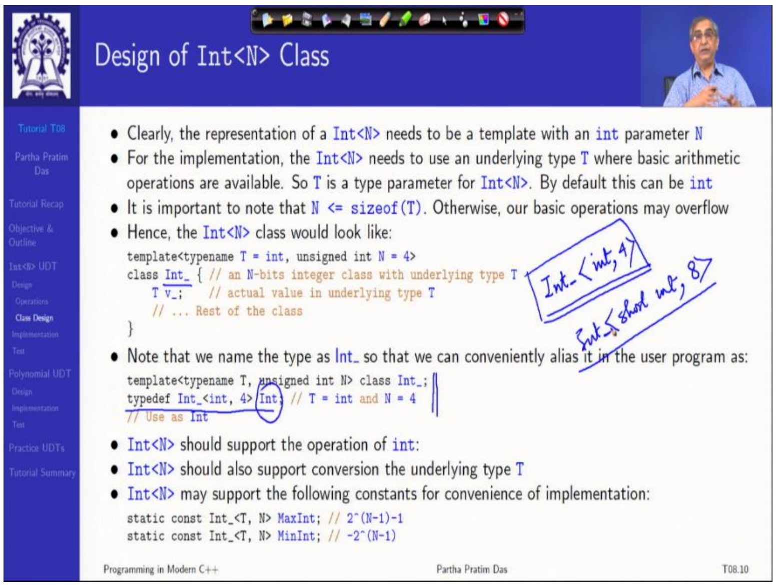
```
template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // T = int and N = 4
// Use as Int
```

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

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

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

Programming in Modern C++ Partha Pratim Das T08.10


Design of Int<N> Class

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

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

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

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

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

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

Int_<int, 4>

Int_<short int, 8>

Programming in Modern C++ Partha Pratim Das T08.10

So, in terms of the Int<N> class, what do we get? We need a value of N which is fixed for the datatype. So, N typically is a compile time constant. If N is a compile time constant, then the way I can specify N is possibly as an integer parameter for a template. So, that when I instantiate Int for the first time, the N is specified and every Int in my system because Int cannot keep on changing in my system because then it will be inconsistent.

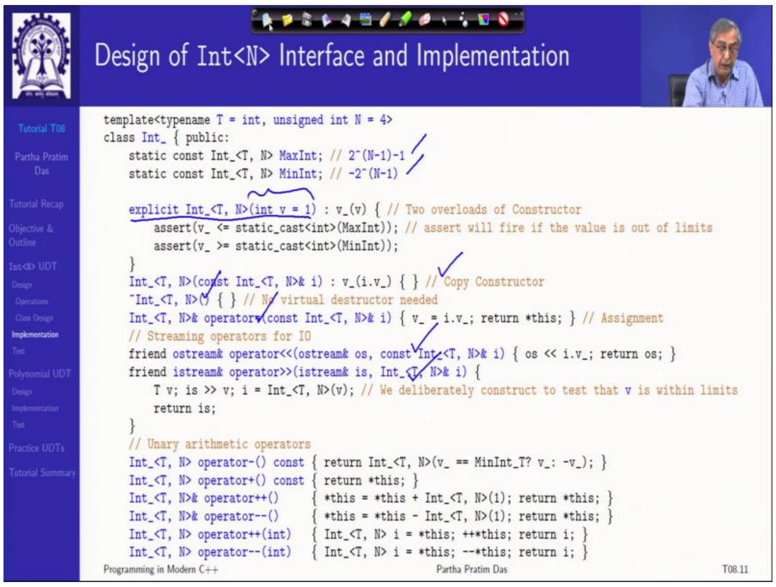
So, I will have a template integer parameter that is non-type parameter unsigned N. And I have just given a default value 4. Also, so this is for the interface, this is what the user will need but to be able to implement this, I need an addition operation or subtraction operation, multiplication operation of some pre-defined types. I cannot keep on I do not want to start writing from the bit level operation that will be very inefficient also. And typically, by N, what I mean is a value which is typically less than what is implemented available in the system by default.

So, I want to make use of some underlying type a type in which the actual computation will be done and then the wraparound will be applied on to that. So, I talk about an underlying type T which I want to use, I could use for example, I could use a signed char, I could use short int, I could use int, I could use long int and so on so forth.

So, as it turns out, that I therefore have a template where there is one type T which by default I have kept as an int. I have a non-type parameter N which by default I have kept 4 and naturally, to represent the value, I have a data member v_ have type T. So, this is basically my type basically my representation and since it is templatized to be able to use this, mind you, I am using a name, not Int, Int_ because actually to be able to write this, I have to write this something like Int_<int, 4> like that. That is actually the name of my type. And it is going to be very inconvenient to use that in every time.

So, what I want to do is I want to typedef that to the nice name Int. So, I want to keep that name Int for the user who can typedef it to this or whatever, if the underlying type is short is intended to be short and the number of bits is 8 then the user could write it like this and typedef it to Int so that Int can be used. So, in addition, we will also define two static constants, the MAX_INT and MIN_INT because I will need them for the implementation.

(Refer Slide Time: 14:36)



Design of Int<N> Interface and Implementation

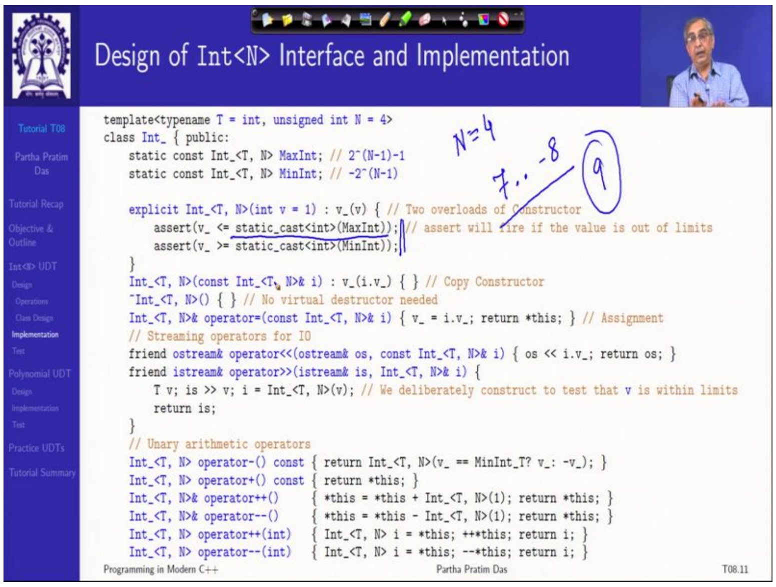
Tutorial T08

Partha Pratim Das

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

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

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

Programming in Modern C++ Partha Pratim Das T08.11



Design of Int<N> Interface and Implementation

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

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

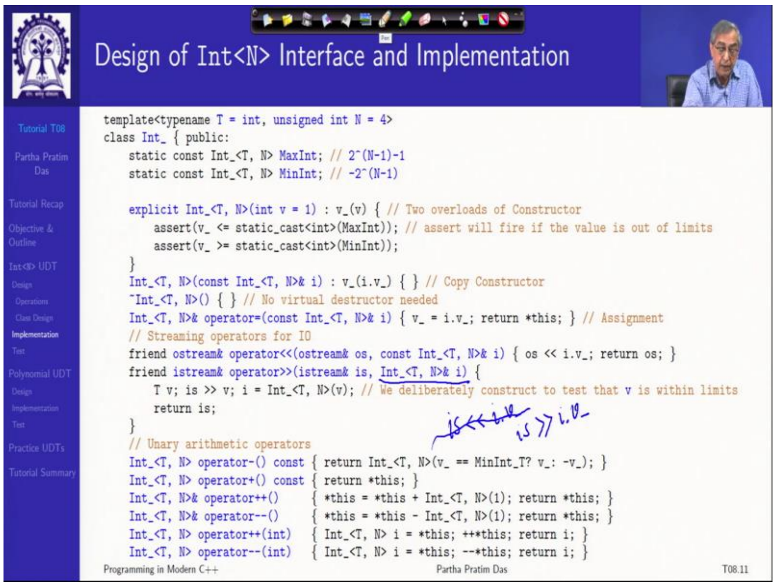
Programming in Modern C++ Partha Pratim Das T08.11

So, with that we can very easily implement this entire interface design and implement interface together. I am not presenting them separately. So, I have MAX_INT, MIN_INT. I have an explicit constructor. I prefer explicit constructor because when the conversion is required, I can explicitly do that. Unless there is any reason, I will use that. So, there is a default that is available which will give me unity number otherwise, I have to give the specified number. I have copy constructor destructor which does nothing copy assignment operator, output streaming operator, input streaming operator and so on.

The two points to note here in the constructor, I have put two assert statements. Asserts are run time statements, so that if I give a Boolean clause within the assert, then if the clause fails, then the program immediately gives a message and terminates. Since I have to be within this limit and so, if N is 4, then my range is from 7 to -8. So, if somebody wants to build an integer with the value default Int value 9, then certainly this is this will not satisfy my system. So, I would like to give an assert in doing that.

So, my assert says that it has to be less than MAX_INT and it has to be more than MIN_INT otherwise, the system will fail. I do not want to use a I do not want to throw an exception here because it is not at all a good practice to throw an exception from a constructor because then you do not know really what is happening with that constructed object because this object is ill constructed.

(Refer Slide Time: 16:35)



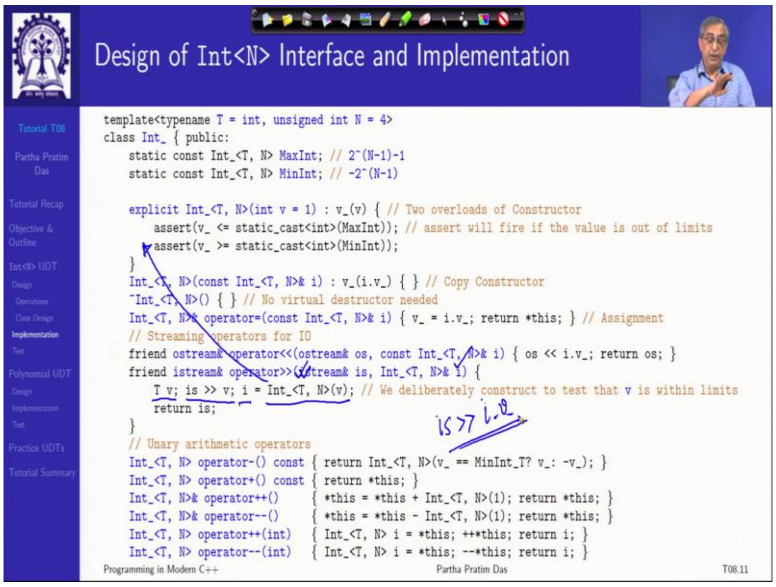
Design of Int<N> Interface and Implementation

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

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

~~is << i.v_~~ is >> i.v_

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

Programming in Modern C++ | Partha Pratim Das | T08.11



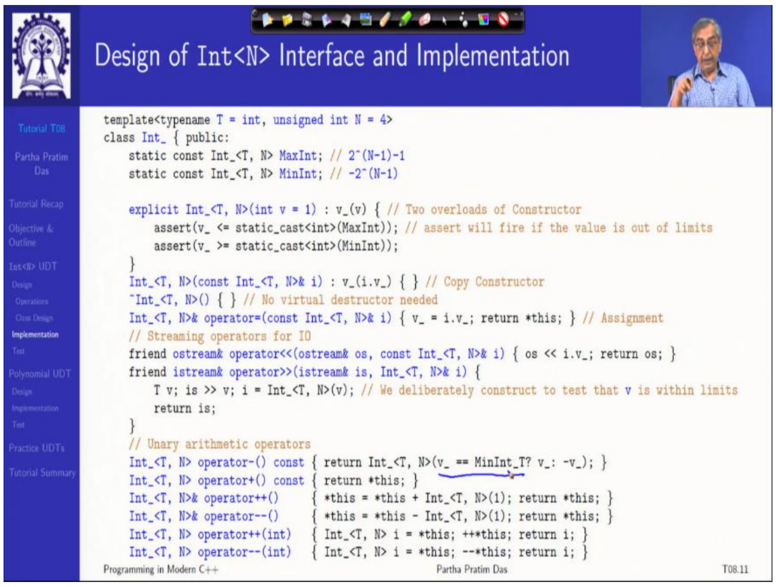
Design of Int<N> Interface and Implementation

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

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

is >> i.v_

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

Programming in Modern C++ | Partha Pratim Das | T08.11



Design of Int<N> Interface and Implementation

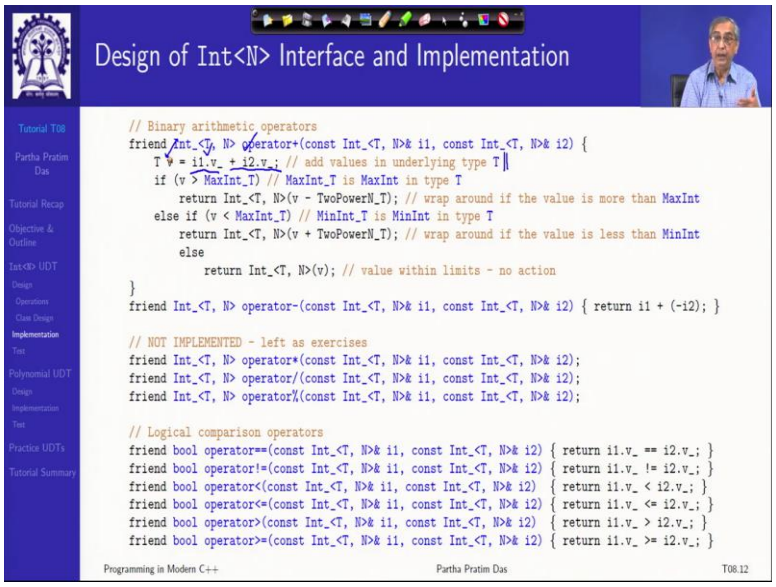
```
template<typename T = int, unsigned int N = 4>
class Int_ { public:
    static const Int_<T, N> MaxInt; // 2^(N-1)-1
    static const Int_<T, N> MinInt; // -2^(N-1)

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

Programming in Modern C++ Partha Pratim Das T08.11

In a similar manner, if you look into the input streaming operator, then in the input streaming operator I am passing an already constructed into object i. So, what could have been simply this operation is the input stream is reads i.v_. I could have simply written that but I am not choosing to do that. Just see what I am doing. I have declared a temporary variable v and I have read the input value in v and I have tried to construct an Int object and then that constructed object I have assigned to the given parameter i.

The reason for that is this construction process will naturally lead to the assertions being checked. Otherwise, by this input, if I just follow is taken reading into i.v_, then it will be possible to construct an object which will violate my condition. So, these are the nuances of, you know, datatype design that we will have to remember. So, rest of these, these are all unary operators, the only thing to remember is for negation, if v_ is MIN_INT if the value is already MIN_INT, then it remains MIN_INT otherwise it is minus of the the rest of the operators are same as what we did before.

(Refer Slide Time: 18:10)



Design of Int<N> Interface and Implementation

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

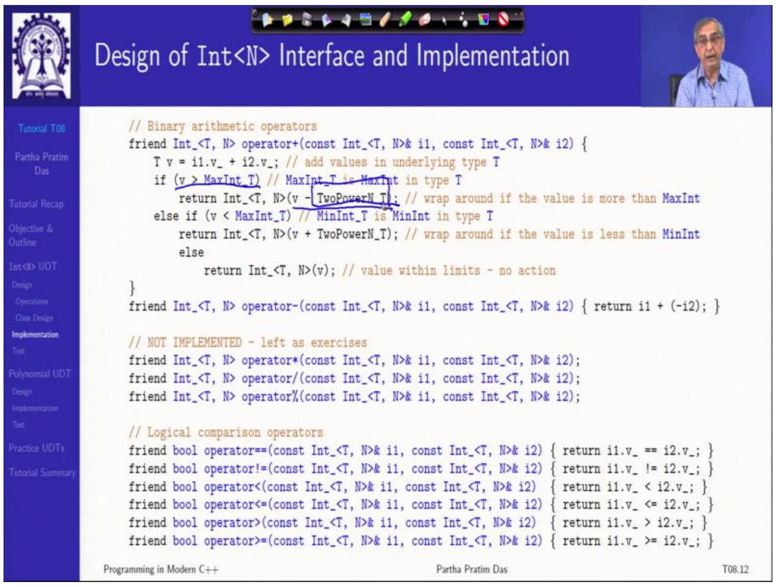
Test

Practice UDTs

Tutorial Summary

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

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

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

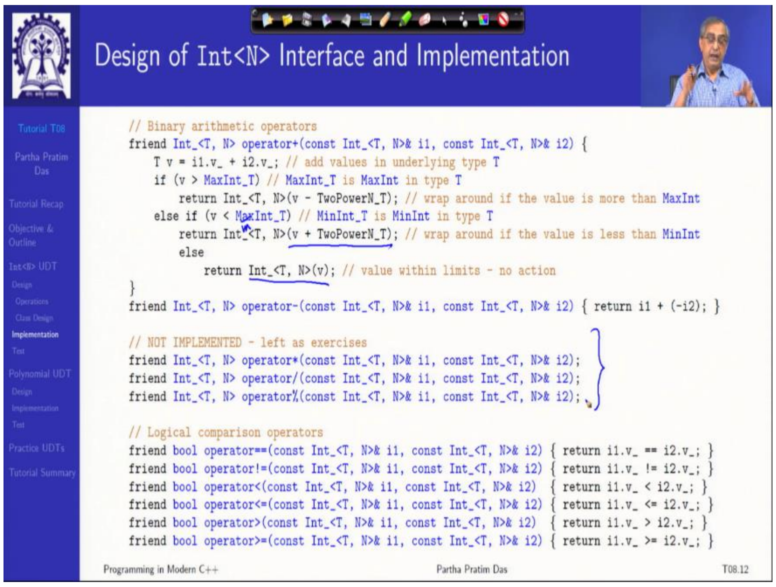
Programming in Modern C++ Partha Pratim Das T08.12



Design of Int<N> Interface and Implementation

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

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

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

Programming in Modern C++ Partha Pratim Das T08.12



Design of Int<N> Interface and Implementation

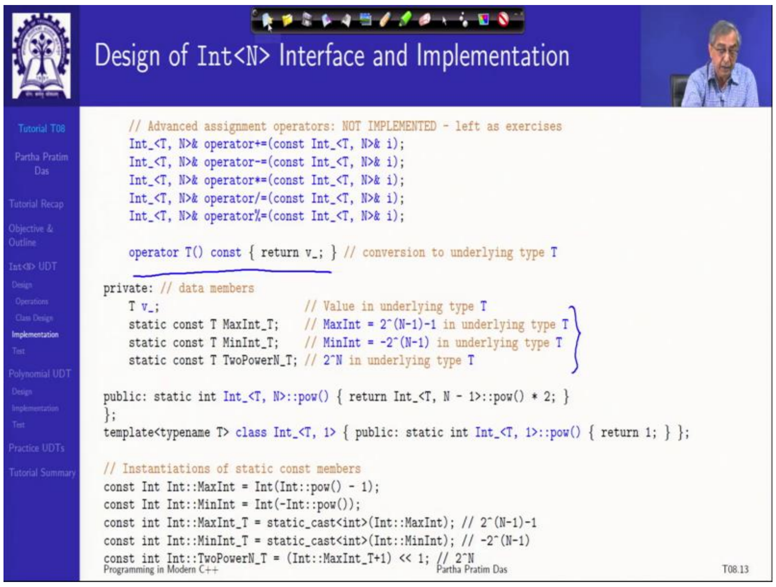
```
// Binary arithmetic operators
friend Int_<T, N> operator+(const Int_<T, N>& i1, const Int_<T, N>& i2) {
    T v = i1.v_ + i2.v_; // add values in underlying type T
    if (v > MaxInt_T) // MaxInt_T is MaxInt in type T
        return Int_<T, N>(v - TwoPowerN_T); // wrap around if the value is more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

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

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

Programming in Modern C++ Partha Pratim Das T08.12

Coming to the addition operator, we will certainly have to do a few steps to make sure that the wraparound happens. So, we first take the values from the underlying type add them this is where the underlying type is important. So, these are values of type T and this is my temporary result and then I see whether I can construct an object of value v_ using my type whether it exceeds MAX_INT or is less than MIN_INT. So, if it is greater than MAX_INT, I have to subtract 2 to the power N. So, I have kept another constant pre-computed which is 2 to the power N, a combined constant I have computed so I can use it every time.

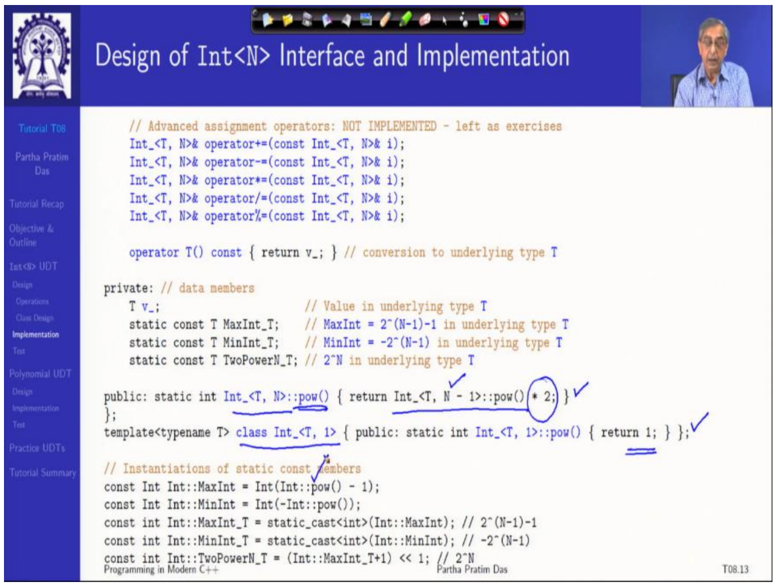
Similarly, if it is less than MIN_INT then I have to add 2 to the power of N. Otherwise, it is just that number. So, this is all that is required for the wraparound. I have left the implementation of these operators on to you. You have to think through and using this wraparound logic, you have to implement them and test them.

(Refer Slide Time: 19:32)



Design of Int<N> Interface and Implementation

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

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

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

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

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



Design of Int<N> Interface and Implementation

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

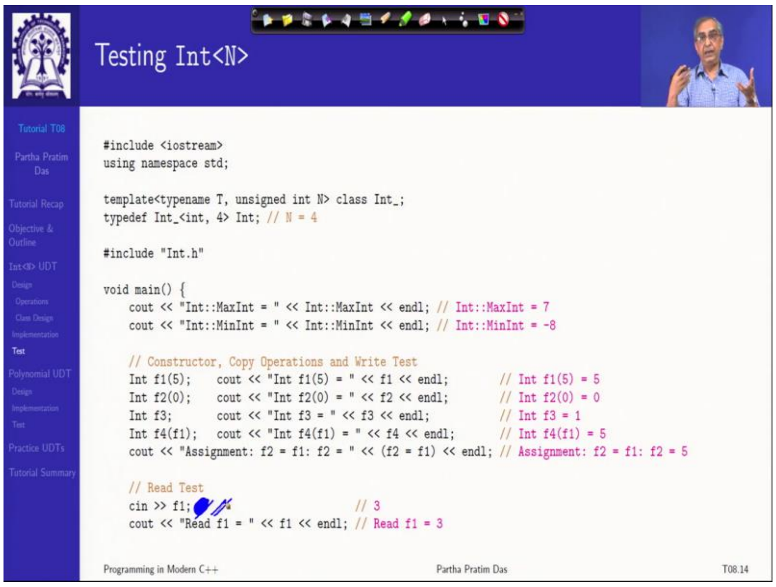
```
    // Advanced assignment operators: NOT IMPLEMENTED - left as exercises
    Int_<T, N>& operator+=(const Int_<T, N>& i);
    Int_<T, N>& operator-=(const Int_<T, N>& i);
    Int_<T, N>& operator*=(const Int_<T, N>& i);
    Int_<T, N>& operator/=(const Int_<T, N>& i);
    Int_<T, N>& operator%=(const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

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

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

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

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

Programming in Modern C++ Partha Pratim Das T08.13

Similarly, we can have the advanced assignment operators which given the operators are straightforward, I have left as exercise. I have a conversion operator to the basic underlying type which will be very convenient. And then I have the different constants and these constants I have kept in the underlying type also. So, there is a MAX_INT which is my object and there is the underlying value of MAX_INT in the underlying type that is, if my underlying type is int, your value is in that because actually, I will have to do the implementation computations in terms of that.

Here, what I need, I will need the computation of 2 to the power N. So, there is a nice illustration of a function template here where a function pow is being used. So, for function pow(), it basically returns T, N–1, pow() times 2. So, it keeps on doing that till N becomes 1 in which case it returns 1. So, a call to this function given a fixed N will give you 2 to the power N-1. So, that is something which you can very easily do and this is a very an all these are being done in the templates because these are to be computed only once. And so, I want that efficiency as well as once this is compiled, all these values constants computed rest of it can run very efficiently.

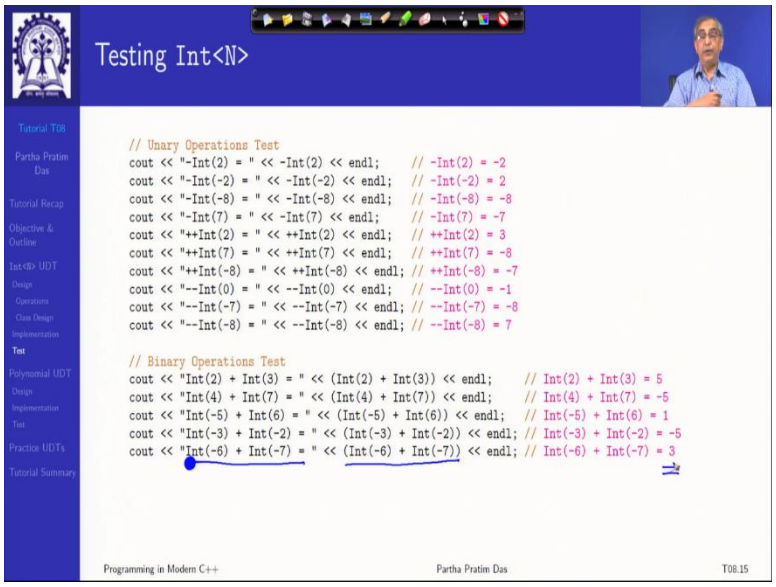
(Refer Slide Time: 21:14)



Testing Int<N>

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

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

#include "Int.h"

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

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

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

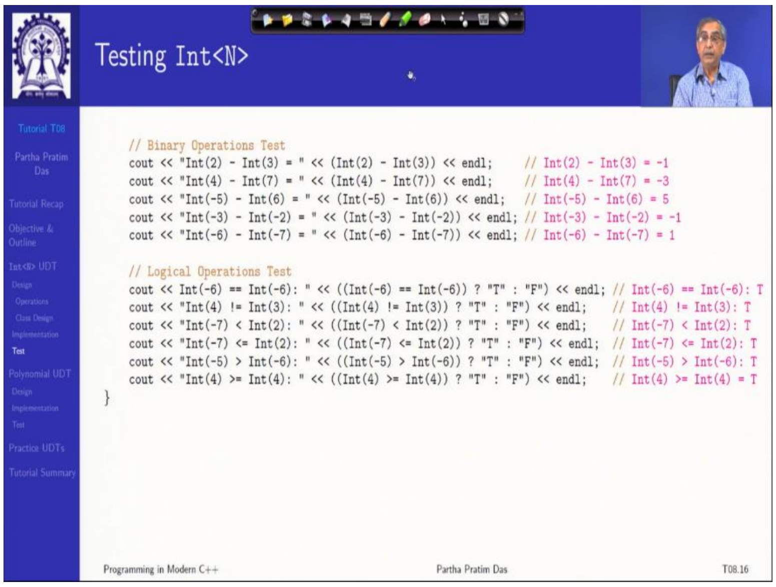
Programming in Modern C++ Partha Pratim Das T08.14



Testing Int<N>

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

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

Programming in Modern C++ Partha Pratim Das T08.15



Testing Int<N>

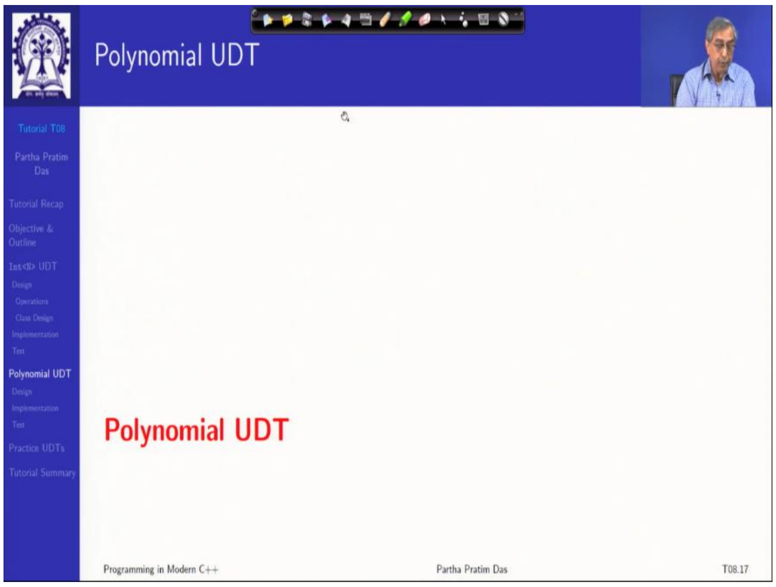
```
    // Binary Operations Test
    cout << "Int(2) - Int(3) = " << (Int(2) - Int(3)) << endl;     // Int(2) - Int(3) = -1
    cout << "Int(4) - Int(7) = " << (Int(4) - Int(7)) << endl;     // Int(4) - Int(7) = -3
    cout << "Int(-5) - Int(6) = " << (Int(-5) - Int(6)) << endl;   // Int(-5) - Int(6)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

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

Programming in Modern C++ Partha Pratim Das T08.16

So, with that now I have a test code which is very similar to the code we wrote for a fraction. So, I will not go through them each and every line. So, we have we are testing different types of constructors, we are testing for reading, you can here in terms of reading here, you can give some large value beyond the MAX_INT or smaller than the MIN_INT and see that actually you are getting the assertions. Then we have different kinds of unary operation test, binary operation test and so on. There are several places where wraparound keeps on happening.

For example, if we see here, here is one addition which is Int(-6) being added to Int(-7). So, Int(-6) added to Int(-7) will actually be in terms of underlying type it is -13. So, since it is a 4 bits, it cannot fit in there because it is less than -8. Therefore, I have to add 16 to that. So, as I add 16 to that, my actual value wrapped around value is 3. So, all these are tested out. You can check the other minus operation, logical operations and so on.

(Refer Slide Time: 22:34)



Polynomial UDT

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<8> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

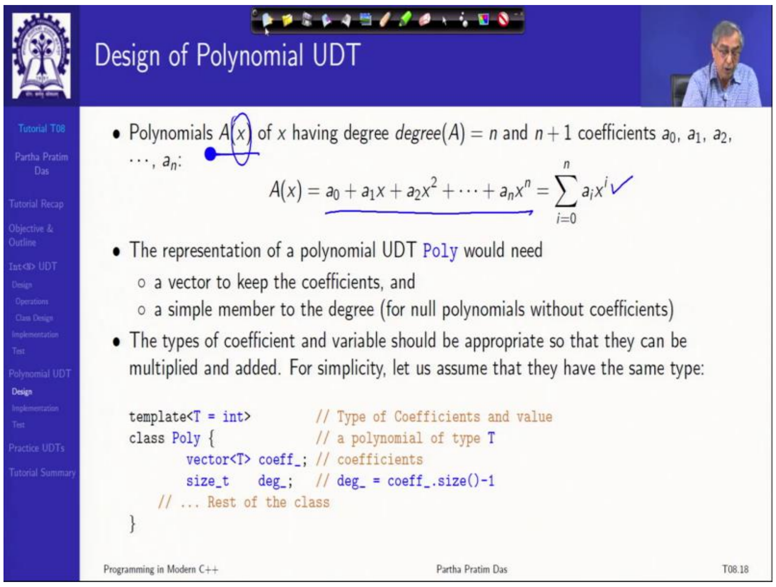
Tutorial Summary

Polynomial UDT

Programming in Modern C++ Partha Pratim Das T08.17

So, with that I have a nice integer type of N but it is great fun and, you know that exactly gives us a picture of how the Int type works in the system.

(Refer Slide Time: 22:46)



Design of Polynomial UDT

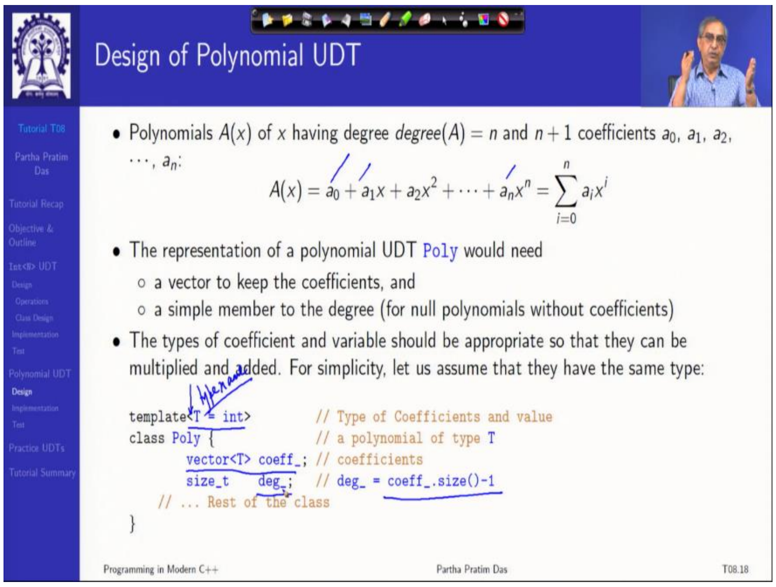
- Polynomials $A(x)$ of x having degree $degree(A) = n$ and $n+1$ coefficients $a_0, a_1, a_2, \cdots, a_n$:

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

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

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

Programming in Modern C++ Partha Pratim Das T08.18



Design of Polynomial UDT

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

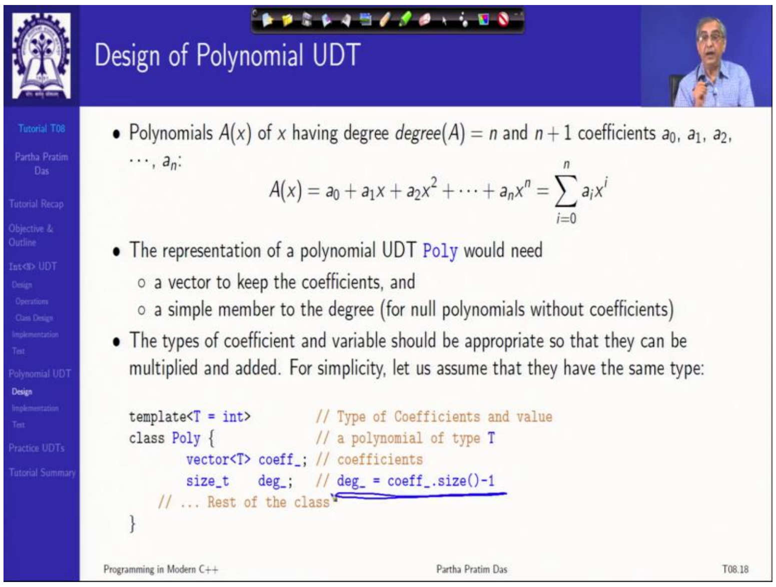
$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \sum_{i=0}^{n} a_ix^i$$

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

typename

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

Programming in Modern C++ Partha Pratim Das T08.18



Design of Polynomial UDT

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

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

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

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

Programming in Modern C++ Partha Pratim Das T08.18

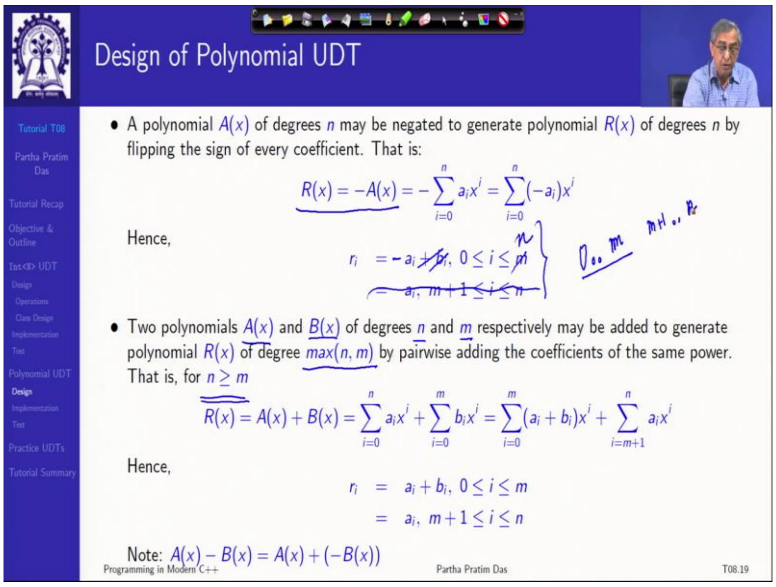
Going forward, let us do something very different. Let us we often deal with polynomials, right? So, let us try to built-in a user defined type of a polynomial. So, what is a polynomial? A polynomial is an expression is a polynomial expression like this which has degree n, n+1 coefficients a zero to an and every coefficient is multiplied by different powers of the variable of the polynomial, the independent variable of the polynomial which is x.

So, this is the polynomial given a value of x, I can evaluate the polynomial. So, if I want to represent this polynomial, then what are the things I need? I need to represent these coefficients. So, it is kind of an array and, you know, we know that in C++, we should actually be using vector. So, I will have a vector of polynomials. Now, a vector of coefficients.

Now naturally, every coefficient needs to have a type which I would like to make as generic as possible. So, I make that type as a type parameter in the template. There is a small typo here, there should be the word typename. I will correct this in the final slide. Now, along with that now if we have that then we can actually do not need if we are using a vector, we do not need the information of degree. Why?

Because in terms of this vector, I always know the size of the vector. So, the size of the vector is a number of coefficients. So, the degree has to be 1 less than that but we will not make that assumption and we will have a degree data member - degree also separately in the representation. The reason for that is having that will make sure that I can even have a null polynomial of any degree whatsoever. So, it is not necessary that I need to know the degree from the vector but I can have degree and then build the vector and that is always an invariant, this is always an invariant that must be satisfied in my polynomial type.

(Refer Slide Time: 25:11)



Design of Polynomial UDT

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

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

Hence,

$$\left.\begin{array}{rcl} r_i & = -a_i \cancel{+ b_i}, & 0 \le i \le \overset{n}{\cancel{m}} \\ & \cancel{= a_i,} & \cancel{m+1 \le i \le n} \end{array}\right\} \quad \underline{0 .. m} \quad m+1 .. n$$

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

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

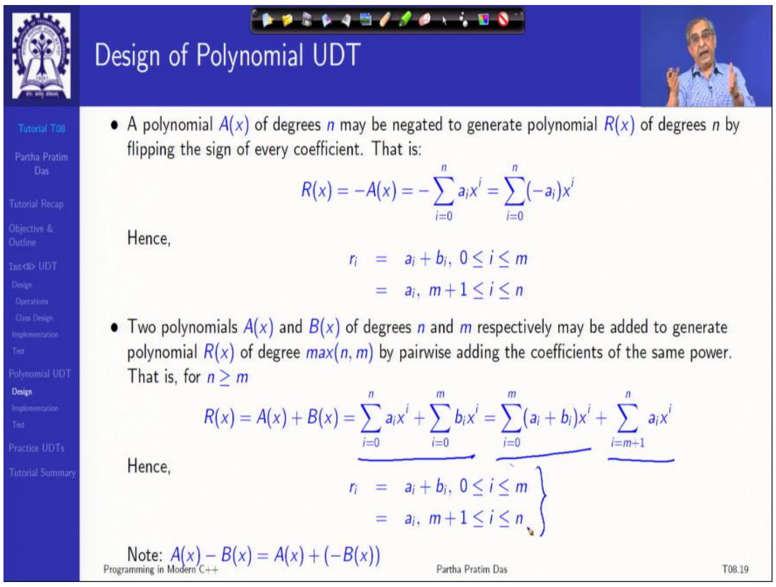
Hence,

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

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

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

Programming in Modern C++ · Partha Pratim Das · T08.19



Design of Polynomial UDT

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

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

Hence,

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

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

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

Hence,

$$\left.\begin{array}{rcl} r_i & = & a_i + b_i, \ 0 \le i \le m \\ & = & a_i, \ m+1 \le i \le n \end{array}\right\}$$

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

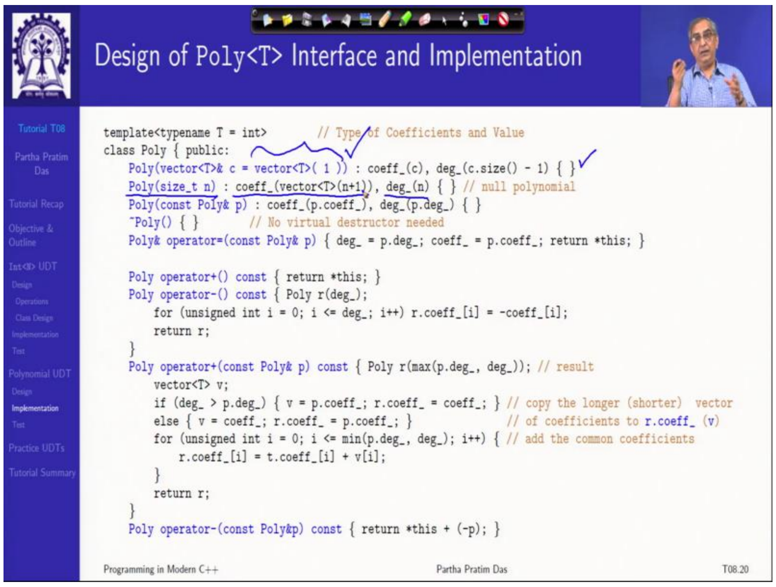
Tutorial T08 · Partha Pratim Das · Tutorial Recap · Objective & Outline · Int<k> UDT · Design · Operations · Class Design · Implementation · Test · Polynomial UDT · Design · Implementation · Test · Practice UDTs · Tutorial Summary

Programming in Modern C++ · Partha Pratim Das · T08.19

Now, what are the basic operations? Now, naturally, again, you can do several operations with polynomial. The simplest one is I can negate a polynomial, that is, negating every coefficient of that. So, that is what is defined here, a polynomial -A(x) is basically when you negate the polynomial. So, this basically, this is a wrong statement, this should be and so, this is r_i, i should be $-a_i$ for all of zero through n right. If I negate that I get minus polynomial.

Also, if I have two polynomials, I can add them. Now, two polynomials of degree n and m. Now, since they have degree n and m, it is possible that they are unequal. So, the resulting polynomial naturally will have a degree which is larger of these two which is max(n, m). So, in the max of nm from 0 to let us say if we take that n is greater, m is less. So, from 0 to m, the coefficients must be added because they are there in both whereas from m+1 to n it is the coefficients of the larger polynomial which will only exist because the rest of it is 0. So, this is how it is defined that if I add these two polynomials, then up to the smaller degree, the coefficients are added then the remaining coefficients of the larger polynomial is basically put at the end of it. And this is the definition in terms of the coefficients. And since we can add, we can negate we can do subtraction.

(Refer Slide Time: 27:15)

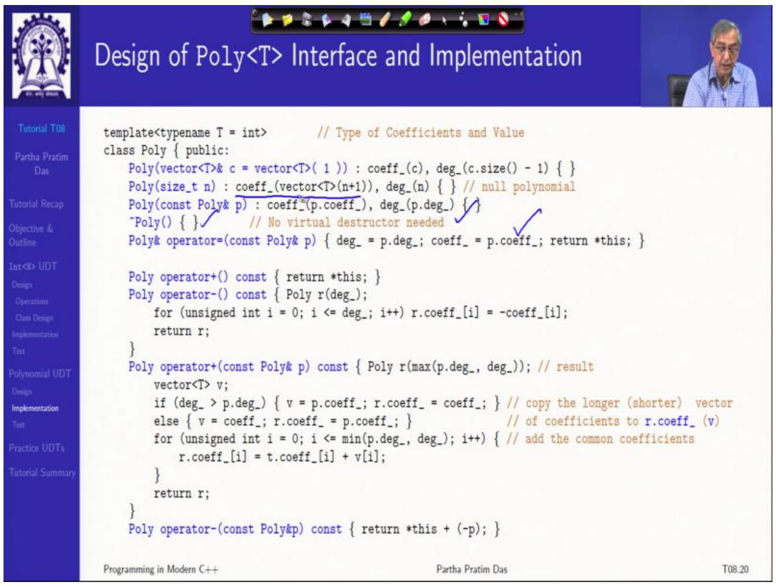


Design of Poly<T> Interface and Implementation

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

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

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

Programming in Modern C++ Partha Pratim Das T08.20

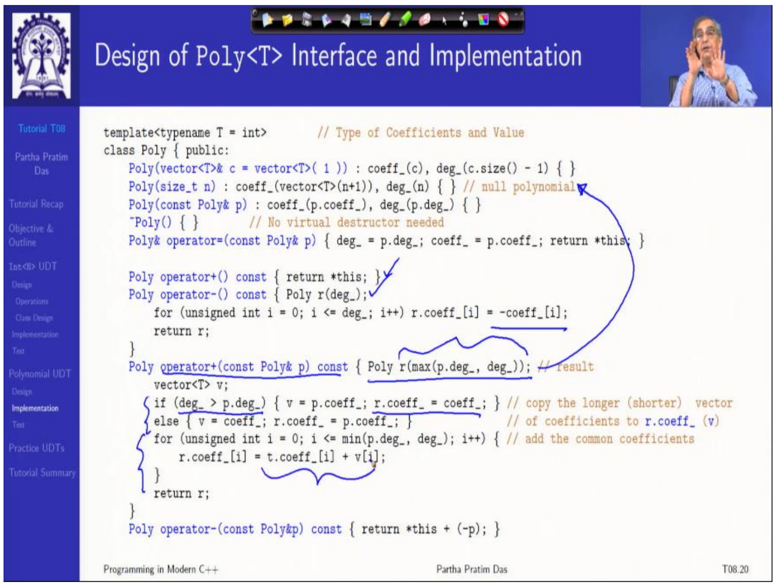


Design of Poly<T> Interface and Implementation

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

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

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

Programming in Modern C++ Partha Pratim Das T08.20



Design of Poly<T> Interface and Implementation

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

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

Programming in Modern C++ Partha Pratim Das T08.20

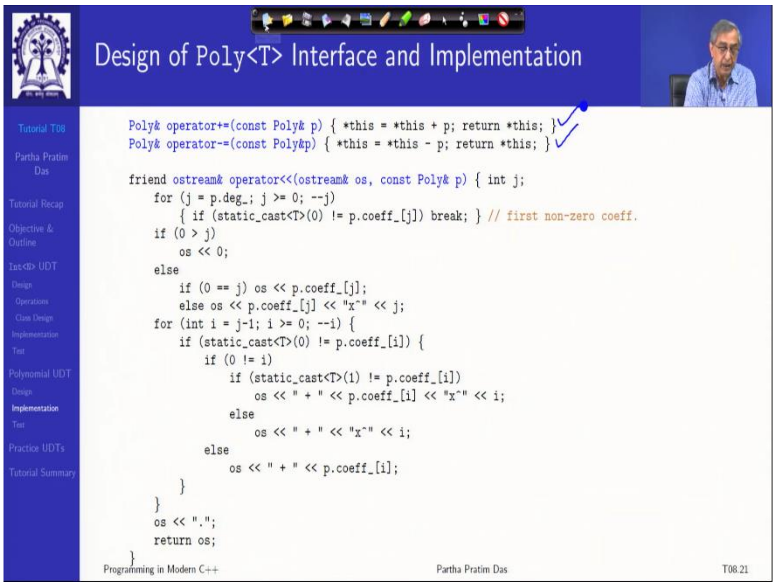
Obviously, we can define multiplication of polynomials, division of polynomials are relatively more difficult because a polynomial may not divide another polynomial totally. It might be an infinite process or it has to be defined in a different way. So, but at least multiplication can be defined and I will leave that as an exercise. But with this itself, we can do quite a lot.

So, what are the what is the UDT that we have? I have a polynomial constructor which takes a vector of coefficients and build the polynomial by default. It takes a vector having a single value 1. So, by default, you can have a polynomial having just a constant 1. So, this is your default as well as parametric constructor or the polynomial may just take the size just take the degree n and build the polynomial which is a null polynomial. So, you put n to degree and do a coefficient vector of n+1. If you do that, then it naturally means that all coefficients are 0. So, you get a null polynomial of nth degree.

Other than you can copy construct, you can copy assign you can destruct of course, that, in turn basically destructs the vector by calling the destructor of the vector. This is the sign preservation, this is the negation of the polynomial. So, you are simply negating, this is that add operation. So, you first take a result polynomial as a temporary and then what you do you compare the degree of the two polynomials, the current one and the one that you are adding and you check which one has a higher degree. Whichever has a higher degree, you copy that to the result because the result is going to have that degree.

You can see that we have already created this result polynomial as a null polynomial, right that is, that was what was the purpose. So, we choose depending on the degree, we copy the larger coefficients of the larger polynomial into the result polynomial and then we copy, then we add the coefficients up to the smaller degree from both the polynomials and put it to the result polynomial. So, this is the simple addition, just the formula we wrote, we are just implementing that. Similarly, you can do its subtraction here.

(Refer Slide Time: 30:03)



Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

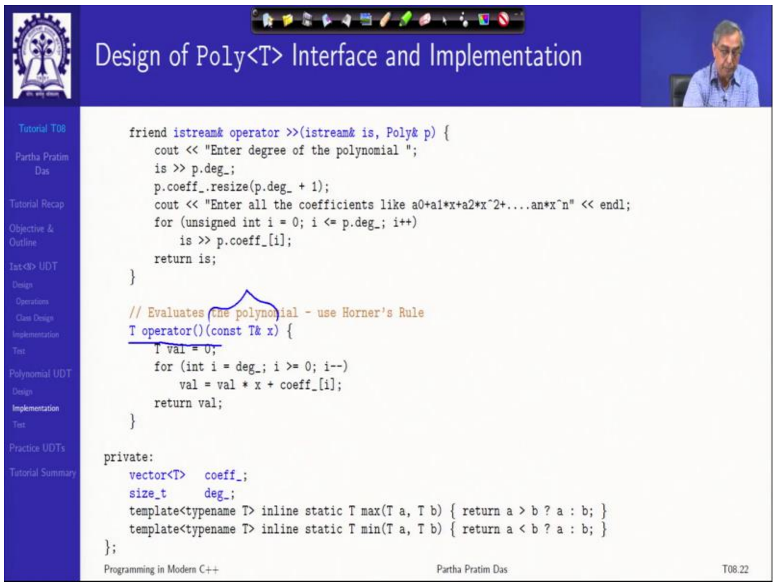
```
Poly& operator+=(const Poly& p) { *this = *this + p; return *this; }✓
Poly& operator-=(const Poly&p) { *this = *this - p; return *this; }✓

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

Programming in Modern C++ Partha Pratim Das T08.21

So, you can have you know plus assigned, minus assigned operators based on that and then you can write out a polynomial. This code looks a little bit clumsy because possibly that is the best way to write it because there are several spatial cases you need to take care for example, if a coefficient is 0, you do not want to write that term. If a coefficient is 1 you do not want to write 1 in place of that you just want x to the power that.

If say your constant is 0, then you just do not write that constant and so on so forth, you do not want to write if constant is 5, you do not want to write it as 5 into x to the power 0, you will just have to write it as 5. So, these different cases are checked and this is just a you know, simple logic-based C kind of program for output streaming. Similarly, you have input streaming which is simpler.

(Refer Slide Time: 31:02)



Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<8> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

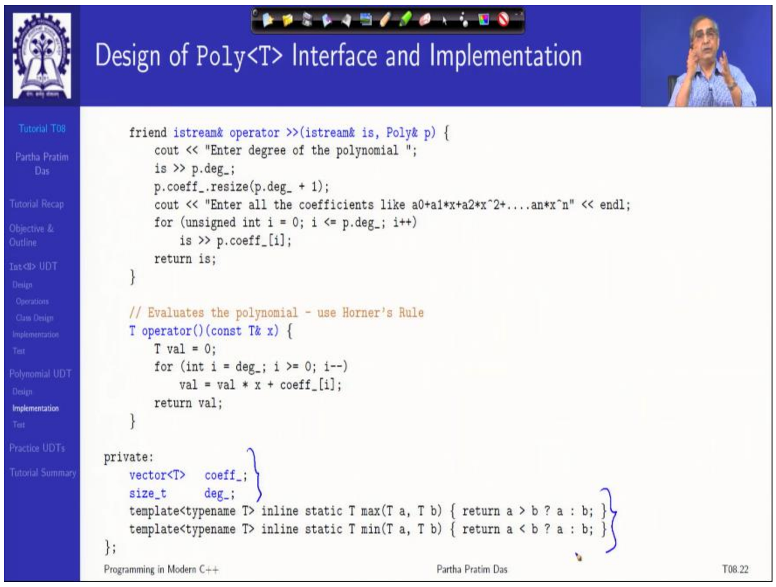
Test

Practice UDTs

Tutorial Summary

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

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

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

Programming in Modern C++ Partha Pratim Das T08.22



Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<8> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

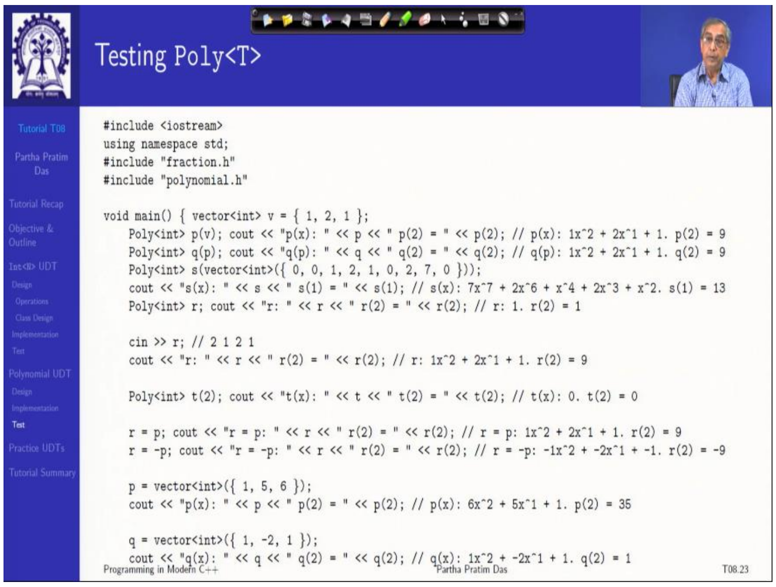
Tutorial Summary

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

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

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

Programming in Modern C++ Partha Pratim Das T08.22

And finally, you need a function to evaluate the polynomial. So, what I do is instead of giving a different function, I actually overload the function called operator. So, this is a polynomial UDT which is also a polynomial functor so that if I invoke the polynomial in this class, if I invoke the function call operator with any given parameter value, then the polynomial will automatically get evaluated. So, you can see how things are setting up together. Here the data members are two convenience functions min() and max(), if you use, you can also use library functions from standard library but I just wanted to keep everything within this class.

(Refer Slide Time: 31:55)



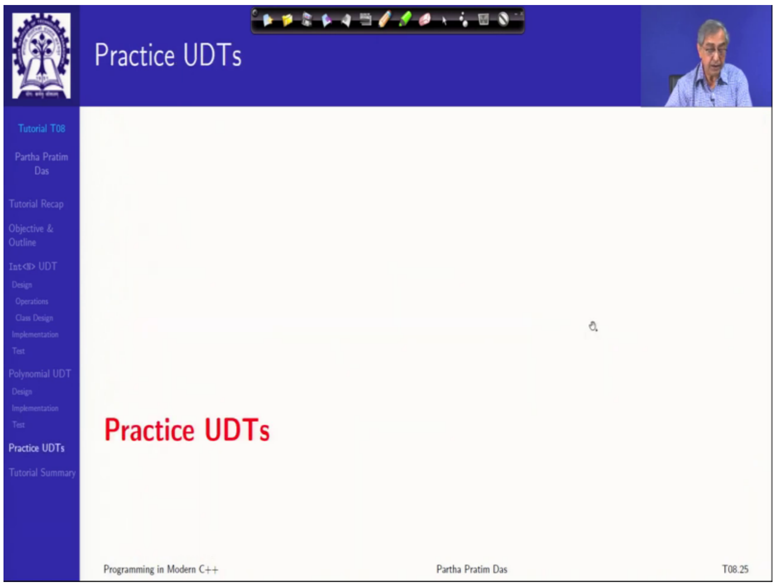
Testing Poly<T>

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

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

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

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

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

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

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

Programming in Modern C++ Partha Pratim Das T08.23

So, with this you can write the test code building different constructing different polynomials, reading polynomials from input, negating polynomials, adding polynomials, subtracting polynomials and so on.

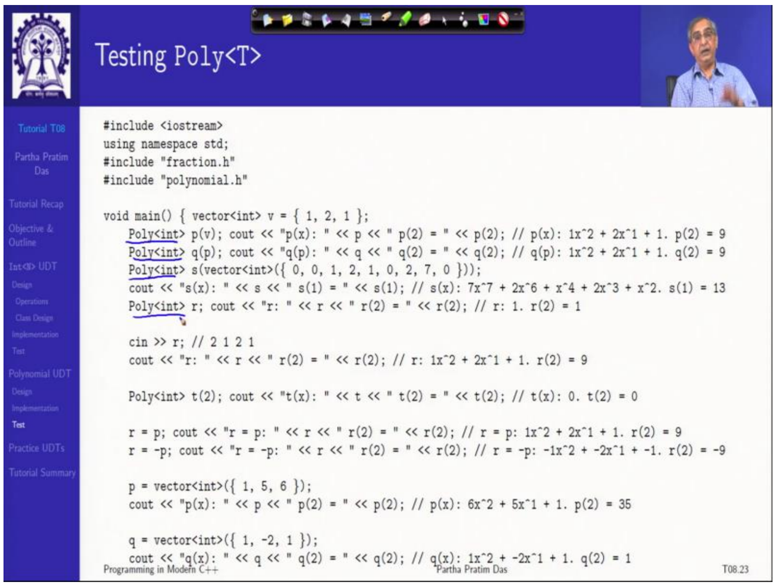
(Refer Slide Time: 32:11)



Practice UDTs

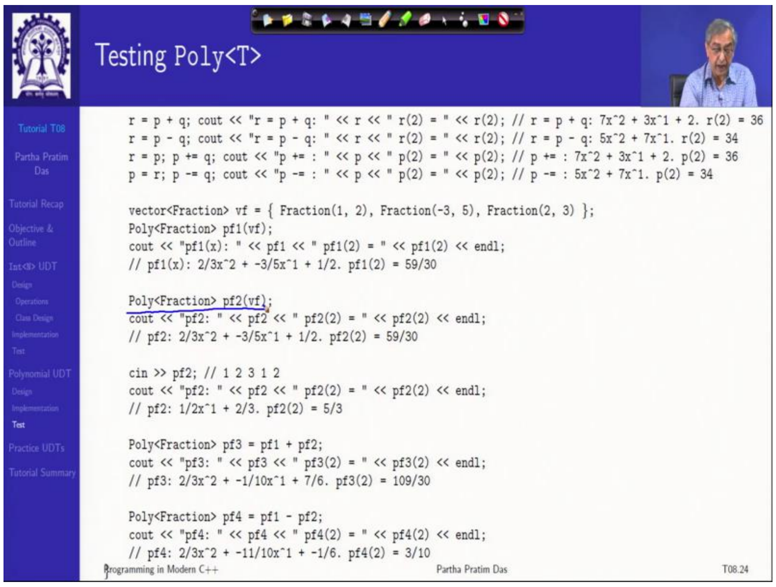
Practice UDTs

Programming in Modern C++ Partha Pratim Das T08.25



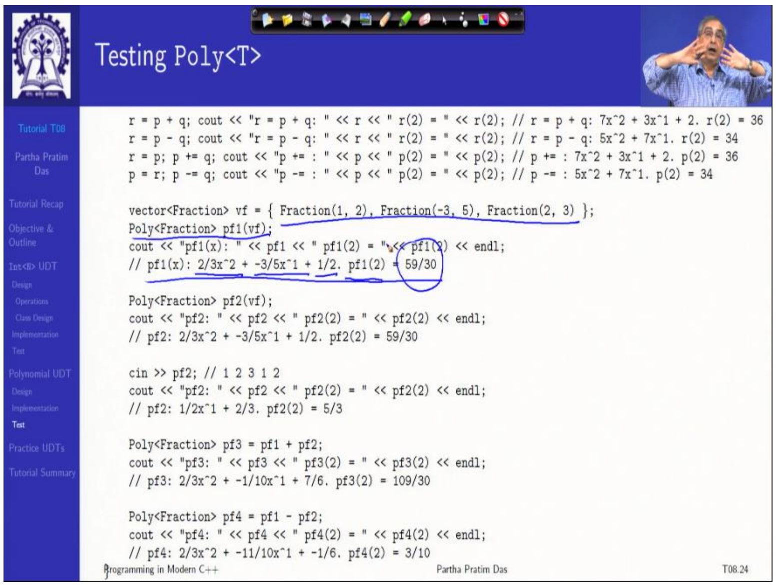
Testing Poly<T>

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

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

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

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

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

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

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



Testing Poly<T>

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

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

    Poly<Fraction> pf2(vf);
    cout << "pf2: " << pf2 << " pf2(2) = " << pf2(2) << endl;
    // pf2: 2/3x^2 + -3/5x^1 + 1/2. pf2(2) = 59/30

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

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

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



Testing Poly<T>

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

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

    Poly<Fraction> pf2(vf);
    cout << "pf2: " << pf2 << " pf2(2) = " << pf2(2) << endl;
    // pf2: 2/3x^2 + -3/5x^1 + 1/2. pf2(2) = 59/30

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

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

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

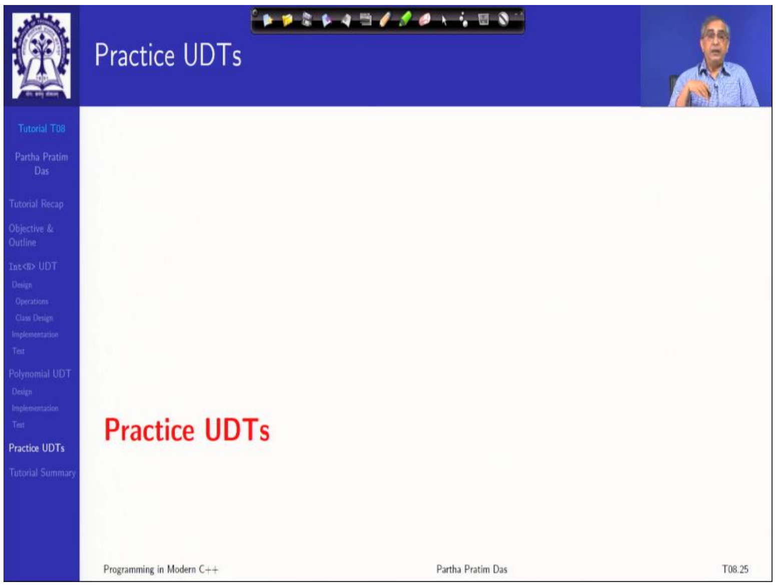
Programming in Modern C++ Partha Pratim Das T08.24

And, if you note, let me just go back, if you note, then all these polynomials I have done, I have taken it to be polynomial of integer. So, integer coefficients and naturally you will do an integer evaluation. But interestingly that where we can you can now keep make things mixed is you can make use of the Fraction class that we have already designed and make a polynomial of Fractions that is where coefficients are fractions, even the value that you evaluate with is a fraction.

So, if I define say I define a vector of 3 fractions 1/2, -3/4, 2/3 and construct a polynomial. If I print it, this is what I get $2/3x^2$, –3/5x and 1/1 and if I evaluate this by the functor for 2 I get a fraction. So, now, everything can happen in fractions. So, you can see that we had built a Fraction class (Fraction UDT) to operate only with fractions.

We have built a polynomial UDT to operate of polynomials of various types and now, we are using the fraction to use in a polynomial. So, I can have a polynomial with fractional coefficients and fractional value. I can use my limited size int also to define fraction. I can use polynomial of Int<N> sized numbers and so on.

(Refer Slide Time: 34:05)



Practice UDTs

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT
Design
Operations
Class Design
Implementation
Test

Polynomial UDT
Design
Implementation
Test

Practice UDTs

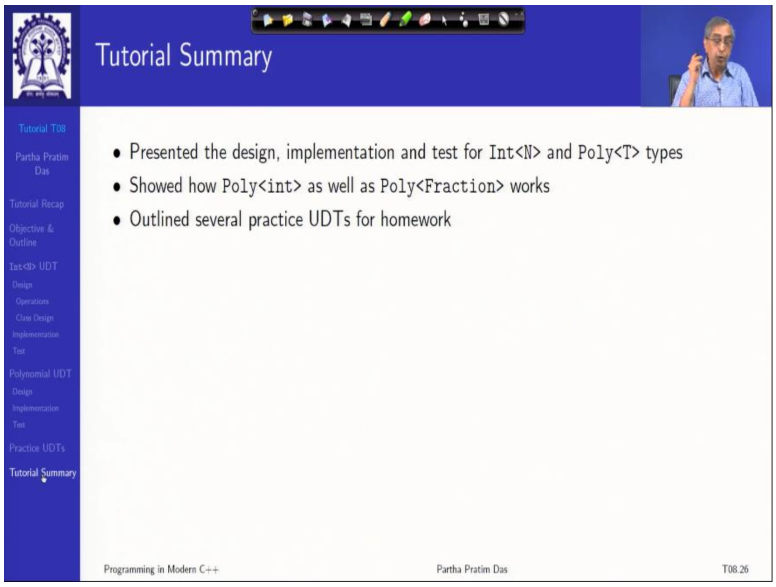
Tutorial Summary

Practice UDTs

Programming in Modern C++ Partha Pratim Das T08.25

So, there are several ways these things can be mixed and, in the practice UDTs, I am not going through the slides. I have given a number of problems for you to design and practice and try out.

(Refer Slide Time: 34:19)



Tutorial Summary

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

Tutorial T08

Partha Pratim Das

Tutorial Recap

Objective & Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Programming in Modern C++ Partha Pratim Das T08.26

So, to summarize, we have presented the design implementation and testing for Int<N>, limited size N, integer signed integer and polynomial type of based on any coefficient and value type. And we have showed how Poly<Int> and Poly<Fraction> works and outline several practice UDTs for homework which you should try out and with that, I think you will get a good command over doing data types or writing classes in C++. Thank you very much for your attention. Meet you in the next session.