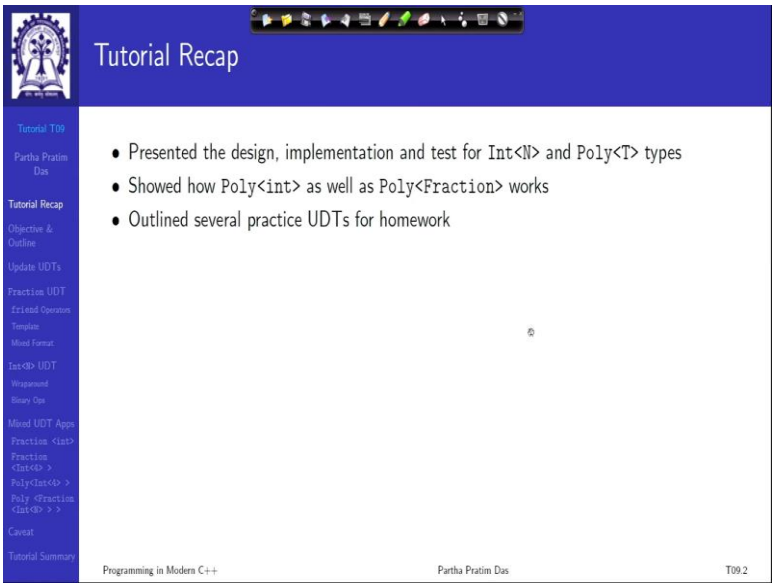
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 09
How to Design a UDT Like Built-in Types Part 3 Updates and Mixes of UDTs

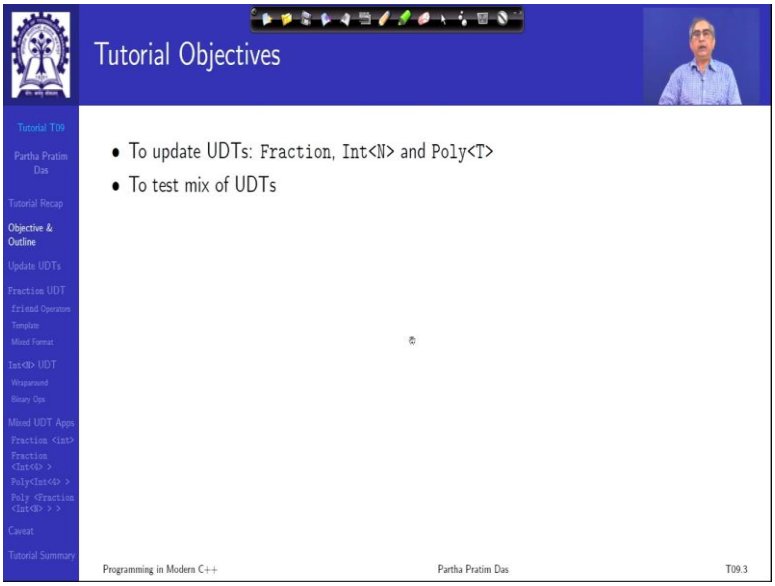
Welcome to Programming in Modern C++. We are going to discuss Tutorial 9.

(Refer Slide Time: 00:39)



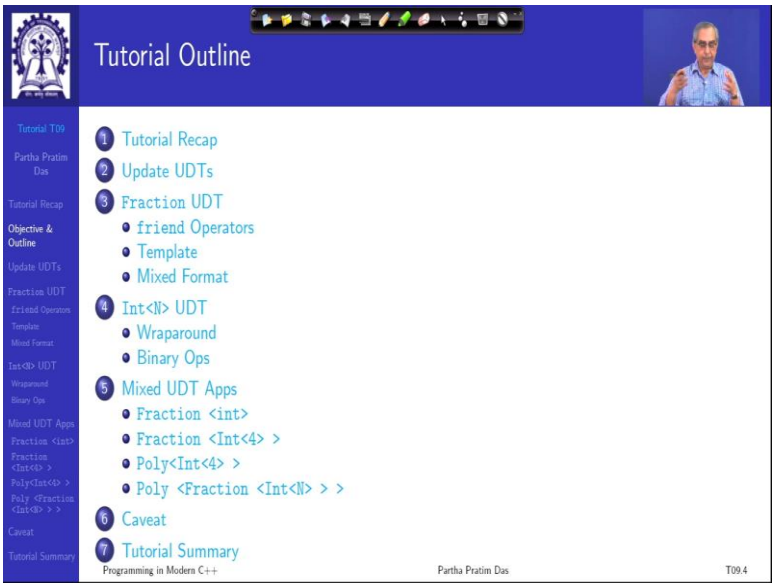
We have been talking about designing user-defined types much like the built-in types, how to design them, how to implement and finally test. And after a simple design of a Fraction type, in the last tutorial we have talked about implementing, designing and implementing a limited size integer type with Int kind of operations but restricted to a given number of bits. And we have also discussed the design of a polynomial type. And we showed how polynomial of integer or polynomial of Fraction would work. And we gave some outline some practices for UDTs for homework.

(Refer Slide Time: 01:37)



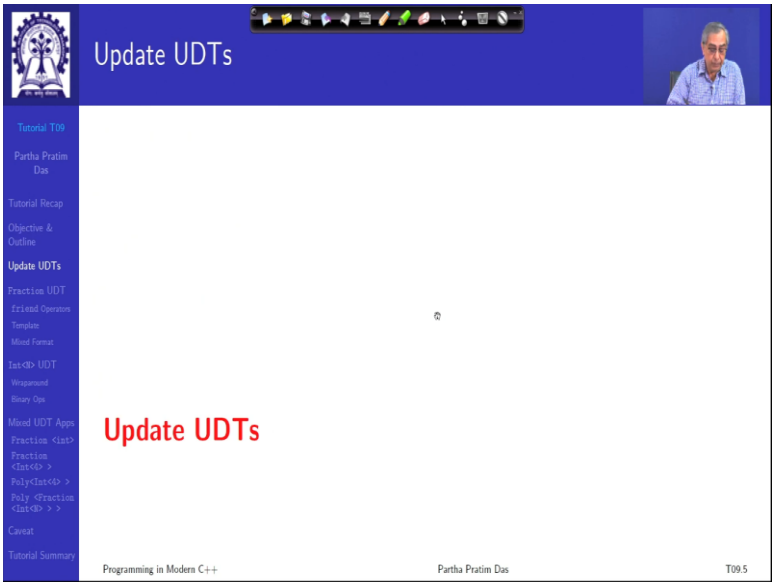
Continuing on the last two tutorials on this, we will talk about some updates in the Fraction limited size integer and polynomial types. And then we will try out mixing these data types, user defined data types that we create much like the way we can mix and match around with the built in types.

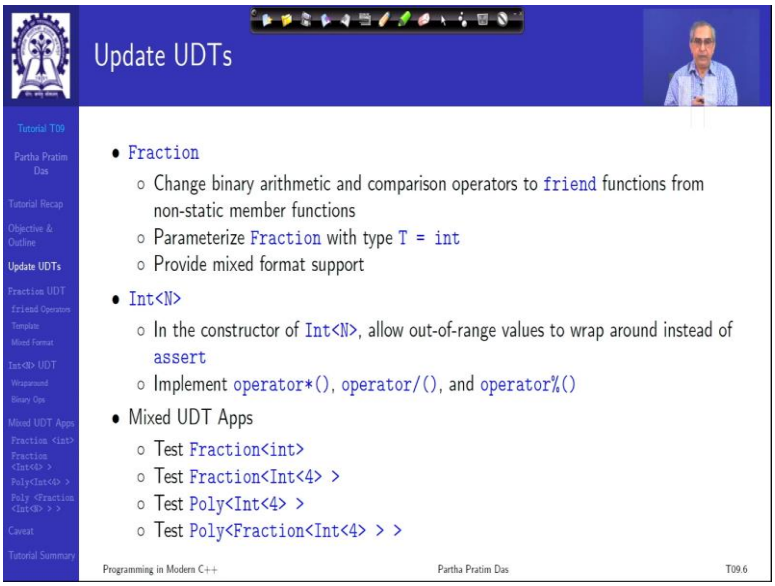
(Refer Slide Time: 2:04)



So, this is the outline which will be there on the left panel as.

(Refer Slide Time: 02:09)





So, let us pick up some update agenda. The first, if we talk about the Fraction class, the Fraction class had implemented the operators -- binary, arithmetic and comparison operators primarily, in terms of non-static member functions. So, that can work in a limited way and as it is most ideal in majority of cases to provide operators as friend functions.

So, we will change these non-static member functions into friend functions which is straightforward. The second is when we talked about the implementation of the Fraction we had assumed that the numerator will be of the Int type and the denominator will be unsigned Int type, expecting that unsigned Int, the denominator cannot be negative and so on.

But it is not necessary that the implementation be on the Int type. I can do the implementation on any integral type and the Fraction will behave according to that integral type. So, for this we parameterize the Fraction as a template with a type parameter T which is defaulted to Int. So, that it falls back on the last design that we did.

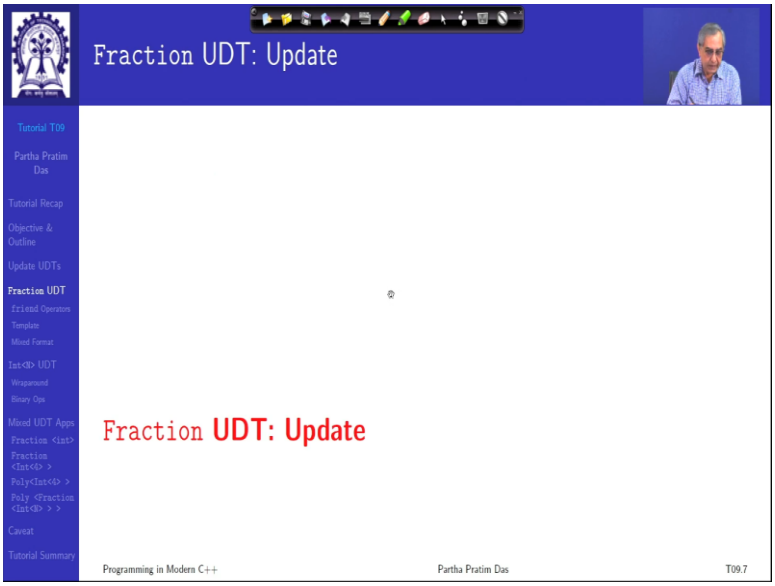
And the third is, we will provide, we intend to provide mixed format support, as we have discussed that Fractions, if particularly, if they are greater than 1 in their absolute value, then we have an alternate way of writing the Fraction where we put the maximum integer as a whole number and then the Fractional part. So, this is called the mixed format. So, that support is not present in that class.

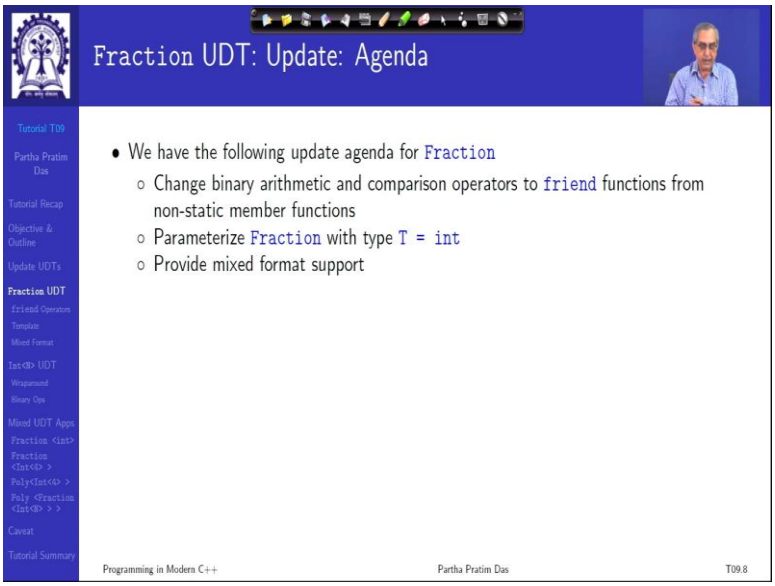
So, we try to provide that support. The second set of update agenda comprise the Int, limited size integer where in the constructor we if an integer was out of bound, so far as the n number of bits are concerned then we simply did an assert and stop the program. But that is not what actually happens with Int. In Int it actually wraps around.

If you cross the maximum Int, it will become mean Int and then start going there, if you cross the MIN_Int become even, try to become even smaller it will jump back to MAX_Int. So, that wrap around so behavior we would like to put in. And then operators -- multiplication, division, residue were not supported. So, with this wrap around, they will become very easy to support.

Finally, we will take these three types and mix and match around them to write a number of test application and test out, which not only individually tests the data type that we have created but it checks whether the mix will also work. We did little bit of that when we tried to do polynomial of Fraction but here you would try out a number of combinations.

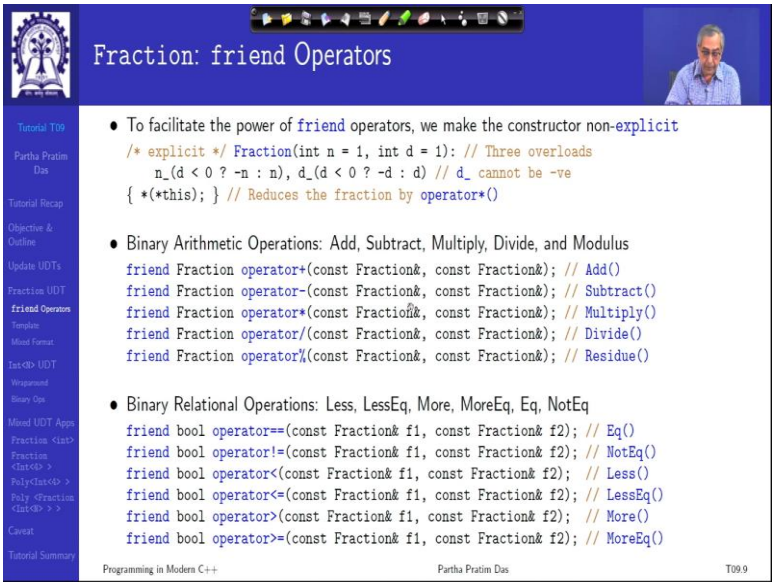
(Refer Slide Time: 05:33)





So, with that agenda we start with the update of the Fraction with the three agenda that we have I have just explained and taking up one by one.

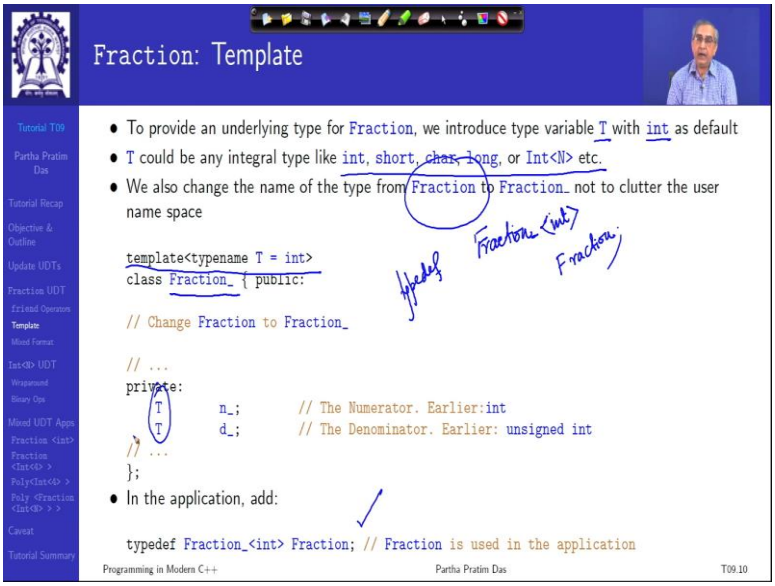
(Refer Slide Time: 05:42)



First is converting these non-static member functions for binary, arithmetic as well as relational operators is a trivial task. So, you just introduce a first parameter which will be, which earlier was hidden because it was this, the current object itself, now, it is any of these and make that function friend. I am not going into the details of their implementations, I am sure you will be able to do that.

In the process, we are also removing the explicit constructor qualifier that we had. And the reason for that will become soon clear because if a constructor is explicit then it is, then we cannot just take an integer, take a pair of integers and use them in the context of a Fraction, we will have to specifically write Fractions. So, to remove that restriction, we remove the explicit qualifier here.

(Refer Slide Time: 06:48)

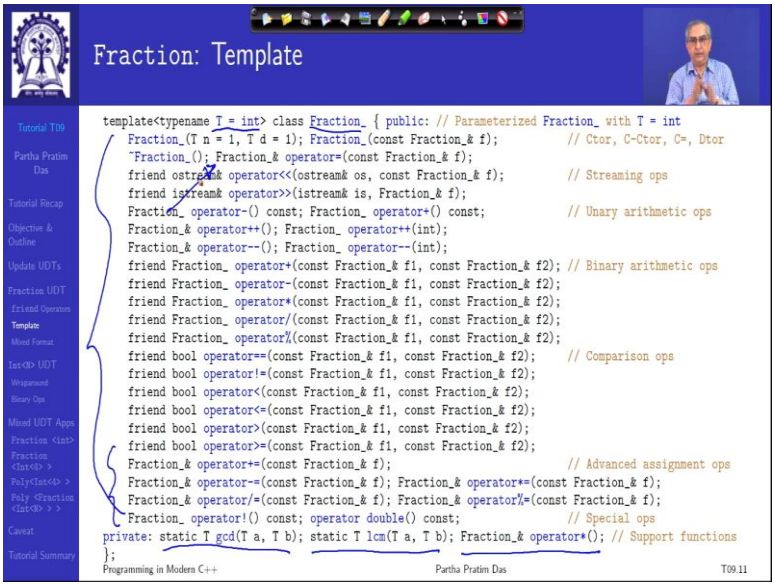


Making it into template is also relatively easy because all that we need to do is we intend to use a templatized type T with default as Int. And this type T could be any of the integral types, int, short, char, long, any of the signed integral types. And it could be some integral types that the user has defined we have designed like Int<10>. And then we just need to wrap the function class within this template.

So, if we do that then actually the class references now in the code will become like whatever, if we want to do Int then it will be like this. Now, that naturally becomes little elaborate for the user to write. So, what we want to do? We want to typedef this as simply as Fraction. We want to do this kind of a typedef, so that the user can just use it as a Fraction. Because the user already knows the underlying type the user has specified.

But this will not be possible because the name will clash. So, what we are doing is we are simply changing the name of the class in the definition with a simple underscore added at the end of it. With that the user space remain free and the user can use simply the name Fraction with a typedef by doing something like this. Naturally, the types of the data members as well as wherever the type underlying type is involved, we have to change to T. Those details are pretty straightforward and you will be able to do that.

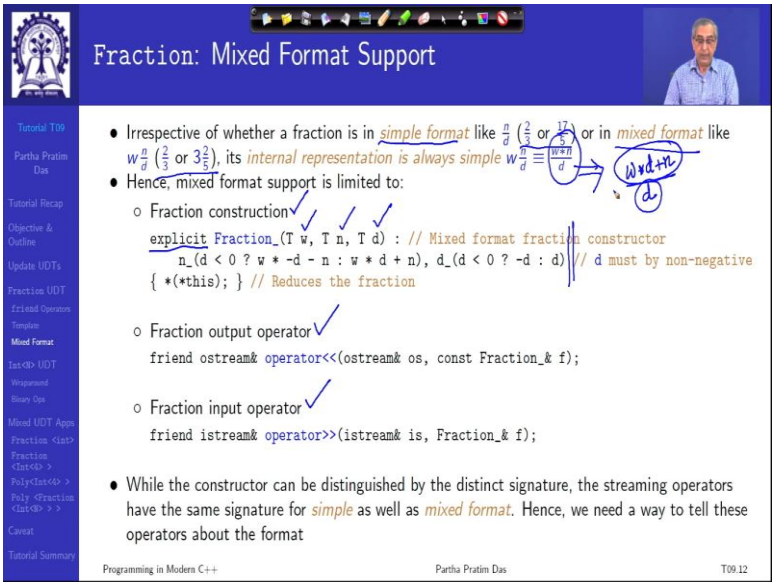
(Refer Slide Time: 8:40)

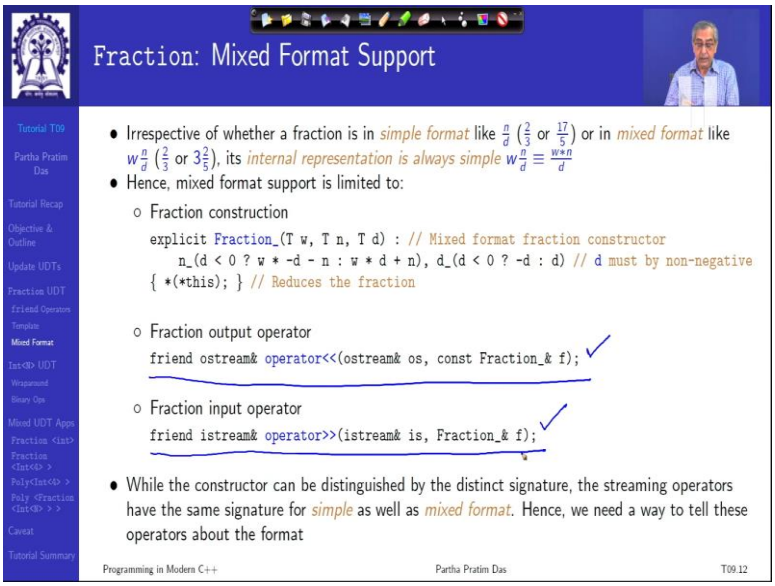


With that change, if we now take a look at the templatize Fraction class; this is how it will look. I have a template parameter defaulted. This is a Fraction class. Then I have all these interface member functions, here I have the, of course, I will have the data members and then I have some support functions out here. So, this is now a templatized Fraction class which has, I mean, all operators, majority of operators that people need to use where you actually do not have a necessarily a current object but you talk about pairs of objects as such, those will be friend functions.

But some specific like assignment operators and assignment operator is, where is the assignment operator, here is an assignment operator, the assignment operator and the advanced assignment operators certainly will not be friend because they need to change the current object on which the assignment is being done. So, this achieves the first two objectives of update.

(Refer Slide Time: 09:57)





So, let us look at the third one. That is if I want to write, if I want to provide support for mixed format. So, Fraction can be simple format written like this or it could be mixed format where 17/5 will be written as 3+2/5. Now, whatever I mean whether, it you want to give a mixed format view of the Fraction, internally, we will always maintain it as a sample format because that is what makes sense that is what it really is.

Mixed format is just for the interaction with the user, with the external world. So, therefore there are three places where the changes are required. One is for the mixed format construction I need a separate constructor which can take three arguments now including the whole number part I need to change the output and input operators. Because they will take input by reading or print out the output in a fixed in a mixed format of the Fraction.

Now, the support in the constructed is simple. We will just have another constructor which has now three parameters. We do not default anything. So, that there is no confusion with the sample format constructors that we had. And we make that this must be explicit, so that I mean mixed format is when you are using, you must, it must be clear that you are using that.

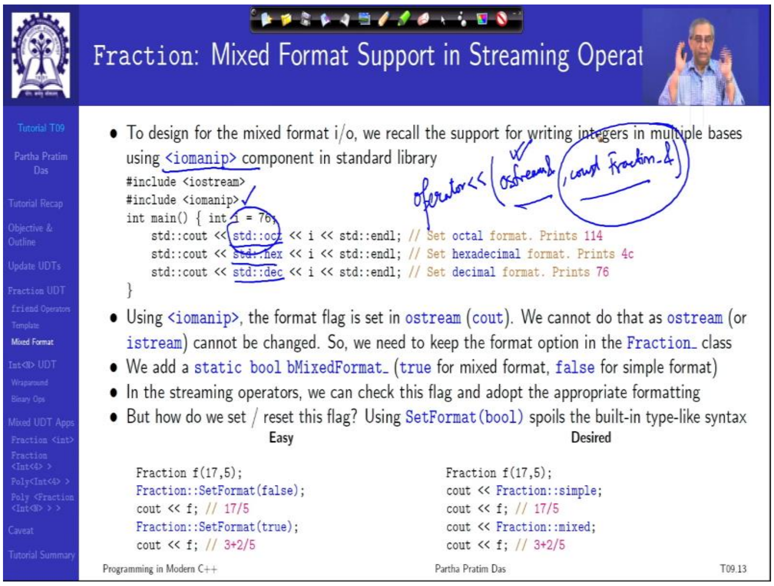
Rest of the support is simple. All that we need to consider is the fact that the mixed format as inputted will be internally kept as w times, there is a typo here so which I will correct. It is basically (w* d+n)/d. So, this is the numerator that you will set and denominator, of course, would be the same as is given. I will correct that type, please note. Now, this is easy because the constructor has a distinct signature. So, it is easy to understand. It is easy for the user to use as well as for us to implement.

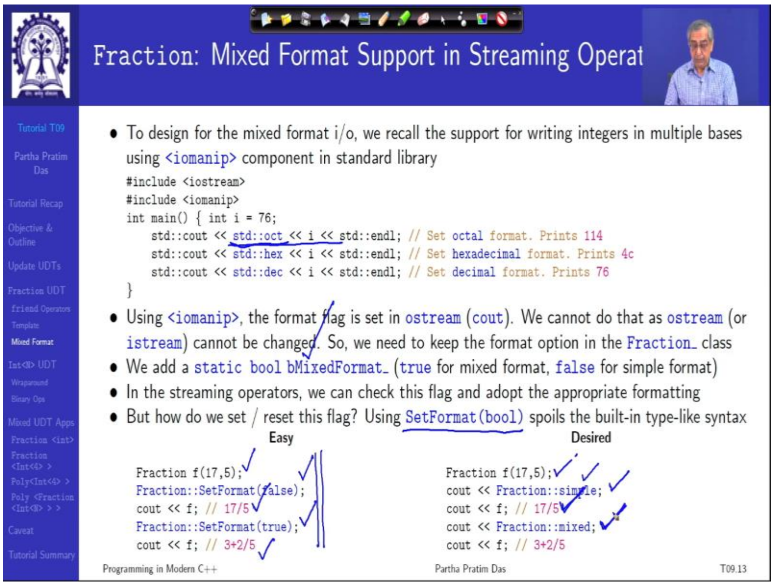
The issue turns out in terms of the input and output parameters, output operators. For an output operator, certainly, we have already given an overload for the output streaming operator for the Fraction object. Now, how do I tell the compiler, how do I tell the program or the system that this particular output should happen in simple format or in mixed format?

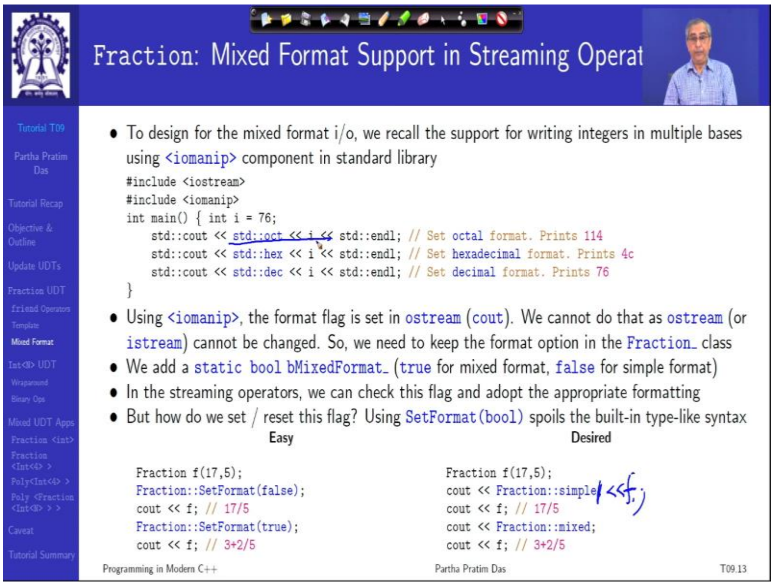
That there is no place to input that information. Because as you know that these streaming operators have a fixed signature their binary operator, so they take two arguments, one is the string and the other is the object on which. Similar thing will happen for the input streaming operator also.

If you want to say that I want to take the input in the mixed format then naturally I need to take three separate integers. Whereas, if I want to take the input as simple format then I need to read two integers. So, the user must know, in what way we are doing. And for that the operator has to know what is the way to go. So, the question is, how do we implement this kind of a behavior?

(Refer Slide Time: 13:44)







So, we just fall back and recall a similar support that we have seen in the standard library to, for example, print out integers in terms of different base of the number system. For example, I can print it out in terms of decimal which is default but again I can print it out in terms of octal format or in hexadecimal format and so on. And that support is available in iomanip and using that I can using that, including that I can just stream it like this. So, I first stream the format in which I want and then I stream the object.

Now, the question is how this does, is this information passed? So, if I look at operator say, output streaming then you have two. This is a binary operator and you have the const Fraction. So, these are the two parameters you have. So, the operator must get this information through one of these parameters that is very clear. So, what iomanip does?

It is implemented by the library. So, it actually when I do a streaming of std::oct, it accesses the cout, output stream object and sets a particular flag which is supported already. So, these formats are already supported iomanip is just giving us an access to manipulate that. So, it is actually using this property of the first argument to do this.

Now, our problem is, we are writing user code. So, we cannot change the, we cannot introduce a flag for writing mixed format Fraction in the ostream we do not have that option. So, for us this becomes, we will have to make use of the second argument to do this. Now, we cannot directly do that.

Because in that case the particular Fraction that we are about to print, how will that Fraction again know that it needs to print in mixed format or in simple format and so on. But overall if you just think of that if the information has to come from the Fraction class itself. So, what we can think of?

We can think of having a static bool flag which I can set to true to mean mixed format and false to mean simple format. Now, what if I do that then what will the output stream operator do? Now, while the output stream operator goes to output. It will first check the flag in the Fraction class that is a static flag. So, it will be able to check. It is already a friend function. So, it has access to that flag. So, it will check and accordingly it will decide which way to print. So that, that kind of solves.

But that still leaves with a question of how do I selectively set that flag. So, one way could be that I again design a static public member function, say SetFormat() which takes a bool if I

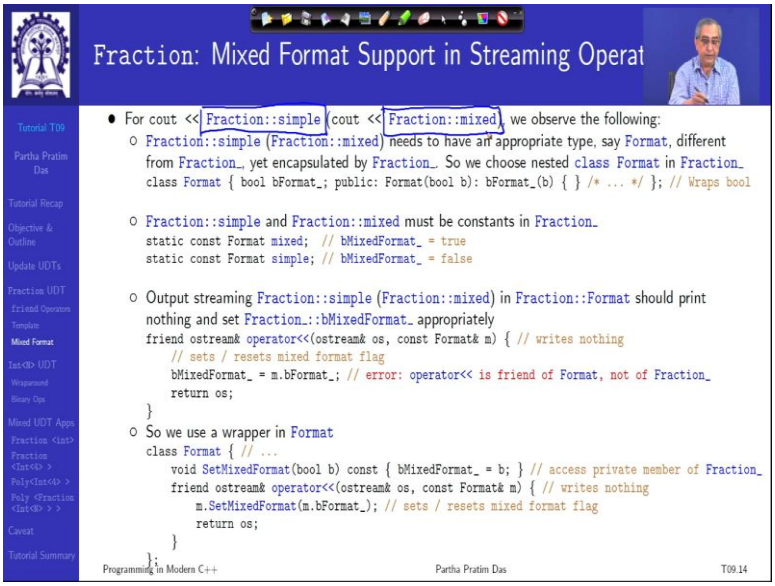
pass true. It will set this flag as true meaning mixed format. If I said false then it will set it to false meaning simple format.

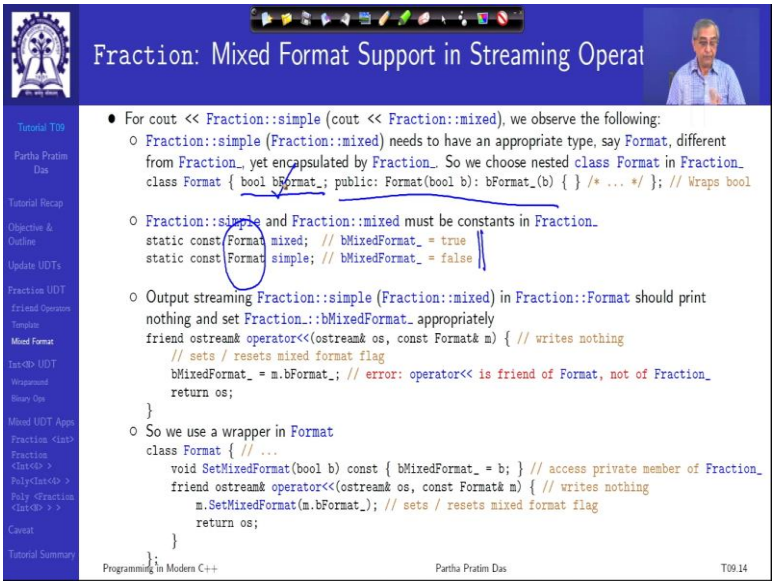
So, a code will look something like this in that case, that you have constructed a Fraction which is actually not proper. I can set the flag to false and I print, it should print 17/5. I can set this to true and it should print 3+2/5 in the mixed format. This is easy to implement, you can easily do that. But the question is it is not, what is natural, all that we are trying to do is to create data types which will behave like the built-in types. So, how does built-in type do it?

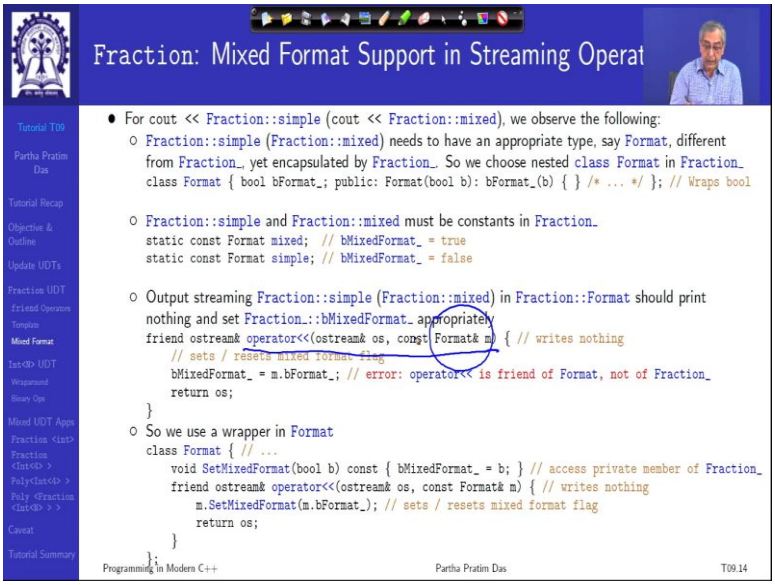
Built-in type is doing it like this, that it streams the format information to the output stream and then it streams the object and it automatically happens. So, in that same context, I will try to do, I would like to have something like this that if we have that format information as some value or some flag or some object as in the Fraction class as simple, meaning simple format and another as mixed meaning.

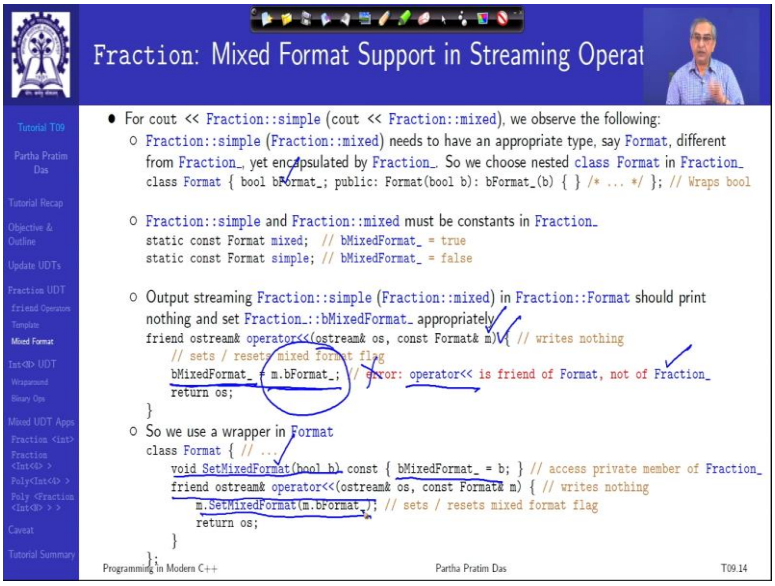
So, Fraction::simple much like std::oct can be streamed to the output stream. And after that when I do the printing, it should print in the format in which I have actually streamed. So, this is given. This is simple. This will print as simple, given I have streamed mixed; this will print as mixed format. So, this also gives me the advantage that instead of breaking it here, I can simply also just stream it right away, which makes it look exactly like the same thing. So, the question then remains as to how do I do this streaming?

(Refer Slide Time: 19:50)




Fraction: Mixed Format Support in Streaming Operat
Tutorial T09
Partha Pratim Das
Tutorial Recap
Objective & Outline
Update UDTs
Fraction UDT
friend Operators
Template
Mixed Format
Int<B> UDT
Wraparound
Binary Ops
Mixed UDT Apps
Fraction <int>
Fraction <Int<B> >
Poly<Int<B> >
Poly <Fraction <Int<B> > >
Caveat
Tutorial Summary
• For cout << Fraction::simple (cout << Fraction::mixed), we observe the following:
○ Fraction::simple (Fraction::mixed) needs to have an appropriate type, say Format, different from Fraction_, yet encapsulated by Fraction_. So we choose nested class Format in Fraction_
class Format { bool bFormat_; public: Format(bool b): bFormat_(b) { } /* ... */ }; // Wraps bool
○ Fraction::simple and Fraction::mixed must be constants in Fraction_
static const Format mixed; // bMixedFormat_ = true
static const Format simple; // bMixedFormat_ = false
○ Output streaming Fraction::simple (Fraction::mixed) in Fraction::Format should print nothing and set Fraction_::bMixedFormat_ appropriately
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
// sets / resets mixed format flag
bMixedFormat_ = m.bFormat_; // error: operator<< is friend of Format, not of Fraction_
return os;
}
○ So we use a wrapper in Format
class Format { // ...
void SetMixedFormat(bool b) const { bMixedFormat_ = b; } // access private member of Fraction_
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
m.SetMixedFormat(m.bFormat_); // sets / resets mixed format flag
return os;
}
};
Programming in Modern C++
Partha Pratim Das
T09.14


Fraction: Mixed Format Support in Streaming Operat
Tutorial T09
Partha Pratim Das
Tutorial Recap
Objective & Outline
Update UDTs
Fraction UDT
friend Operators
Template
Mixed Format
Int<B> UDT
Wraparound
Binary Ops
Mixed UDT Apps
Fraction <int>
Fraction <Int<B> >
Poly<Int<B> >
Poly <Fraction <Int<B> > >
Caveat
Tutorial Summary
• For cout << Fraction::simple (cout << Fraction::mixed), we observe the following:
○ Fraction::simple (Fraction::mixed) needs to have an appropriate type, say Format, different from Fraction_, yet encapsulated by Fraction_. So we choose nested class Format in Fraction_
class Format { bool bFormat_; public: Format(bool b): bFormat_(b) { } /* ... */ }; // Wraps bool
○ Fraction::simple and Fraction::mixed must be constants in Fraction_
static const Format mixed; // bMixedFormat_ = true
static const Format simple; // bMixedFormat_ = false
○ Output streaming Fraction::simple (Fraction::mixed) in Fraction::Format should print nothing and set Fraction_::bMixedFormat_ appropriately
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
// sets / resets mixed format flag
bMixedFormat_ = m.bFormat_; // error: operator<< is friend of Format, not of Fraction_
return os;
}
○ So we use a wrapper in Format
class Format { // ...
void SetMixedFormat(bool b) const { bMixedFormat_ = b; } // access private member of Fraction_
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
m.SetMixedFormat(m.bFormat_); // sets / resets mixed format flag
return os;
}
};
Programming in Modern C++
Partha Pratim Das
T09.14


Fraction: Mixed Format Support in Streaming Operat
Tutorial T09
Partha Pratim Das
Tutorial Recap
Objective & Outline
Update UDTs
Fraction UDT
friend Operators
Template
Mixed Format
Int<B> UDT
Wraparound
Binary Ops
Mixed UDT Apps
Fraction <int>
Fraction <Int<B> >
Poly<Int<B> >
Poly <Fraction <Int<B> > >
Caveat
Tutorial Summary
• For cout << Fraction::simple (cout << Fraction::mixed), we observe the following:
○ Fraction::simple (Fraction::mixed) needs to have an appropriate type, say Format, different from Fraction_, yet encapsulated by Fraction_. So we choose nested class Format in Fraction_
class Format { bool bFormat_; public: Format(bool b): bFormat_(b) { } /* ... */ }; // Wraps bool
○ Fraction::simple and Fraction::mixed must be constants in Fraction_
static const Format mixed; // bMixedFormat_ = true
static const Format simple; // bMixedFormat_ = false
○ Output streaming Fraction::simple (Fraction::mixed) in Fraction::Format should print nothing and set Fraction_::bMixedFormat_ appropriately
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
// sets / resets mixed format flag
bMixedFormat_ = m.bFormat_; // error: operator<< is friend of Format, not of Fraction_
return os;
}
○ So we use a wrapper in Format
class Format { // ...
void SetMixedFormat(bool b) const { bMixedFormat_ = b; } // access private member of Fraction_
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
m.SetMixedFormat(m.bFormat_); // sets / resets mixed format flag
return os;
}
};
Programming in Modern C++
Partha Pratim Das
T09.14

So, let us take a look at how we can do this. Now, think about this. I think we have yeah. So, now, if istream this to this format setter object to see out, what is the type of that object? If it is a Fraction object itself then it will invoke the behavior of the output streaming friend function that we have written for the Fraction object and it will behave in that way try to print the Fraction. So, it cannot be that. It has to be some other object type. The basic information that I want to set is whether it is true or it is false.

Now, say this object Fraction simple or Fraction mixed actually is a Boolean information. But I cannot make it a bool type object because if I stream a bool to see out then it will behave like the built-in Boolean type, I will not be able to control that it will just print true false or 1 0 something like that.

So, the conclusion is it has to be a type which kind of wraps a Boolean information but is not a bool by itself it cannot be a Fraction, but certainly, it is a support is within the Fraction. So, it has to be encapsulated by the Fraction. So, these are the two basic information that we reason out.

So, it say suppose, we have a format type which is encapsulated within Fraction. Since it is encapsulated within Fraction, we choose to make this type, this class a nested class within Fraction. So, it will be totally within Fraction, it will not clutter anything outside. So, in that format class, we will have a bool flag and a constructor to set that bool flag for the object. So, this is basically a wrapper of bool.

And given that we can make two kinds of static constant objects mixed and simple which are of this type format which in mixed, the format object will have the bFormat set as true, in the other one it will be set as false. So, to set this I can now have an output streaming operator in format itself. So, this is a different streaming operator than what I am using in the Fraction printing.

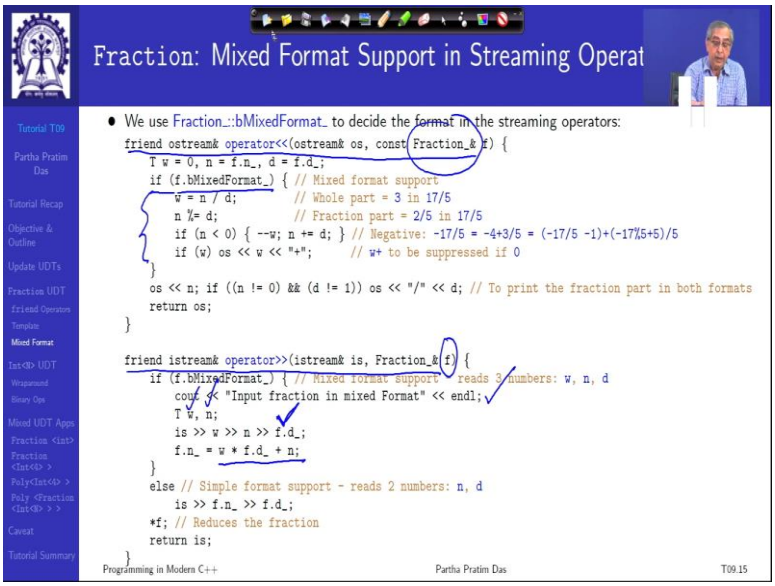
So, this is output-streaming operator for format, where I have to, I know the format object m, which I am streaming, which say is Fraction::mixed. So, I can access the bFormat, this Boolean value, and I have to set it to the bMixedFormat in the Fraction class. Now, this is straightforward. How can this format class access bMixedFormat data member of Fraction because it is a nested class.

So, it can access anything in the enclosing class. So, that is fine. But the problem is the operator streaming is a friend of the format class. It is not a friend of the Fraction class. So, it cannot directly access members of the Fraction class, you know the friend the format class has an access to the members of the Fraction class because it is nested. The operator streaming here has access to the members of the format class because it is a friend. But that is not, there is no transitivity as we had mentioned. So, this will not compile. This will give an error. The fix for that is simple. All that I need to do is to have an again a wrapper for setting this value.

So, if I have a SetMixedFormat function in the format class which actually sets the value to be mixed format, then this wrapper can access. This wrapper is accessing only the Fraction class which it is permitted to do because it is a member of the format which is enclosed within Fraction.

And the streaming operator will simply call this member function which it can do because it is this setting SetMixedFormat function is a member of format class of which the streaming operator is a friend. So, that solves the overall issues and we have something which we can consistently do.

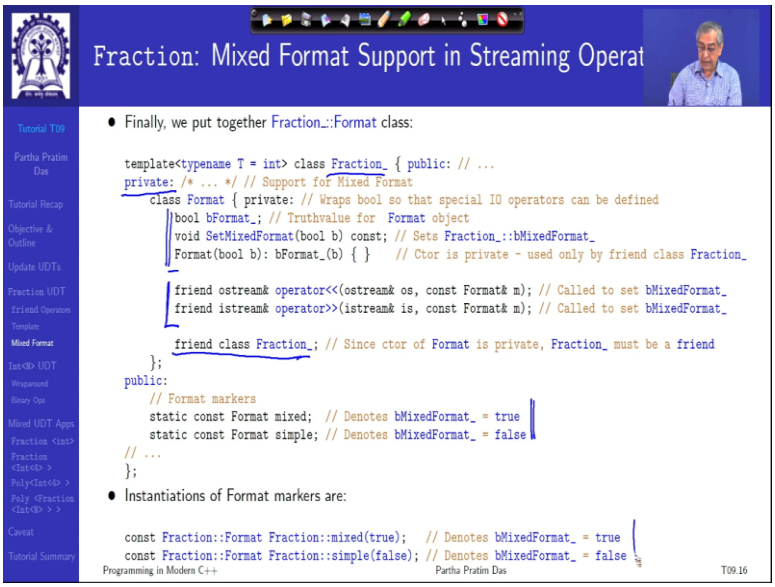
(Refer Slide Time: 25:02)



So, now, we know how to set the flag. Rest of it is very simple. All that we need to change is the streaming operators in the Fraction class. Now, we check the flag for mixed format, if it is mixed format. I express the given Fraction as w+n/d, w may be 0, also if it is a proper Fraction, otherwise, the w will, the value will be set.

And similar thing I can do in the input streaming operator where first I print a message saying that now I am expecting if a Fraction to be given as in the mixed format. So, I expect three integers to be input which are here w, n and since the denominator is not separately changed, so, I read it directly in the d component, d_ component of f and we edit the, we change the numerator part.

(Refer Slide Time: 26:19)

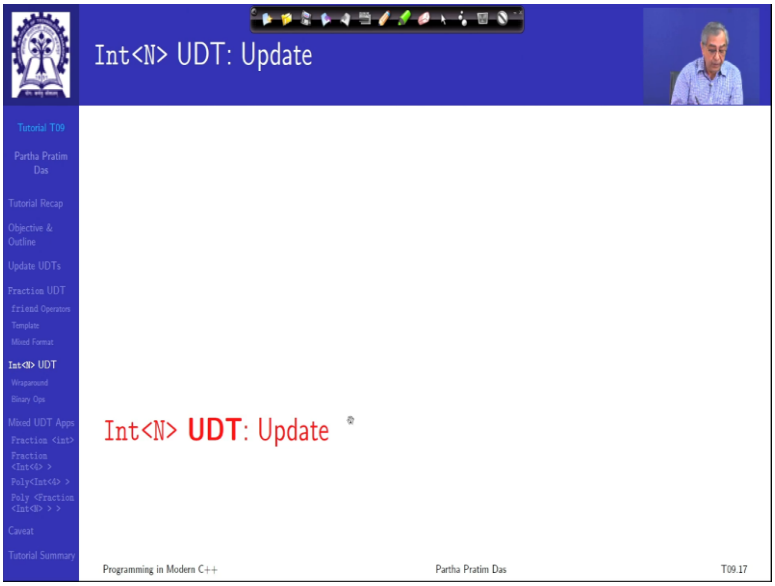


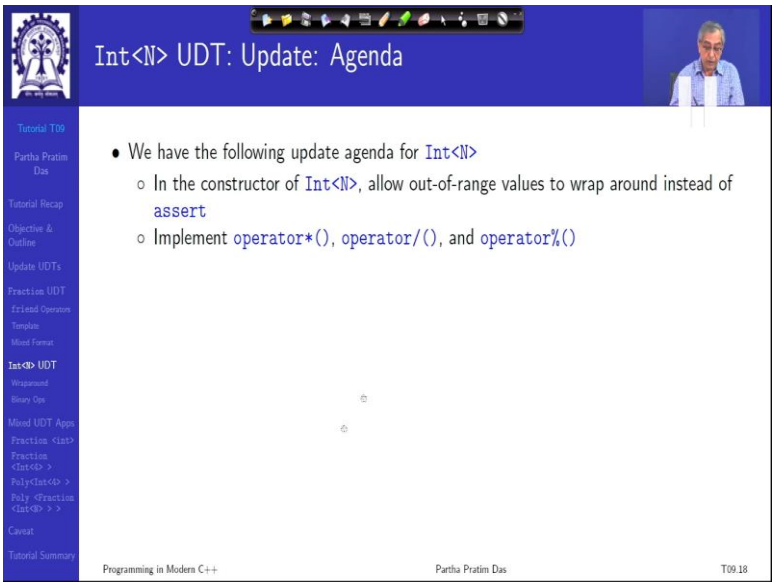
So, this will simply provide the support and with that if we put together the format class, this is how it will look. Format class is a private member or it is nested in the Fraction class, but it is nested as private, so that no one else can access the format class and mess around with it. And all members of the format class I want to be private as well, because is the only the Fraction class which is going to use it.

So, the Fraction class will have to be a friend of the format class. Because the nested class can access everything of the enclosing class but the enclosing class cannot access members of the nested class unless it is a friend. So, I make it a friend and have two operators. So, that I can give the streaming setting of the streaming information to the format.

And according to that I set, also set two public static members of the format information and initialize that in the application code. So, that solves the entire problem and I have the mixed format support. Try it out. Think it carefully. This is a very very interesting support that we have provided which you can provide for other kinds of, where do you have multiple formats to deal with, in all those contexts you can use this design idea.

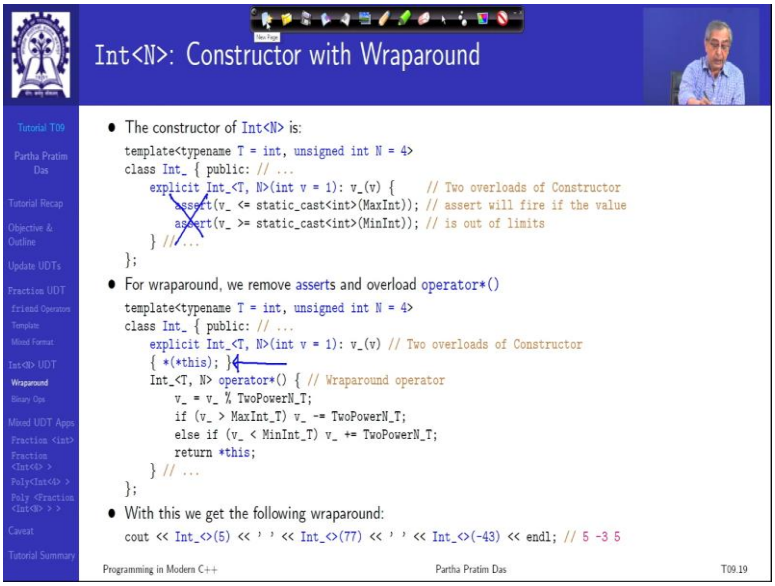
(Refer Slide Time: 27:46)

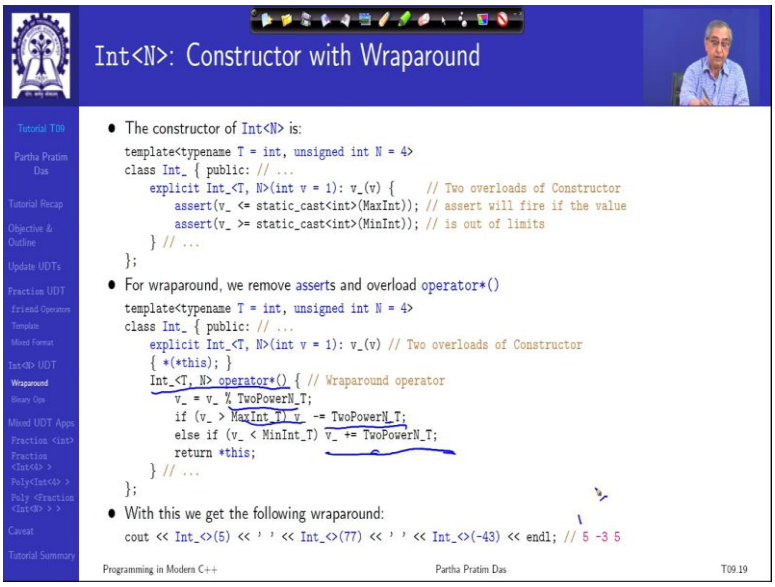




Moving on in terms of Int<N>, we just want to do two things, out of range to be replaced by wrap around and these operators to be implemented.

(Refer Slide Time: 27:56)



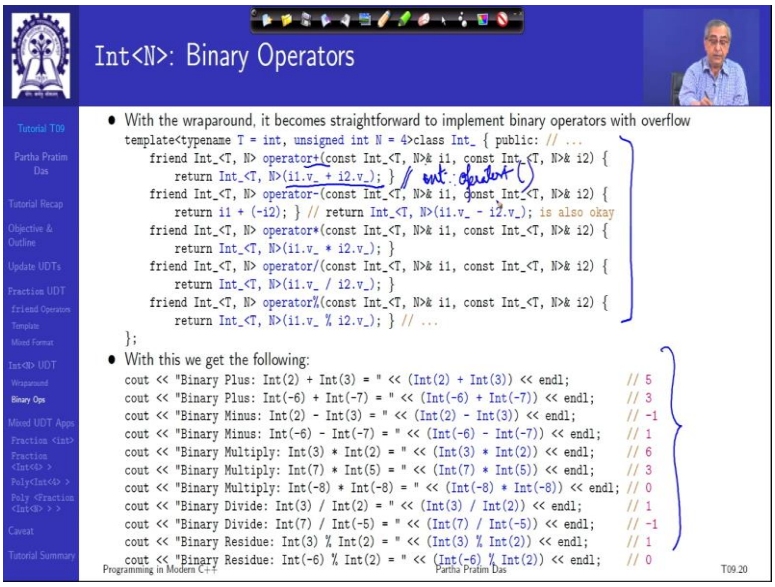


So, doing the first is pretty straightforward. We will have to remove these assets in the constructor and replace it by a kind of wrap around function, like we used operator *, content access operator, the unary operator * in the Fraction class to denote reducing a Fraction, so taking the value properly. Finding out the proper value from the given representation, the kind of an access information.

So, we are using the similar concept here. We can overload the operator * for doing the wrap around. So, now, this operator * gets the current object and all that it has to do is to count, how many times wrapping around up or wrapping around down is required. So, it makes use of you can just look through this arithmetic which is pretty straightforward to see that how many times you are going around the wrapping.

For example, if you are trying to input a value 19 then you have gone around once. So, you will have to basically subtract 16 and say it is 13. But if you have a value, say 77 then you have gone about multiple times. So, 16, 32, 48, 64, then it crosses 80. So, if you subtract 64, it is 13 which is also beyond the maxing. So, now you will have to subtract the range 16. So, it becomes minus 3. So, this wrap around will work and you can try it out. That is pretty straightforward to do.

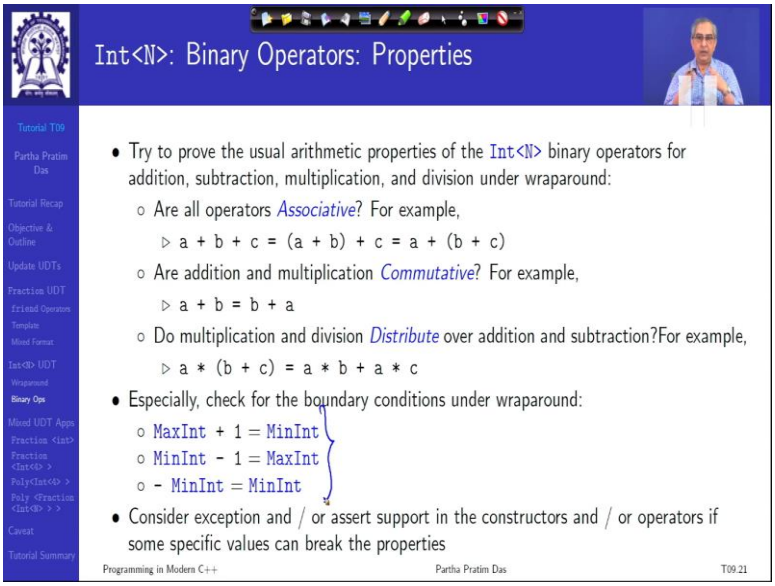
(Refer Slide Time: 29:42)



And with that now, implementing the operators become very simple because all that we do, we take advantage of the fact that Int<N> is actually represented the underlying type is an int. So, all that we do, you take the values that you need to operate. Operate them according to the underlying type. And then you construct the new intent object which will do the appropriate wrap around and bring the values.

You can see several examples worked out here based on the support. So, all that you are doing. So, if you have to do operator plus, you take the component values and add them according to. So, this addition is the addition of, I mean there is no such notation but if I can write it the operator + of Int that is all that you do.

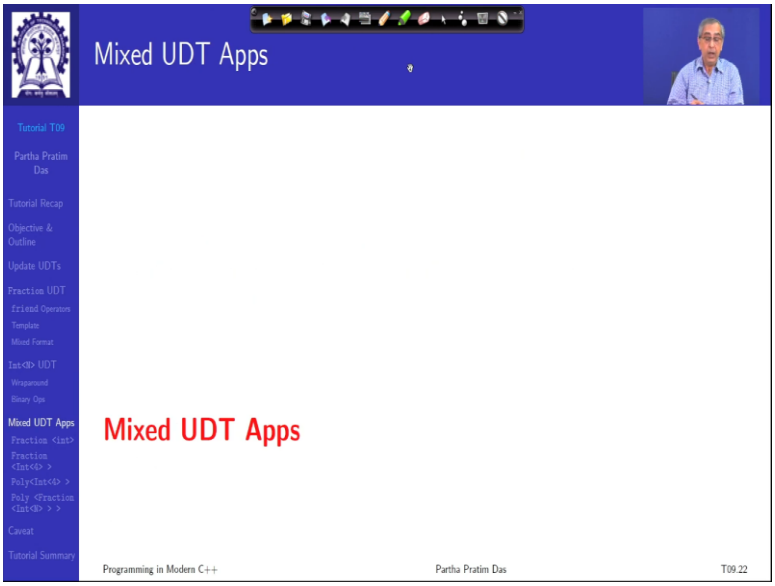
(Refer Slide Time: 30:36)

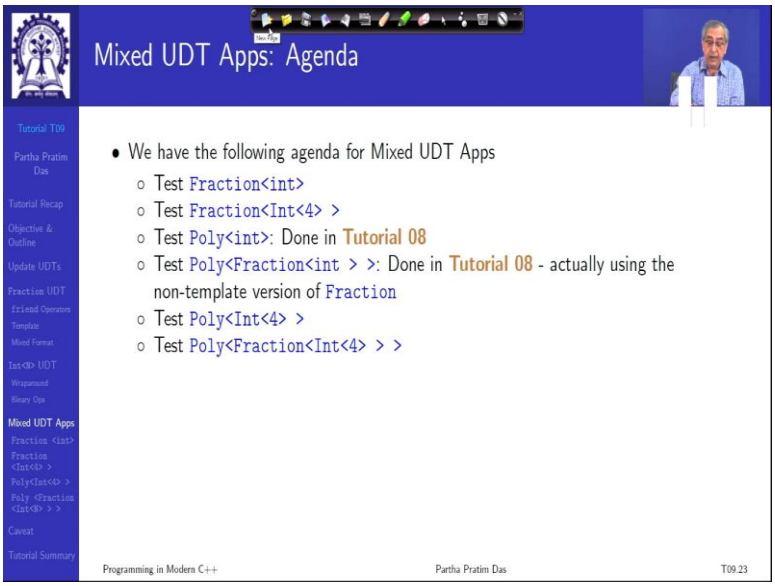


Now, this I have left as an exercise that when you do this then your Int<N> has a set of operators, binary operators with wrap around. So, you expect them to follow the basic rules of arithmetic that int operator also follows, like associativity, commutativity and distributivity.

Precedence, you do not need to worry about. Because precedence is simply a syntactic information and that will always be preserved. So, I would request you to try to prove that if these laws hold and particularly, keep these three boundary conditions in mind that for the wraparound MaxInt + becomes MinInt, MinInt - becomes MaxInt and minus of MinInt is MinInt itself. Because if at those boundary points, the support will fail or the rules will fail then we need to do something like through an exception or assert to let the user know that you are going to get into wrong values now.

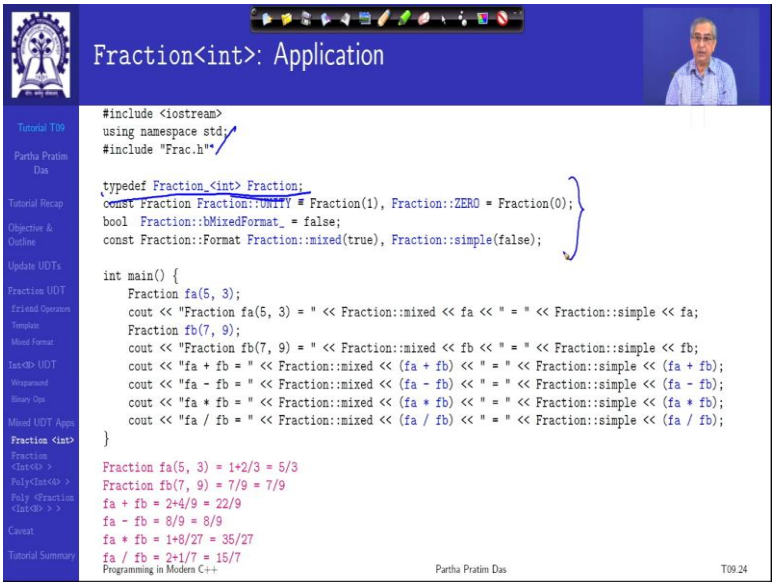
(Refer Slide Time: 31:38)





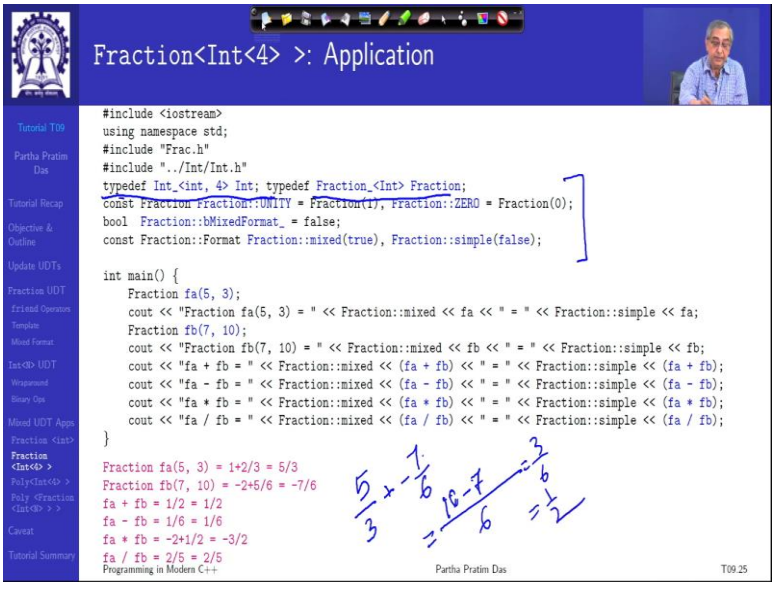
Now, we have updated this. So, we can we can very quickly create lot of mixed applications. Fraction with Int underlying type, Fraction with Int<4> underlying type, Poly with Int type, we had already done Poly with the Fraction type, we had already done that that time Fraction was not kind of templatized. So, we use just the Fraction and it now, it is Fraction<Int>. That also is done in the last tutorial. And we can do Poly with Int 4; we can do Poly for Fraction for Int<4>.

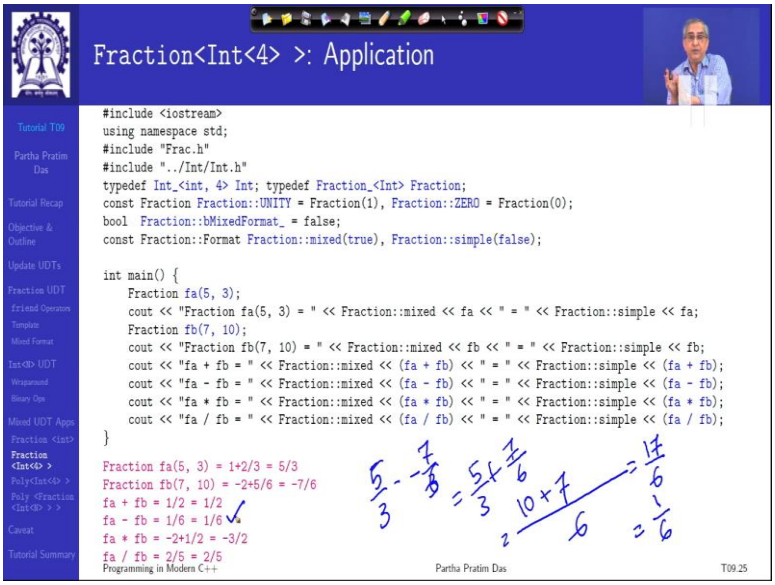
(Refer Slide Time: 32:18)



So, the remaining slides there is nothing much to discuss. Because once we have done this then it will simply, it should simply work. So, say if we have a Fraction for Int then you include the Fraction class and you typedef Fraction_<int> as Fraction and then you do these operations. Everything should happen as is expected. Naturally, all the static members in the class, you have to instantiate for your particular template selection type.

(Refer Slide Time: 32:53)



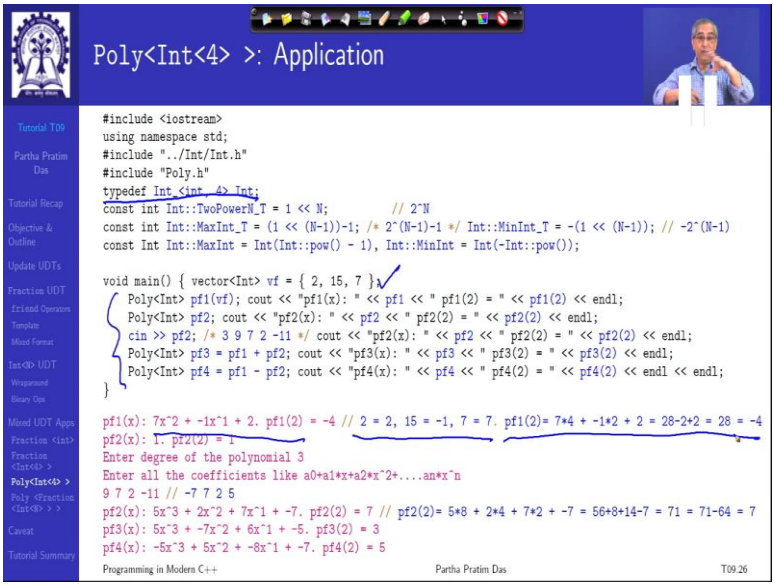


If you use a Fraction for Int<4> then you have to give a typedef for the Int<4> itself. Say that is the Int type otherwise you will have to write lot of code. And using that now, you can instantiate your Fraction class and rest of it the static members and rest of the code remains the same and just the values become different.

Now, you have wraparound in terms of the Fraction operations. You see how nice we implemented that in Int n and trying it out for Int<4>. And we have implemented Fraction separately. But now you are able to add 5/3 + -7/6. And you should be able to, if you have this, you have 6, you have 10 - 7, which is 3/6, which is half, which is easy.

But if you do minus then it is 5/3 - -7/3, 7/6. So, it is 5/3 + 7/6, which is 6, 10+7 is 17/6. Now, 17/6 goes beyond. So, with wrap around this becomes 1/6, 17/6. So, that is why this becomes 1/6. So, you can see the Fraction algebra also gets a new meaning.

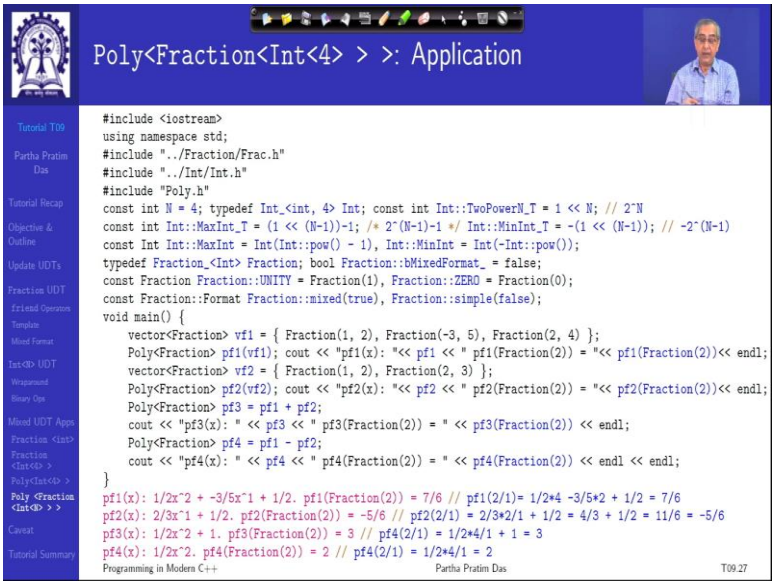
(Refer Slide Time: 34:47)



Obviously, you can have polynomials of Int type. Again, you have to typed up this and you have polynomials exactly as you had before, only thing is instead of the built-in Int type, now we are using user defined Int type. And you can see that again when you evaluate these polynomials, you will have the wrap around effect coming in.

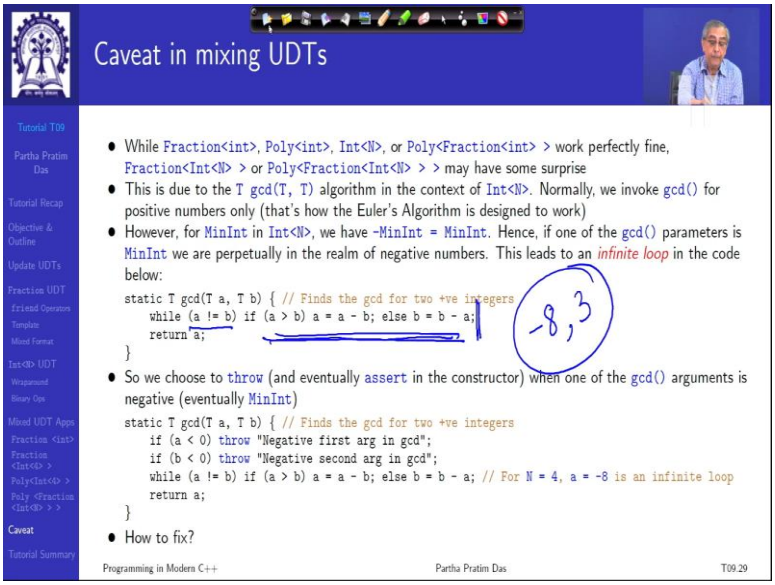
So, for example, if you try the first one, 2, 15 and 7, actual parameters coefficients become 2-1 and 7, because 15 is crossing the limit of Int<4> and therefore with that when you evaluate, you get again the evaluation itself will turn out to be 28 which is beyond the range. So, the result will be -4. So, you are having polynomial with wraparounds.

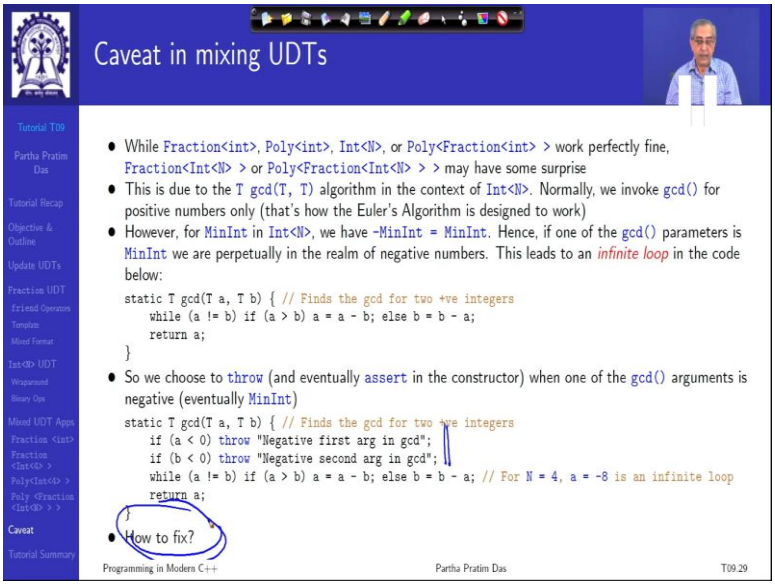
(Refer Slide Time: 35:45)



And finally, you can mix them around. You can have a polynomial on Fractions which underlying type is Int<4>. Just read this code carefully and you will be able to see how different interesting algebra is emerging out.

(Refer Slide Time: 36:03)





Of course, not everything is hunky dory; there are places where things can fail. For example, you have a gcd algorithm in the Fraction. Now, when you invoke that gcd algorithm with Int<4>, the gcd algorithm expects that the numbers, the two numbers given are always positive. But you know that if one of the numbers or both the number, say, one of the numbers is negative then you can call, before calling gcd, you have to make it positive.

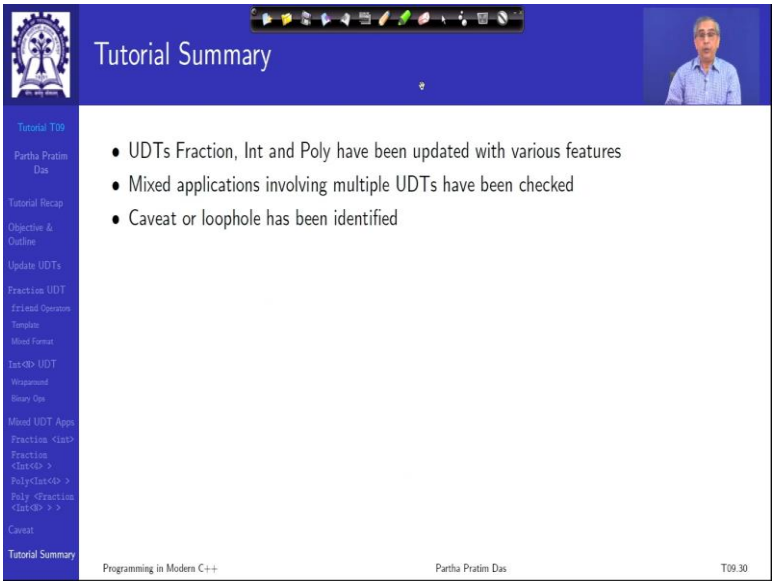
So, instead of -3, you call it with 3, fine. But what happens if a number is MinInt? If you take negative it remains MinInt, it remains negative. So, the gcd's algorithm, which is based on the fact that both parameters are positive, will not work at this line if one of the parameters is MinInt. If both parameters are MinInt, it will work. Because this condition is violated and you get MinInt as gcd which is correct.

But, if one parameter is MinInt say, in Int<4> one parameter is 8 and another parameter is 3 then this actually will never work, because it will continue forever. It will keep on flipping, flipping, flipping, flipping. So, it will not work. So, that is the reason I told you in Int<10> to look at, how the arithmetic operator rules will work, because there are cases where you will have this kind of different behavior coming in. And please try this out for your actual Int; it will have that same behavior for this gcd code.

So, it is not different for them from the built-in type, but what we want that we want to warn the user that this kind of a thing is going to happen. So, maybe we will throw from the gcd, but if we throw from the gcd then it means that the Fraction cannot actually be reduced. So, that object cannot be properly constructed. So, you have to assert from the constructor and so

on. and just think about how to fix this. Is it possible to give a gcd algorithm, specifically for Int? And how to use that? I leave that as an exercise.

(Refer Slide Time: 38:24)



So, to sum up we have given extended on our discussions on the user-defined types with Fraction Int and Poly updated and there are several mixed applications that we have tried out. Just try out these codes. I will make all the codes also available separately as a single project. So, that if you are unable to complete then you can always try out that code and check whether I mean really learn how your implementations is going. And the code that I am going to share is not unique but it is just a representative one. So, you can make changes in that. Thank you very much for your attention and we meet in the next discussion.