Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 16
Operator Overloading (Contd.)

Welcome to module 9 of Programming in C++. We have been discussing operator overloading. We have seen the similarity and differences between operators and functions and we have seen how in C++, a features have been given to define operator functions and overload them, and with that we have taken two examples in the earlier part to overload operator plus for a string type that we have defined and concatenate two strings and we have also overloaded operator plus in a different context for an enum type to show how we can have a closed add operation for enum types.

(Refer Slide Time: 01:04)

Now, we will move forward to take a more detailed look into what can be overloaded and how and what if you are defining your own type, any type then what are the operators you can overload and how you can do that overload? What is the possible? What is not possible? What is advisable and so on?

So, I present here a summary of rules. Let us go through them very carefully. The first naturally; if we have to overload your first question is - what is the operator that can overload? Certainly, a number of operators are given in C++; plus, minus, division, multiplication, assignments, various kinds of extensions of assignment, all of those exist. So, can you define a new operator symbol and overload that? If that is a question if you have then the answer is no; that is you have to restrict yourself only to the set of operators that are defined in the system.

You cannot, for example, you cannot say that I have a overloading for say this operator or I will have this operator. Actually, some of you had been the old programmers in Pascal, you will recognize that this was the inequality in Pascal, but this is not a defined symbol in C++. So, you cannot overload operators with new symbols.

The second point which is very important is when you overload an operator; you cannot change its intrinsic properties. Intrinsic properties will have to remain same and there are 3 intrinsic properties for an operator; that is arity, there is a number of operand it takes, the precedence in with respect to other operators and associativity with respect to its own group of operators or operators of the same precedence. These 3 intrinsic properties cannot be changed.

(Refer Slide Time: 03:08)

So, if you argue that I do have an operator plus, which can be written like this or it can be written like this. I have an operator minus which can be written like this, which can be written like this and so on. So, which means here it is, arity is 2, here the arity is 1, but just such kind of changes you will not be able to do yourself. If for the operator multiple versions of arity exist and correspondingly different precedence and associativity are defined, you will have to go by that, but you cannot for any of the operators define what change its arity precedence and associativity. So, intrinsic properties will have to be totally honored.

The third are a list of; these are the commonly used 38 operators in C++ which can be overloaded. So, you can see that you have almost all the operators that you can think of, including the basic arithmetic operators and a whole lot of assignment operators, then your shift operators, your logical operators, your pointer referencing operator, your carry operator, your function operator and so on. So, all this can be overloaded.

(Refer Slide Time: 04:43)

If you have a unary operator, you know unary operators are of two types, one are prefix operators which happen before the operand, that is this kind of; these are all prefix operator or they can be of the post fix type.

(Refer Slide Time: 04:47)

So, the question is a same operator particularly, if you look at ++, I can write ++a or I can write a++. The question naturally is given that the correspondence between an operator and then operator function, we have said that the operator function corresponding to an operator is just the operator keyword. So, it is operator keyword followed by the operator symbol. So, both of these will necessarily have the same operator function name.

So, your question will be, but that different operators prefix and post fix at different operator pre increment and post increment at different behaviors. So, how do we distinguish that? So, in this two points you will find that answer that if we unary operator is prefix, then like this you simply write it, the signature will look something like this, which takes one the operand because it is unary of your type and it returns the operand that it had taken after the incrementation.

Whereas, if it is post fix operator then the interesting thing is you will have to specify a int as a second parameter and this int actually is not an active parameter. That is when you actually invoke the operator there is no int that you are going to pass it is just there in the signature. So, but this helps the compiler to decide that this operator this instance of the operator function is for the post fix type and not of the prefix kind. So, if I write

a++ then it minds to this. If I write ++a, it minds to this. This is the basic mechanism of resolving prefix and post unary operators for overloading.

Next, please note that there are some operators which are not allowed to be overloaded like scope resolution operator, like member access, for example, if we have a structure it is in complex structure with r e n i m as components. So, I can take the structure name put a dot and put r e. So, that is the r e component of the structure. So, the member access cannot be overloaded. The size of, to find the number of bytes of any variable or type cannot be overloaded; the ternary operator cannot be overloaded and so on.

There are some operators which are allowed to be overloaded like logical and logical or comma etcetera, but you have to keep in mind that if you overload them then their basic properties, some additional properties might get destroyed.

(Refer Slide Time: 07:53)

That is in ampersand 'logical and' if I say a, if I write this expression, then this logical and has a behavior that if this part of the expression gets false then it does not evaluate the second part. If this part of the expression gets false then it does not evaluate the second part, at do you see the correctness of the logic. If a is not equal to b, a and b are different then a equal to b will become false and once this become false, since it is and

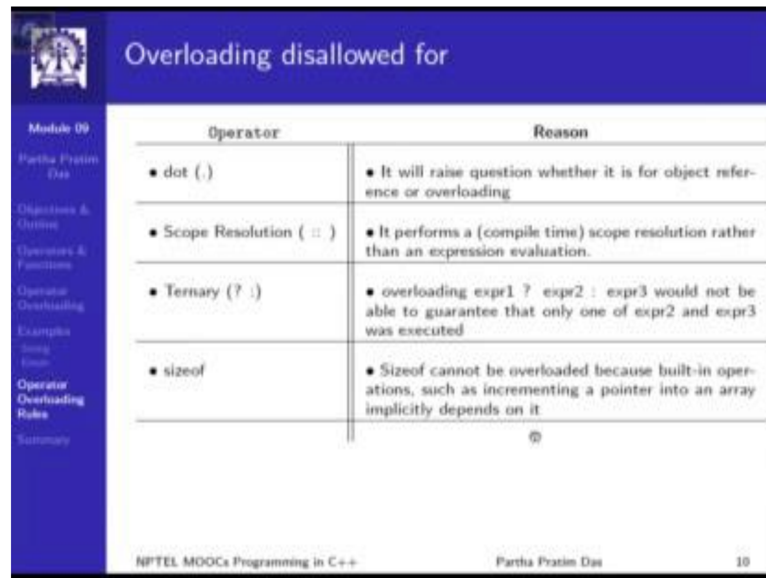
operation no matter what that truth or falsity of b being equal to c whatever whether the second part is true or second part is false, this whole expression is going to be get false anyway.

So, this kind of just evaluating one part and not evaluating the other is known as the short circuit in evaluation, and is done by sequencing because you need to decide in which order you evaluate them. These are special behaviors of this operator. So, if you overload those behaviors will get lost, you will have to be careful that after you overload those behaviors cannot be assumed.

Finally, if you overload the pointer redirection operator, indirection operator then that operator must return in other pointer either it is a directly a pointer. This point just you note, it is not easy to understand at this stage. We will at some point talk about particularly overloading of this operator, which is a very strong feature of C++ programming not though of the language. It is called smart pointers and that use it. So, if you are overloading this operator then you need to necessarily return a pointer or something that can become again a pointer.

These are the basic rules of operator overloading. So, following this you can start writing your operators and I have shown examples as in string and enum that, you really though the rules have to be strictly followed, but they are quite intuitive and straight forward and taking the examples that I have discussed here. You can simply write more and more types of your own, it would be good to write a full complex type where you could actually slowly overload other operators and really make the complex numbers behave as in the integers once.

(Refer Slide Time: 10:36)



In the following slides, which I will not go through in detail in the lectures, I have tried to put down the operators which are not allowed to be overloaded as I had mentioned and I have tried to provide a reasons so that you not only have; we just do not simply need to remember is to why it is not allowed to overload the ternary operator or why is it not allowed to overload the size of operator, you can actually understand the reason and then it will be easier for you to remember.

(Refer Slide Time: 11:12)

Do not overload these operators

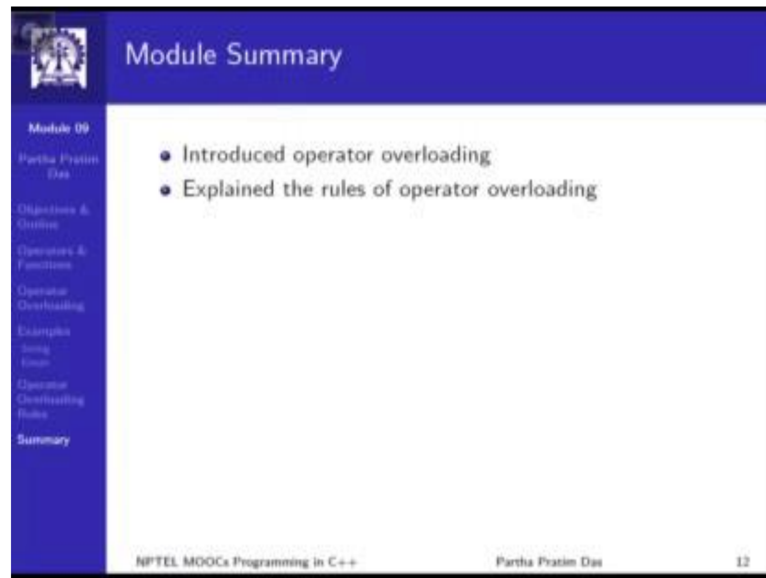
Operator	Reason
• && and ||	• In evaluation, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. However, this evaluation is not possible for overloaded versions of these operators
• Comma (,)	• This operator guarantees that the first operand is evaluated before the second operand. However, if the comma operator is overloaded, its operand evaluation depends on C++'s function parameter mechanism, which does not guarantee the order of evaluation
• Ampersand (&)	• The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator &() as a member function, then the behavior is undefined

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, this one list which is overloading, where overloading is disallowed and another list which I briefly discussed is where there are operators, where overloading is actually allowed, but it is advised that you do not overload them because if you overload them then some of the very basic behaviors of C++ program changes. So, these are to be overloaded by really expert people and till you get to that level, I would advice that you please do not overload that.

The remaining 38 operators that you have at hand, is rich enough to develop any kind of algebra, with any kind of matrix complex fraction any kind of types that you want to.

(Refer Slide Time: 11:56)



With this we come to the close of this module where, here we have introduced the operator overloading and we have explained different rules and exceptions for operator overloading.

And in the next module, in module 10, we will show another special case of extension and special case of operator overloading in terms of dynamic memory management. We will introduce the operators that C++ provide for dynamic memory management and we will again show that how operator overloading can be applied in the context of dynamic memory management operators to get, see various kinds of strong advantages in memory management in C++.