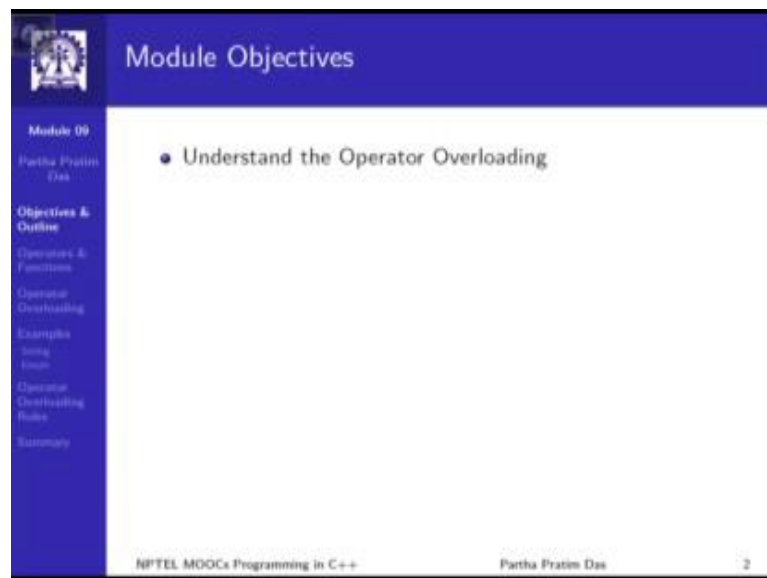


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 15
Operator Overloading

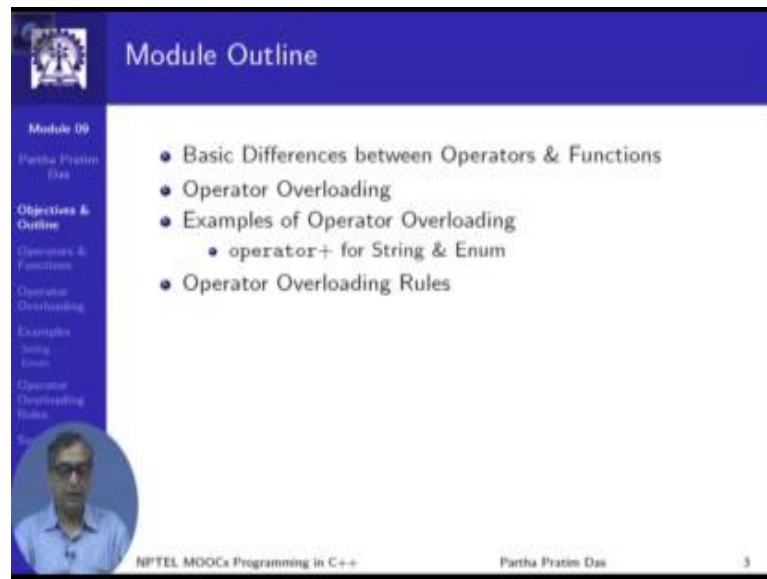
Welcome to module 9 of programming in C++. We are continuing the discussions on the procedural extensions of C for better C in C++. We have already discussed a couple of features like const, inline function, reference and particularly function overloading and default parameters.

(Refer Slide Time: 01:13)



We will extend on the discussions on function overloading into a new topic, operator overloading. We will see, this is a new concept in C++, where behaviors of new behaviors of operators can be defined, and we will take a look how that can be done in C++.

(Refer Slide Time: 01:19)



The slide is titled "Module Outline" and is part of a presentation. It features a blue header with the title and a vertical sidebar on the left containing a navigation menu. The main content area lists the topics to be covered in the module. At the bottom, there is a small circular portrait of the speaker and footer information.

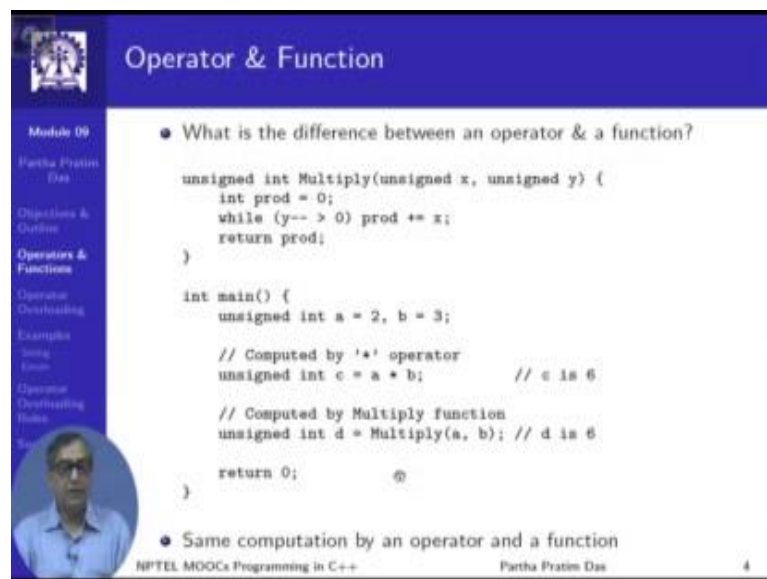
Module Outline

- Basic Differences between Operators & Functions
- Operator Overloading
- Examples of Operator Overloading
 - operator+ for String & Enum
- Operator Overloading Rules

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, the objective of this module would be to understand operator overloading. This is how we will do this, we will try to understand operators and functions better, and then define operator overloading take examples and set down the rules for it.

(Refer Slide Time: 01:35)



The slide is titled "Operator & Function" and is part of a presentation. It features a blue header with the title and a vertical sidebar on the left containing a navigation menu. The main content area contains a bullet point question, a C++ code snippet, and another bullet point. At the bottom, there is a small circular portrait of the speaker and footer information.

Operator & Function

- What is the difference between an operator & a function?

```
unsigned int Multiply(unsigned x, unsigned y) {
    int prod = 0;
    while (y-- > 0) prod += x;
    return prod;
}

int main() {
    unsigned int a = 2, b = 3;

    // Computed by '*' operator
    unsigned int c = a * b; // c is 6

    // Computed by Multiply function
    unsigned int d = Multiply(a, b); // d is 6

    return 0;    @
}
```

- Same computation by an operator and a function

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

We start with asking a question, the question is we know in the programming language we often use operators, we use so if we just look into this code, there is a there is an operator here; a and b are two integer variables, we use the operator multiply which in this case 'a' is a binary operator, which means it takes two operands, these operands are, a and b. They will operate on this operand and will give us a result. Similarly, if you look at here, I have function by the name multiply, which also takes two arguments, two parameters a, and b; and using this algorithm, since this is a function I have to specify the definition of the function.

I have specified the algorithm by this algorithm it does multiply the first operand and second operand together and gives us a result. So, if you ask the broad question, the abstract question as to what is the difference between an operator and a function, you will find some very significant similarity, because both the operator as well as the function takes some parameters and compute the result. Multiply operator, in this case, takes two parameters, two arguments or often we said two operands and produce as a result.

Similarly, here I have a multiply function, which takes two operands or two parameters, in case of function we often say it is arguments or parameters and returns a result value. So, there is we see that there is a significant similarity if we look at the operators and function from the computational point of view. But still we choose to call some of the cases as operator a plus b, which plus is an operator a into b, star is an operator minus a, we say negation is an operator; whereas square root, sort, multiply function, all these we say are functions. So, our basic question we asking is what is the difference between an operator and function.

(Refer Slide Time: 04:14)

Operator	Function
<ul style="list-style-type: none">• Usually written in infix notation• Examples: Infix: <code>a + b</code>; <code>a ? b : c</code>; Prefix: <code>++a</code>; Postfix: <code>a++</code>;• Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)• Produces one result• Order of operations is decided by precedence and associativity• Operators are pre-defined	<ul style="list-style-type: none">• Always written in prefix notation• Examples: Prefix: <code>max(a, b)</code>; <code>qsort(int[], int, int, void (*)(void*, void*))</code>;• Operates on zero or more arguments• Produces up to one result• Order of application is decided by depth of nesting• Functions can be defined as needed

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

So, here we chart out the various similarities and differences between an operator and a function. The notable of this is a very first point, you see an operator is always written in the terms of an infix notation, this is how an operator is disturbing. So, the operator symbols it is in the middle and the two operands this binary operator, so it takes two operands the two operands sits on its two sides. Consider the case of this operator, these are written little bit differently in some cases the operator precedes the operand; we say this is a prefix case. In some cases, the operator follows the operand, we say this is a postfix case, but often we will write things like `a into b`, we will write `a divided by b`, `a minus b`, `a ampersand b`, and so on and so forth, minus `a` all different once. But all of these are necessarily in the infix notation.

Now, if you look into the function, you all know, what is a function suppose this is a function or I have another I write in another function `add a b` will always; see that the function name occurs before the list of parameters that the function takes. So, a function is always written in the prefix notation. So, primarily what do we say is an operator, what do we say is a function has to be primarily first decided by the notation that we use for it and computationally, they are interchangeable; computationally where we have an operator maybe I could have had a function or where we have a function I may be able to have an operator and so on. But notation wise it is very different function is necessarily in prefix

notation, operators are usually in infix notation. But some of the operators in the infix also could be a prefix position or post fix positions.

Then the following points tell us some of the more commonly observed differences operators operate on one, two or three operands. We know in C, we have unary, binary and ternary operators. Operators having more than three operands are not usual, whereas a function can have any number of arguments. In fact, a function may not have an argument also, where it is never possible that I have an operator which does not have operate and a function can have 3, 4, 5, 10 any number of arguments. Coming to the third point, an operator will always produce a result, it is a part of an expression that will always produce a result where is the function usually produces a result, but we you already know that I can have a function with wide return type which may not produce a result.

(Refer Slide Time: 07:36)

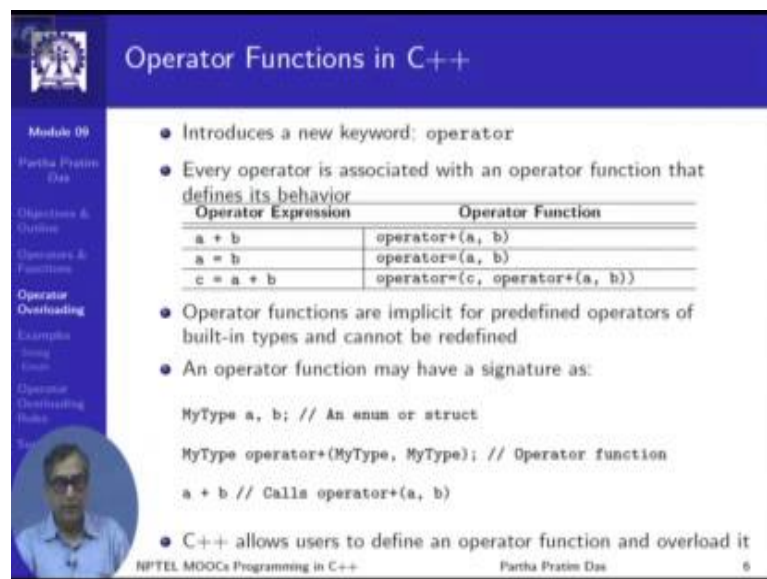
Operator	Function
<ul style="list-style-type: none">• Usually written in infix notation• Examples: Infix: $a + b$; $a * b : c$; Prefix: $++a$; Postfix: $a++$;• Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)• Produces one result• Order of operations is decided by precedence and associativity• Operators are pre-defined <p>$a+b*c$</p>	<ul style="list-style-type: none">• Always written in prefix notation• Examples: Prefix: $\max(a, b)$; $qsort(int[], int, int, void (*)(void*, void*))$;• Operates on zero or more arguments• Produces up to one result• Order of application is decided by depth of nesting• Functions can be defined as needed <p>$f(a, g(b, c), d)$</p>

If I have an expression, in which say I have an expression like a plus b star c. If we have an expression, we know that the order in which the different operators in this expression are evaluated or applied depends on the precedence and associativity of the operators. Whereas in case of function, it necessarily depends on if I have a function say like this if I have function called like this. I will necessarily know that, this function call for g which

takes b and c and computes a result has to precede the function called to f. So, the order of application of function is dependent is decided only by the depth of nesting all parameters have to get evaluated before the function can be applied.

Finally, operators are necessarily predefined, functions can be defined as and when need it. So, these are some of the major differences between the operator and the function, but it is it can be noted that for limited number of operands. And when always there is a result, it may be very convenient to be able to express computations in terms of operators, because as we say the operators allow us write algebra. For anything that we want to do for int, we can write a big expression for double, we can write such expressions, and we can write similar expressions for a say point at types and so on. It would be good if we could do similar things for other kinds of types as well.

(Refer Slide Time: 09:30)



Operator Functions in C++

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
<code>a + b</code>	<code>operator+(a, b)</code>
<code>a = b</code>	<code>operator=(a, b)</code>
<code>c = a + b</code>	<code>operator=(c, operator+(a, b))</code>

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct  
MyType operator+(MyType, MyType); // Operator function  
a + b // Calls operator+(a, b)
```
- C++ allows users to define an operator function and overload it

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Next what we look at and what C++ introduce us is the notion of operator functions. That is if I have an operator then, so far the operators, that we have seen like operator plus for int or operator plus for double or operator plus multiplication for int. The behavior of this operators, that is a algorithm that the operator follows and the kind of result that the operator returns; gives us is pre decided by the compiler. Now, we are saying that to we want to do something different for an operator we will associate a function with that

operator and to do that association C++ introduces a new keyword called operator, this is how this is spelt it is a new keyword. It is the reserved word as well and with this, we make some notational equivalence between the infix notation of the operator and the corresponding operator function that may be thought of.

So, if we have an expression a plus b which is very common as you will see; I can think of this corresponding to operator plus, I can think of that there is a function whose name is operator the plus symbol. So, given an operator the corresponding operator function gets a name which is a operator key word followed by the operator symbol. So, a plus b, the plus operator gets the name operator plus, there are two operands here a, and b naturally they become the two arguments a, and b in that same order. If you look at a at a different case, let say a is a signed b you would recall that in C assignment is actually an operation, is an operator a assigned b is actually an expression, which has a value which is a value assigned to the left hand side.

So, if assignment is an operator then I can for this assignment symbol, I can write a corresponding operator function operator assignment and pass a and b as two parameters, whenever we write operator function. It is very important to remember that the order of the operands, if there are more than one operands, the order of the operands cannot be changed. The first operand has to be become the first argument in the operator function and so on. So, in the same way this is just the third example is just to elaborate little bit more, that if I have c assigned a plus b then, we know that plus has a higher precedence who are assignment; so first there addition has to happen and then this added value will be assigned. So, in terms of operator function notation, this is what operator plus a b is what adds and by next in this will necessarily happen earlier, we have already discussed in terms of functions the order is decided by nesting. So, this is more nested. So, this will happen first and the result that it produces will be operated with operator assignment that is it will be assigned to c.

So, corresponding to any operator based expression that we have, we can write equivalent expressions using operator function notation. Please note at this point, that the operator functions are implicit for all predefined operators of built in types. That is all the examples, that I am showing above in that a, and b these operands are not cannot be

assumed to be of int or double or float type, because for them the operative function notation has not been allowed. Because we do not expect that you would like to define a new kind of addition for integers. So, to come to how you do it, how you take a look at the operator function, say my type is some type which I am just assuming that it could be an enum type or a struct type as I said it cannot be a built-in type. So, it is some type that, I am defining and will see more lot more of these as we actually going C++. We actually have class definitions coming in. So, my type is some type defined a, and b are two variables.

So, for in that context, say I can define an operator plus for my type, which takes two my type values, because we know operator plus is a binary operator. So, it needs two values, so it has two arguments and it returns a my type value as a result, and this what we says the operator function for operator plus that we can define; an if this has been defined the interesting thing is then, for this my type value I can write a plus b as an expression. If I write a plus b, then this expression actually calls the operator plus a b that is the operator function that have written for operator plus for my particular class. This is the very, very interesting part and this is what we will particularly focus on here.

(Refer Slide Time: 15:30)

Program 09.01: String Concatenation

Concatenation by string functions	Concatenation operator
<pre>#include <iostream> #include <cstring> using namespace std; typedef struct _String { char *str; } String; int main() { String fName, lName, name; fName.str = strdup("Partha "); lName.str = strdup("Das"); name.str = (char *) malloc(strlen(fName.str) + strlen(lName.str) + 1); strcpy(name.str, fName.str); strcat(name.str, lName.str); cout << "First Name: " << fName.str << endl; cout << "Last Name: " << lName.str << endl; cout << "Full Name: " << name.str << endl; return 0; } ----- First Name: Partha Last Name: Das Full Name: Partha Das</pre>	<pre>#include <iostream> #include <cstring> using namespace std; typedef struct _String { char *str; } String; String operator+(const String& s1, const String& s2) { String s; s.str = (char *) malloc(strlen(s1.str) + strlen(s2.str) + 1); strcpy(s.str, s1.str); strcat(s.str, s2.str); return s; } int main() { String fName, lName, name; fName.str = strdup("Partha "); lName.str = strdup("Das"); name = fName + lName; // Overload operator + cout << "First Name: " << fName.str << endl; cout << "Last Name: " << lName.str << endl; cout << "Full Name: " << name.str << endl; return 0; } ----- First Name: Partha Last Name: Das Full Name: Partha Das</pre>

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, in exciting part is C++ now allows us to allow the user to define an operator function as I have shown you and to overload it; that is I can write my own algorithm for this operator function based on my type. Let me take an example, the example looks little long do not worry this is a simple example the objective here is to show that really if I use the operator overloading how am I expected to benefit from that. So, this example shows the code for concatenating two strings, you know concatenation of strings, if we have two strings we want to put the second string after the first string put them together we say this is a concatenation. So, on the left, if you look into the left column, this is what we assume is the type of the string, this is just something some type I am defining which is nothing, but a pointer to character. So obviously, the string will actually have to be dynamically allocated.

Now, my objective is to take two names of a person f name standing for the first name l name standing for the last name and to compose the total name by concatenating the l name after f name. So, just to keep things simple I show a hard coded example. So, f name dot str, which is the string component in the structure that have defined for the type I duplicate a constant string by a str dup. Similarly, for l name dot s t r in that other string I duplicate another thus the last name part of the intended name. Now, if I have to put the l name after the f name, then what I need to do, I need to concatenate these two strings in terms of a new string. So, the first thing that I will require is a new string name, which is went with the concatenation must have enough space allocated.

So, this is the typical code, you will do to allocate, you will find out the length of the f name, find out the length of the l name add one to that length,, because you need the extra space for the null and malloc that. I did not multiply this by size of, because I am assuming the size of the character is one. So, malloc and cast two character star cast star. So, you have enough space in the name and then you do this these two typical steps, that is you copy the first name to name dot str and then copy l name dot str; then concatenate I am sorry not copy concatenate str cat, the l name dot s t r to the name dot s t r. So, name dot str already had the f name; and at the end of it, the l name dot str will get concatenated. And if you output you will get the concatenation happening which is here. So, this is the traditional way of doing it.

Now, look at how we want to do this in C++ using the operator function or the overloading of the operator function. We say that conceptually concatenation can be thought of as an addition of two strings; that is it is putting one string after the another and therefore, if I can take two strings and be able to write it as string one plus string two is my result, then I it would be minimize expression wise it will very convenient. So, look at the main function in the right hand side column, we have the same set of variables, we have the same initialization here that is f name dot str and l name dot str are duped from constant strings. But then to concatenate we just express that, we should be able to write f name plus l name and that result should be the concatenated name together.

Just to show you to do this whole thing here, I need to write this code which is this part of the code is, what this addition is. Certainly this on the left hand side, the code, I had to write is not only is longer, but has to be written carefully taking care of the stl. And remembering that the first string has to do strcpy why the second has to do strcat here just conceptually you just add. Now, how does it work to make this work, what I need to do is I need to define an overload of operator plus for the string type that have defined. So, I define it on the top it takes two strings such certainly I do not want these to be tempered in anyway. So, I pass them as constant reference and within the function within this operator function that have implemented I allocate enough memory I copy the first string, concatenate the second; this basically is the code that I was writing here actually goes here.

But the advantage is now this code has been written only once. Now no matter how many times I want to concatenate strings, I do not need to write this code over and over again, where as in case of the style show in the left hand side, if I have now two other strings different strings I need to do the tried this code again. So, that increases the error that increases, reduces the readability of this. Here after this has been done if I write it with operand notation of plus then, what it means is the compiler finds out that for the operator plus is there any operator function which takes on the left and right two string type variables, because f name is of string type l name is of string type. It finds do I have an operator plus, where the left hand side is string type f name, right hand side is also

string type which is l name and if it exists then it will simply call this function passing f name as s one and l name as s 2.

Then this function computes the value and returns the result which is copied to the name. So, this is this gives us the huge advantage in terms of being able to write very compact algebraic kind of expressions by defining our own operators for our own type and the that the length that is can go is huge. For example, we know I mean we often do matrix algebra, we add matrices, we multiply matrices, we invert matrices, and we take difference of matrices and so on. Now there is no the matrix type is just an array it is a two-dimensional array. So, in if I have to do it in C I have to write all different functions and a simple addition of two matrices will turn out be a major function call and all those details will come in. Whereas, if I could overload in the way we have show operators like plus for adding two matrices operated like binary multiplication star to multiply two matrices and so on, and the whole life of the programmer will get much easier, so that is a basic objective.

(Refer Slide Time: 23:22)

Page 23/23

Program 09.02: A new semantics for operator+

w/o Overloading +	Overloading operator +
<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; E operator+(const E& a, const E& b) { unsigned int u1a = a, u1b = b; unsigned int t = (u1a + u1b) % 3; return (E) t; } int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; return 0; }</pre>
<ul style="list-style-type: none">• Implicitly converts enum E values to int• Adds by operators of int• Result is outside enum E range	<ul style="list-style-type: none">• operator + is overloaded for enum E• Result is a valid enum E value

I will just for interest show another simple interesting example, where I show how certainly you can change the semantics of these operators in certain context. On the left if you where focus, the basic type that I am going to use here is an enum type e, if that is

an enumeration which basically the sub-range of selective sub-range of integer type. Where I define three enumerated constants is c 0, c 1 and c 2 with the constant value 0, 1 and 2. So, in this enum world there are only three symbols. So, it is a domain of three symbols three values, then here I define two variables a and b with c 1 and c 2, that is they will actually have values 1 and 2 and then i add a and a with b and put that value to x.

Now what will happen, if I add a and b enum as you know is the sub range of int, so if I try to add one enum with another naturally there is no separate addition of operation for enum. So, what it does it simply implicitly converts the enum to int and adds by the operator of int. So, what will happen this value is c 1, a is c 1 which is 1, b is c 2 which is 2. So, you add these two the result is 1 plus 2 3 and you assign that to x and you see that you have output a value three.

Now, while a according to the C program this is a correct thing do, but we feel somewhat uncomfortable because we wanted to remain in the domain of the enum that we have defined. By computing an addition of c 1 and c 2 possibly what we wanted to do is to wrap around because you go beyond the three constant 0, 1 and 2 that you had and we probably would have like to wrap around. So, that if you are adding 1 to 2 it should actually become 0 that is it should on increment come here and keep on going like that, but because the addition has taken me to integer I have a result three, which is not even there is known corresponding enum constant for that in my type definition. So, I am not happy with this. So, want to define my own addition, where if I add two enums in my from, my domain that is c 0, c 1 and c 2 this is again the same enum type, what I want is the result will always be within this enum type.

So, naturally if you add 0 with 0, it is within, if you add 1 with 1, it is 2, it is within, 0 plus 1, 1 plus 0, all these are within. But if you add 1 with 2, it should become 0 that is it should wrap around and come to 0. Similarly, if you add 2 with 2, it should become 1; if I add 2 plus 2 then it should become 1, so that is a kind of definition that I am looking at. So, it is a wrap around that I want. So, this is this is all the same this code is exactly the same the main routine has not been changed. Only thing that I do in between this, I introduce a operator plus function for this enum type; which now takes two different

enum parameters, which have basically these two parameters in vocation of addition and returns an enum. And what do I do, I do a very simple trick instead of just adding, which is the behavior that we were getting here, instead of just adding, we add and then modulo 3, 3 is a number of symbols you have.

So, what will happen if you add 1 with 2 this addition is 3. So, if when you do modulo with three then the result will turn out to be 0 that is it will wrap around, which is a very common technique you know that the remainder function, your remainder operator, the modulo operator always gives you to get that wrap around. So, we just use the trick and then take again take that value and put it back as at the return statement to be given back here.

So, now if I do a plus b the compiler makes out that a is of type e, b is also of type e. So, it tries to find out an operator plus function which takes an e and then e and that is what it finds here. So, on this a plus b it no more converts a, and b to integer and performs an integer addition rather it invokes his function and performs this operation, which truly preserves the closeness of my enum domain and gives me a value, which is 0 which is what I wanted. So, the operator plus is thus overloaded in enum and allows me to restrict the value within my enum range.

So, you can see that we have just shown two examples of somewhat different kinds, where in both cases, the operator plus have been overloaded in string it allows me to do a replace complex string functions by just using a plus operation to concatenate. And here for enum, I can give a special semantics I can give a new semantics for my type, while the underline type, you may continuous to remain to be unbuilt in integer.