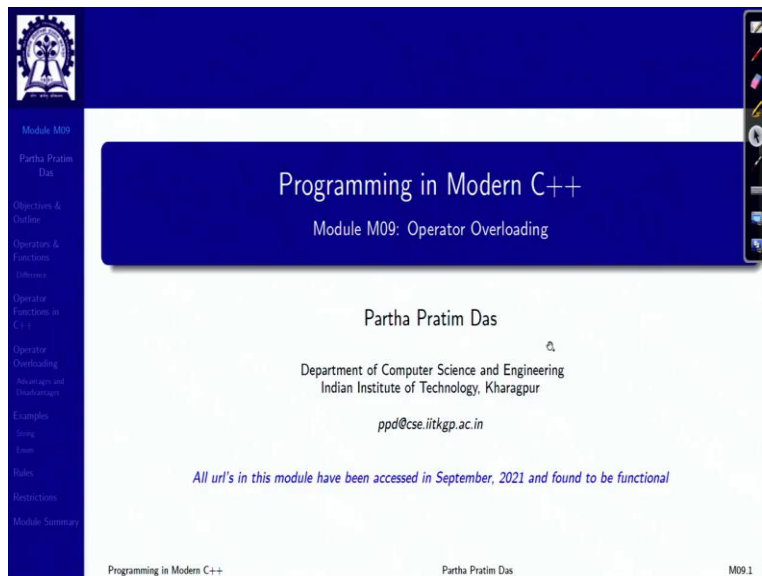
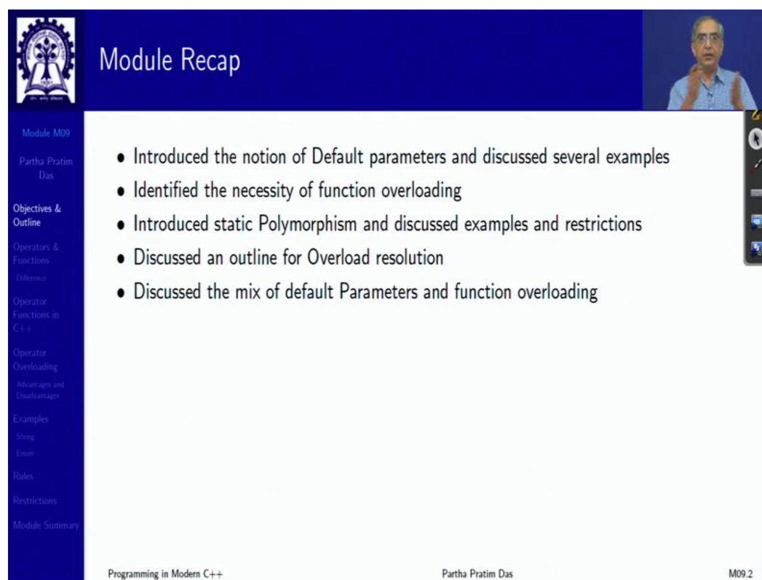
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 9
Operator Overloading

Welcome to programming in modern C++. We are in week 2, and we are going to discuss module 9.

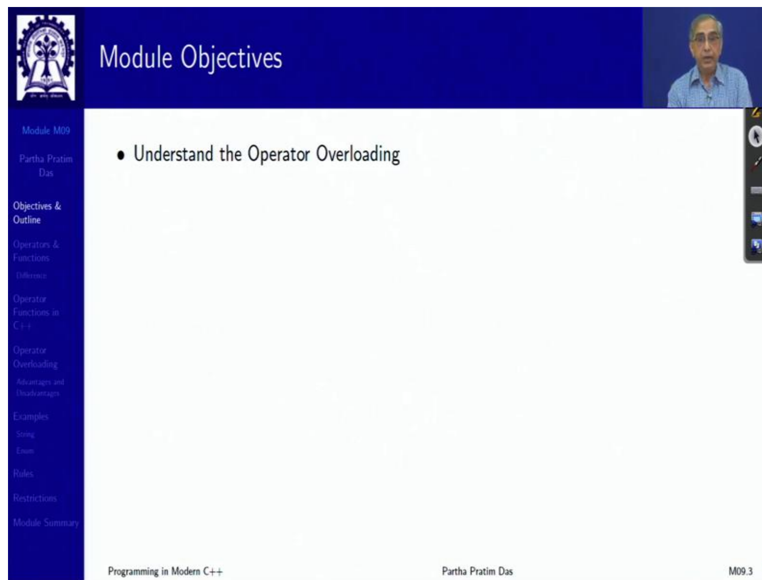
(Refer Slide Time: 0:33)





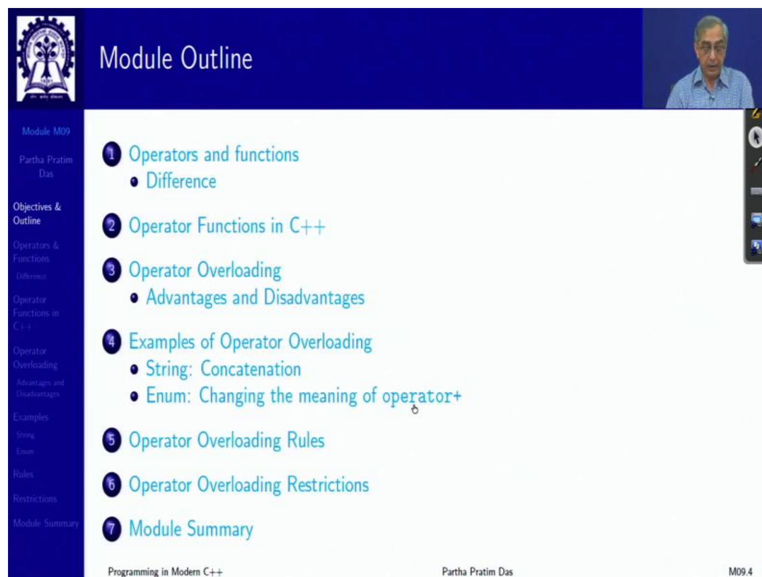
In the last module, we talked about static polymorphism, particularly proliferating into default parameters, and function overloading, and how to resolve that overloads, and how do the default parameters, and function overloading interplay between them.

(Refer Slide Time: 0:56)



We will now extend that concept of overloading into what is known as operator overloading, which will again be a new concept for you all.

(Refer Slide Time: 1:08)



Operators and functions

Module M09

Partha Pratim Das

Objectives & Outline

Operators & Functions

Operator Functions in C++

Operator Overloading

Examples

Rules

Restrictions

Module Summary

Operators and functions

Programming in Modern C++ Partha Pratim Das M09.5

So, this is the overall outline, you will find them on the left. So, let us first, let me first ask you a question, that all of you know operators all of you know functions, what does operators do? Operators take operands, compute something and give you a value on the expression.

What does functions do? They also take parameters, do a computation and give you a value. So, if both of them do the same thing, both of them are expressions, they take 1, 2, 3 different number of parameters, computes and gives you a value, unique value. Then, why do we differentiate between operators and functions? What is the differentiator?

(Refer Slide Time: 1:49)

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

Programming in Modern C++ Partha Pratim Das M09.6

So, here you can see an example, this is a function which in a very, rudimentary way is trying to multiply x with y, by adding x to the product y number of times, which is understandable. Now, this is multiplying by a function, and you know that there is an operator multiplication which

does it by this. So, we have this operator, here we have the function, both of them do the same thing, they will give you the same result. One you call the operator, the other you call the function. So, what is the basic difference?

(Refer Slide Time: 2:36)

I do not know if you have ever come across this question, but this is a profound question to answer. You can wait at this point before you proceed to the video further and try to think of the answer, but you will get the answer here. So, an operator usually is written in infix notation, infix notation means the operator sits in between the operands. Infix a + b, a is here, b is here, + is in the middle infix.

But at times it can be prefix also, that is I can husk first at the operator, then the operand, or it can be postfix also, I can first have the operand then the operator, all three forms in operators are possible, though infix notation is what is most commonly used. So, this is an infix notation, this is an infix notation. Whereas, ++a is prefix notation, - a is prefix notation, + b is prefix notation, &b is prefix notation and so on.

Similarly, you have a++, which is a postfix notation. So, in operators C, C++ operators typically you have up to three operands, and there is one ternary operator otherwise everything is one or two. It always produces a typical one result. That is what you have. Now, if you look at function, the main factor is that a function is always in the prefix notation. You first see what is the name of the function, like you are doing here.

You first say what is the name of the function, and then you provide the operands in the order. That is a fundamental difference between what is an operator, and what is a function in terms of the mathematical notions. Now, in functions we also have some more differences with the operator for example function may not take an argument, or it may take more than three arguments also it can take any number of arguments.

It produces up to one result, and operate will always produce one result. But a function may produce one result may not produce a result also. It is a return void. In operator the order of applications when I write an expression with the operator, the order of application of the operators are based on precedence and associativity, we all know that BODMAS rule and so on. But in function, it is the depth of nesting.

The innermost function is called first, then the next layer, then the next layer. So, it is the depth of nesting of call, which decides the order in which you evaluate. And finally, operators are predefined. Whereas, functions can be defined as we wish. So, these are the basic differences and it will soon become clear as to why we are talking about such a topic.

(Refer Slide Time: 5:56)

Operator Functions in C++

Operator Functions in C++

Programming in Modern C++ Partha Pratim Das M09.8

Operator Functions in C++

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

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- 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

Programming in Modern C++ Partha Pratim Das M09.9

Operator Functions in C++

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

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- 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

Programming in Modern C++ Partha Pratim Das M09.9

Operator Functions in C++

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

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- 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

Programming in Modern C++ Partha Pratim Das M09.9

So, to introducing to that, I talk about operator functions in C++. What is an operator function? So, you have operators which are predefined, you have functions, which you can write; that is a C scenario. In C++, in terms of syntax, first a new keyword is introduced is called operator.

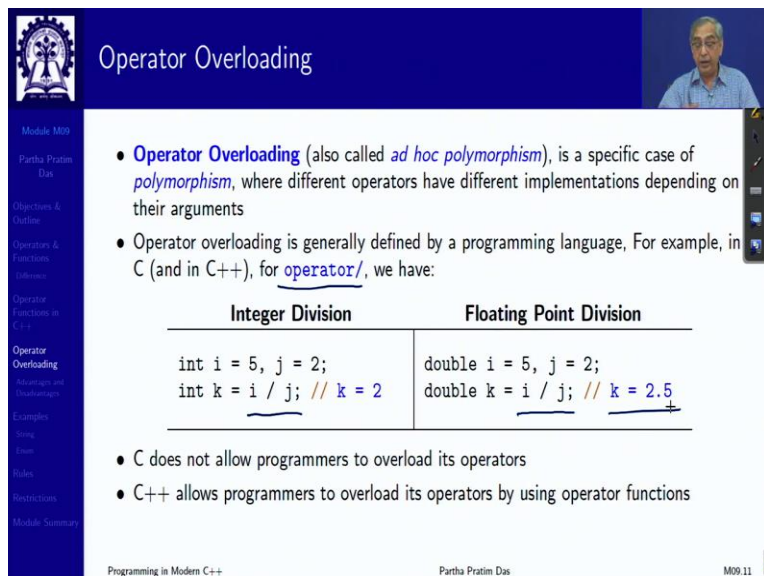
And then what you say that every operator is associated with an operator function that is, if I write a + b, I can also write it in the function notation as operator+(a, b), the prefix notation. What is the name of the function? Name of the function is operator+. If I write a = b, an assignment operator, I can also write it as operator=(a, b).

So, that is corresponding to every operator, I can have an operator function, by using this keyword operator. So, if I have particularly, if I have C assigned a + b precedence that this has to happen first, then this. So, operator+(a, b), is the inner function the outer one cannot be evaluated unless you do this, as I said the depth decides, this is the inner function, and then whatever is the result of this, and c the assignment will happen.

That is the basic idea of treating operators as functions. Now, the operator functions for predefined types are implicit, you cannot do anything with them, you cannot touch them. But, for any user defined type, you can define your operator function. So, if you have a MyType, say some structure, you know something, then you can define them, say operator+ function, for operator+(MyType, MyType), and get a MyType result.

If you do that, then when you write a + b, it will call this function, you are seeing the benefit you have your type of data, and you are doing that operation of addition whatever that means, using a function, but now, you can change the meaning of the plus operator in the context of your MyType to do what you want that operator to do. This is what is known as operator overloading.

(Refer Slide Time: 9:00)



So, operator overloading is also known as ad hoc polymorphism. So, you can see that, the more and more we progress we will have polymorphism, polymorphism, and so on. So, it is a specific case of polymorphism. So, what you do is, in a programming language, if you actually look at, say, your built-in types in C, you already have a lot of overloading.

For example, think about operator division. If you have i, and j which are integers, you write i / j, i divide j, the result is 2. Because it is an integer division we say, and do that same with double types, the result is 2.5 because it is a floating-point division. So syntactically, which looks operator division, actually have different meanings for different types.

But in C, this is only allowed in whatever predefined types you have. In C++, the predefined types will continue or built-in types will continue to have their overloaded operator meaning. But for any type that you define, any structure that you define, any enum that you define, you can define your own operator meaning by providing overloaded operator function.

(Refer Slide Time: 10:31)

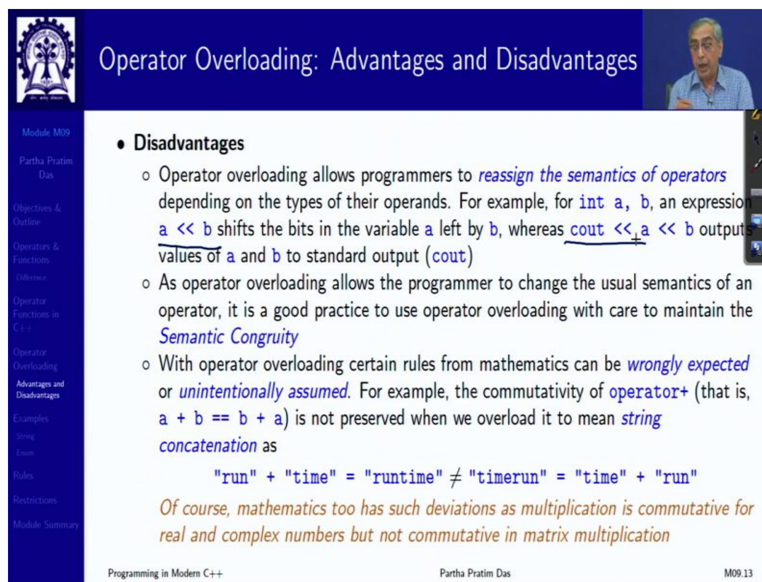
So, what is the advantage, I mean, computationally, you are not doing anything new, it is more like a what is called as syntactic sugar. It makes the syntax of your entire expression much easier. So, if you have if you are dealing with complex numbers, and say you have written a complex created a complex type, and you want to write an expression of this form.

In C, you will always have to write it something like this, multiply b c and then add a, multiply b c, and so on. With overloading, you will be able to exactly write this expression in that form, which you can write for int, which you can write for float, double, you will be able to write that for your complex type, you define a fraction type, you will be able to write similar expressions for that.

Though, the rules of multiplication of complex is different from the rule of multiplication of integer, or double and so on. So, that is, I mean, in scientific community, that is particularly there is a lot of advantages of having this kind of syntactic sugar available. So that your program now

looks like your algebra, what you write on paper in mathematics in physics, your program can be made exactly look like that, not an abstraction of whole range of plethora of function names, and so on, which is clumsy and non-intuitive, and so on.

(Refer Slide Time: 12:03)



There is some disadvantages also, the disadvantages are when you overload at times, you provide very conflicting semantics, conflicting meaning to the operators, one that you have already seen, this is what you have in C or in C++ as well, that a << b is shifting a by b bits, shifting left by b bits, whereas you have seen in the context of output ostream that this means streaming a to cout.

So, the same operator overloaded in the context of two different types, and are semantically very divergent. So, this is, this reassignment of semantics is maybe problematic. So, that should be minimised to the best extent possible to maintain the semantic congruity of the whole language. But that is your design choice.

The second disadvantage that might happen is operators are typically expected to have certain mathematical properties, like plus operator we say is commutative, a + b = b + a. But suppose, you overload the operator plus to mean concatenation of strings, you have already seen examples of that. If you do that, then the operator is no more commutative run plus time is runtime, time plus run his timerun, they are not same.

Now, of course, you accept that because you have accepted it in mathematics, multiplication of integer, complex, double all are commutative, but not of the matrix. So, if these are, there are exceptions in life, there will always be but these are some of the disadvantages that you will have to keep in mind.

And when you overload your operator to do certain, because now you are giving a meaning to your operator, then you should for example, it will be very, non-intuitive to use say the * operator to mean concatenation, you know, somehow the mental sense will not be there. So, that concretely will should be maintained at every level.

(Refer Slide Time: 14:32)

Examples of Operator Overloading

Module M09

Partha Pratim Das

Objectives & Outline

Operators & Functions

Difference

Operator Functions in C++

Operator Overloading

Advantages and Disadvantages

Examples

String

Enum

Rules

Restrictions

Module Summary

Examples of Operator Overloading

Programming in Modern C++ Partha Pratim Das M09.14

Program 09.01: String Concatenation

Concatenation by string functions

```
#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( // Allocation
              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;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Concatenation operator

```
#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); // Allocation
    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; // Overloaded operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Programming in Modern C++ Partha Pratim Das M09.15

So, a couple of examples of operator overloading. Quick once, this is concatenation by using the C string functions, and this is how you will concatenate two parts of the name, copy the first one and then concatenate the second name, and this is using C string again. You are overloading the operator. So, you are saying that operator plus I am overloading in the context of string and string. Look at the parameters carefully plus is binary.

So, it has to take 2 parameters of the same type. The first one is a string, the second one is also a string, string is usually a big structure, you do not want to copy it. So, you are passing it by reference. Certainly, you do not want the operator plus to do something, so that your original string gets changed. So, you make it const. And when you return, you return by value, because it is a new string. Because what you have got by doing this is a new string. So, you will have to it did not exist. So, you cannot have a reference for that.

(Refer Slide Time: 15:53)

Program 09.01: String Concatenation

Module M09 | Partha Pratim Das | Objectives & Outline | Operators & Functions | Difference | Operator Functions in C++ | Operator Overloading | Advantages and Disadvantages | Examples | String | Enum | Rules | Restrictions | Module Summary

Concatenation by string functions

```
#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( // Allocation
                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;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Concatenation operator

```
#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); // Allocation
    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; // Overloaded operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Programming in Modern C++ | Partha Pratim Das | M09.15

So, in the actual terms, you do proper allocation of space for the concatenated string. Using the C function, you do the copy, you do the cat, return s, and this is in to you, for your structure string, which is a character array. This is now your new concatenation operator. And now we can write it simply as if fName plus lName. Beautiful, is not it? That is certainly this means a lot more syntactic convenience in terms of doing this. But conveniences will keep on increasing as we will see.

(Refer Slide Time: 16:44)

Program 09.02: A new semantics for operator+

w/o Overloading +

```
#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; // operator + for int
    cout << x << endl;
}
----------
3
```

- Implicitly converts enum E values to int
- Adds by operator+ of int
- Result is outside enum E range

Overloading operator +

```
#include <iostream>
using namespace std;
enum E { C0 = 0, C1 = 1, C2 = 2 };

E operator+(const E& a, const E& b) { // Overloaded operato
    unsigned int uia = a, uib = b;
    unsigned int t = (uia + uib) % 3; // Redefined addition
    return (E) t;
}
int main() { E a = C1, b = C2;
    int x = -1;

    x = a + b; // Overloaded operator + for enum E
    cout << x << endl;
}
----------
0
```

- operator + is overloaded for enum E
- Result is a valid enum E value

Programming in Modern C++ | Partha Pratim Das | M09.16

Here is an example, where I show that well, you can use operators to change the semantics of the operation. Let us say this is an enumerated data, enum E, it has three enumerated literals C0, C1, C2, standing for 0, 1, 2, as you know. Now, if you do, if you take two enum variable a and b and do a add, what add does it do?

This add does a arithmetic conversion from enum to int. Because enum does not have any predefined operator, but it is represented as integer internally. So, what it does what the compiler does, it does what is known as a arithmetic conversion that converts a to int b to int, does a arithmetic addition, and you get the result. So, if you look at that, then expectedly if this is C1, which is 1, this is C2, which is 2, when you add them, x will be 3, perfect.

Now, what I want to do is for this enum E, I want to make the operator plus behave as a modulo addition. I want operator plus to behave as modular addition, that is just not add, but add, divide and take the remainder. So, I am overloading this operator, I told you overloading is not possible for the built-in types. It is possible for the structures and enums. So, I can do this. So again, I take two enums like I did last time, const E& and so on.

I will return an enum by value, because this will be a new value. Internally, I define unsigned integers, I add them, and I do a percentage 3. So, this means I will do a modulo three. So, 0 added with 1 is 1, 0 added with 2 is 2, but 1 added with 2 is 0. 1 plus 2 is 3, modulo 3 is 0. 2 added with 2 is 1. So that is the kind of.

So, this is a redefined addition that I have, I have a new semantics and then I can use it exactly in the same way. Now, the arithmetic conversion will not apply, because I have already given a overloaded operators. So, the compiler will know that a is of type E, b is of type E, and a overloaded plus operator has been given.

So, operator plus so this is now a function. This actually is a function operator+(a, b) which binds with this function. So, it will call this function instead of doing arithmetic conversion of a and b into int, and doing an integer addition. So, when it does that, it will do the modular addition and you will get it a value 0, try it out great fun.

(Refer Slide Time: 20:15)

Operator Overloading Rules

Operator Overloading Rules

Programming in Modern C++ Partha Pratim Das M09.17

Operator Overloading: Rules

- *No new operator* such as `operators**` or `operators<>` can be defined for overloading
- **Intrinsic properties** of the overloaded operator *cannot be changed*
  - Preserves *arity*
  - Preserves *precedence*
  - Preserves *associativity*
- These operators can be overloaded:

```
[] + - * / % ^ & | ~ ! = += -= *= /= %= ^= &= |=
<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]
```

- For *unary prefix operators*, use: `MyType& operator++(MyType& s1)`
- For *unary postfix operators*, use: `MyType operator++(MyType& s1, int)`
- The `operators::` (*scope resolution*), `operator.` (*member access*), `operator.*` (*member access through pointer to member*), `operator sizeof`, and `operator?:` (*ternary conditional*) *cannot be overloaded*
- The overloads of `operators&&`, `operators||`, and `operators,` (*comma*) *lose their special properties: short-circuit evaluation* and *sequencing*
- The overload of `operators->` must either return a raw pointer or return an object (by reference or by value), for which `operators->` is in turn overloaded

Programming in Modern C++ Partha Pratim Das M09.18

Now, naturally operator overloading has certain rules, and quite a bit of them actually and you have to slowly get them settled in your mind, that is you cannot define any new operator. Operator are as given in the language, I cannot say that I will have an operator **, or an operator <>, not possible.

Then the intrinsic property of every operator must be preserved, that cannot be changed. What are the interesting property? There are three intrinsic properties of an operator. One is the arity, that is how many operands does it take 1, 2, or 3 typically. Two is a precedence in the whole sequence of operators, you cannot change that. And three is its associativity, that is whether it is left associative.

Whether it is, or whether it is right associative, or it may be non-associative, that is it cannot occur one after the other. So, this cannot be changed. Now, given that, this is just the operators that can be overloaded and they are kind of given in the order in which I mean order of their precedence you will not be able to remember that lookup or use parenthesis when you have confusion.

All these operators can be overloaded, they can see there is a huge number of them. So, addition, subtraction, multiplication, division, modulus all of these can be overloaded. Less than, greater than, all binary stuff can be overloaded, assignment can be overloaded, your pointers can be overloaded and so on.

Now, the question is with unary operator because unary operator can be a prefix one unary operator can be a postfix one. Operator ++, or operator --, is a same operator in one context this prefix in one context is postfix. So, how do you differentiate between their functions, because their function both functions will be called as operator ++.

So, what convention is given is, if you want the prefix operator you just write your parameter type, if you want the postfix operator you write the parameter type, and write a dummy int kind of a small compiler hack, write a dummy int that does not have any variable, but just write the dummy int, that tells the compiler that you are dealing with a postfix operator ++, similar for --.

Now, there are certain operators which cannot be overloaded like scope resolution, it is static type, it is just to see the name, we will we will learn more about that. The dot operator like structure components, that dot operator cannot be overloaded, because dot operator is a basic way to take a component out of a structure.

So, if you overload that means something different that becomes a problem. Similarly, your sizeof and all these operators cannot be overloaded. And then some operators you can overload. But if you overload their intrinsic semantic, extended semantics will change. What does that mean? Suppose you think about two Boolean expressions which you are ANDing.

Now, if you are trying to end, then it does a shortcut evaluation, that is if the first operand is false. You do not need to evaluate the second operand, you are ending because if the first operand is false, then it does not matter whether the second operand is true or false, the entire expression is false. So, just avoids that.

That is known as a shortcutting, shortcut evaluation. The similar thing holds for OR, there is a comma operator which sequence the different expressions and so on. So, these kinds of properties will get lost, if you overload operators. Last but not the least, if you overload the function, your pointer operator, indirection operator, then that must return a pointer.

And you will not understand this fully now. If that pointer is not a raw pointer, then it will be applied again to get the next pointer. Till you get a raw pointer. I am sure this is not this is absolutely not making sense to you, we will at some point talk about smart pointers, in one of the later modules and then you will understand this, this overloading how important it is.

Mind you, this is a special, there is a very important operator which can be overloaded which is a function operator, you call it like this. So, that is considered an operator and you can overload the

function operator, and create something which is known as a functor. Again, do not worry too much about these names, all of these will be covered at some later point of time appropriately.

(Refer Slide Time: 25:37)

Operator Overloading Restrictions

Module M09

Partha Pratim Das

Objectives & Outline

Operators & Functions

Difference

Operator Functions in C++

Operator Overloading

Advantages and Disadvantages

Examples

String

Enum

Rules

Restrictions

Module Summary

Operator Overloading Restrictions

Programming in Modern C++ Partha Pratim Das M09.19

Overloading: Restrictions

operator	Reason
dot (.)	It will raise question whether it is for *object reference* or *overloading*
Scope Resolution (::)	It performs a (compile time) *scope resolution* rather than an *expression evaluation*
Ternary (?:)	Overloading `expr1? expr2: expr3` would not guarantee that *only one of* `expr2` and `expr3` was executed
sizeof	Operator `sizeof` cannot be overloaded because *built-in operations*, such as incrementing a pointer into an array *implicitly depends on it*
&& 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

Programming in Modern C++ Partha Pratim Das M09.20

Now, the restrictions you must have realised by now, the dot cannot be overloaded, because with that the object you will not know whether it is overloading or the object reference, the scope resolution is static type cannot be overloaded, ternary operator cannot be overloaded either because ternary operators guarantees out of, what is a ternary operator?

Expression 1 is true means expression 2 is evaluated, 3 skipped. Expression 1 is false, expression 3 is evaluated, 2 skipped. So, only 1 of expression 2 and 3 have to be evaluated. This basic property cannot be maintained if the user redefines that operator. So, it has not been given to redefine. Operator sizeof cannot be redefined, because the compiler itself depends on it.

So, you cannot change its meaning. Similarly, the evaluation may your logical connectives and, and or operations if you overload then you will lose that shortcut property, if you overload comma, you will lose the property of sequencing, and you cannot overload the & operator.

Again, & by itself is overloaded, because it can mean bit end, and it can mean address of, in the context of address of it cannot be overloaded, because it will it has to guarantee that, it should not work for a incomplete type and all. Do not break your head on that, just know that cannot be overloaded, you will slowly understand the reason.

(Refer Slide Time: 27:32)

Module Summary

- Introduced operator overloading with its advantages and disadvantages
- Explained the rules of operator overloading

Programming in Modern C++ Partha Pratim Das M09.21

So, here we have introduced again another very important aspect of C++ the ad hoc polymorphism of operator overloading, how to overload operators. And going forward you will see that we will overload variety of operators, for convenience and we will create our own types like complex I will show you that we will create a complete complex type, which will behave exactly like the int type, but with all the semantics of the complex numbers. Thank you very much for your attention, and we will meet in the next module.