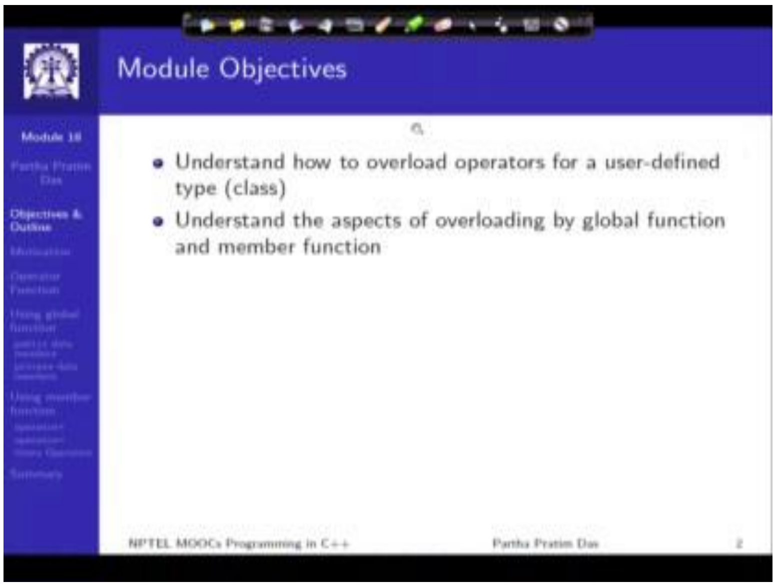
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 33
Overloading Operator for User - Defined Types: Part 1

Welcome to Module 18 of Programming in C++. We have earlier talked about operator overloading. We had discussed that operators in C++, majority of them can be associated with corresponding operator functions that can be overloaded defined by the user. In this module and the next one, we would take a deep look into operator overloading for user-defined types it is a connected discussion. So, this will be the part-1 of this discussion.

(Refer Slide Time: 01:09)



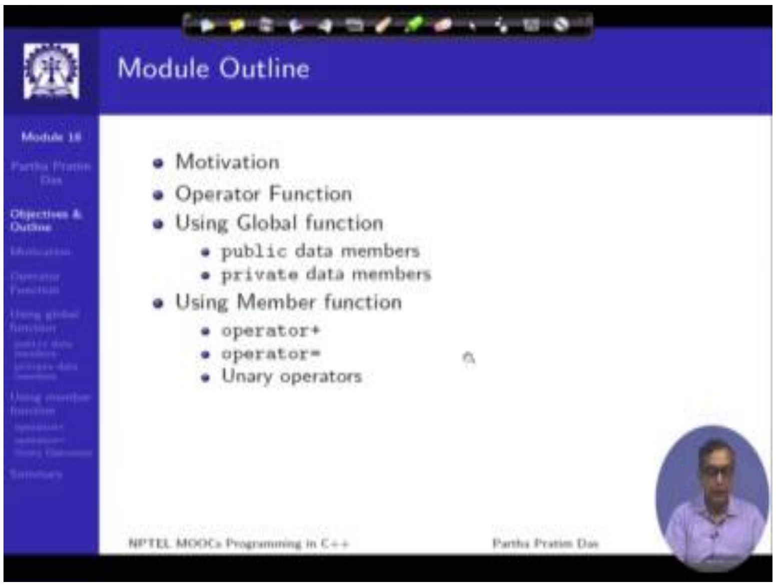
Module Objectives

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by global function and member function

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We will try to understand how to overload operators for user-defined types. And we would explore the aspects of overloading using global functions and member functions in this module.

(Refer Slide Time: 01:27)



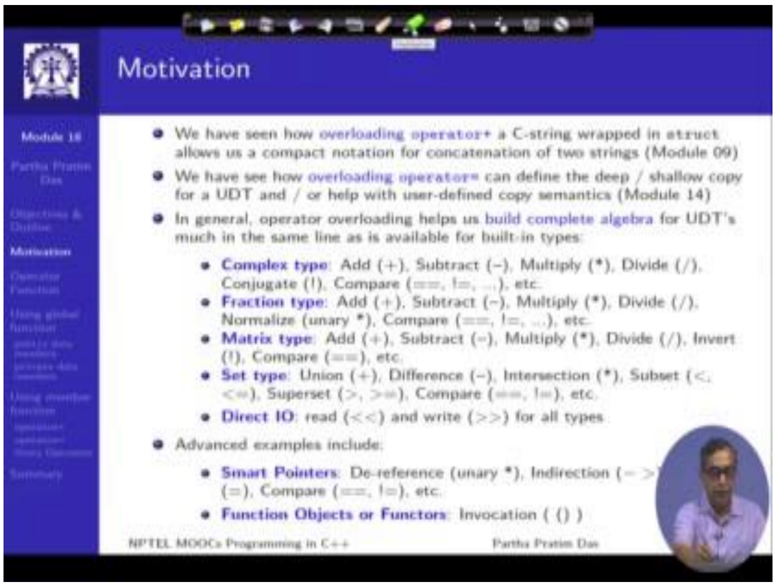
Module Outline

- Motivation
- Operator Function
- Using Global function
  - public data members
  - private data members
- Using Member function
  - operator+
  - operator=
  - Unary operators

NPTEL MOOCs Programming in C++ Partha Pratim Das

The outline as usual would be available on the left of your screen.

(Refer Slide Time: 01:36)



Motivation

- We have seen how overloading operator+ a C-string wrapped in struct allows us a compact notation for concatenation of two strings (Module 09)
- We have see how overloading operator= can define the deep / shallow copy for a UDT and / or help with user-defined copy semantics (Module 14)
- In general, operator overloading helps us build complete algebra for UDT's much in the same line as is available for built-in types:
  - **Complex type**: Add (+), Subtract (–), Multiply (*), Divide (/), Conjugate (!), Compare (==, !=, ...), etc.
  - **Fraction type**: Add (+), Subtract (–), Multiply (*), Divide (/), Normalize (unary *), Compare (==, !=, ...), etc.
  - **Matrix type**: Add (+), Subtract (–), Multiply (*), Divide (/), Invert (!), Compare (==), etc.
  - **Set type**: Union (+), Difference (–), Intersection (*), Subset (<, <=), Superset (>, >=), Compare (==, !=), etc.
  - **Direct IO**: read (<<) and write (>>) for all types
- Advanced examples include:
  - **Smart Pointers**: De-reference (unary *), Indirection (–>), Copy (=), Compare (==, !=), etc.
  - **Function Objects or Functors**: Invocation (())

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, before we actually start the discussions on operator overloading, let me quickly summarize recap as to why we want operators to be overloaded. I would like to refer to two earlier modules; module 9, where we a discussed about overloading operator plus for structured types, enumerated types to show that the operator plus can be overloaded for a string structure to mean concatenation of strings. Similarly, we saw specific overloading for enum types and so on. Later in module 14, we talked about that operator, overloading for assignment operator is critical to define proper semantics for copy specifically with reference to deep and shallow copy.

In general, as we know operators and functions of the same purpose, but the difference is the operator has a very compact notation, which typically is infix. So, it is possible to write a combination of quite varied complex operations in terms of a compact expression; if I can define proper operator for the proper functionality that we want to do. So, these are by default available for built in types like int, like a double some operators are available for pointer type as well and so on.

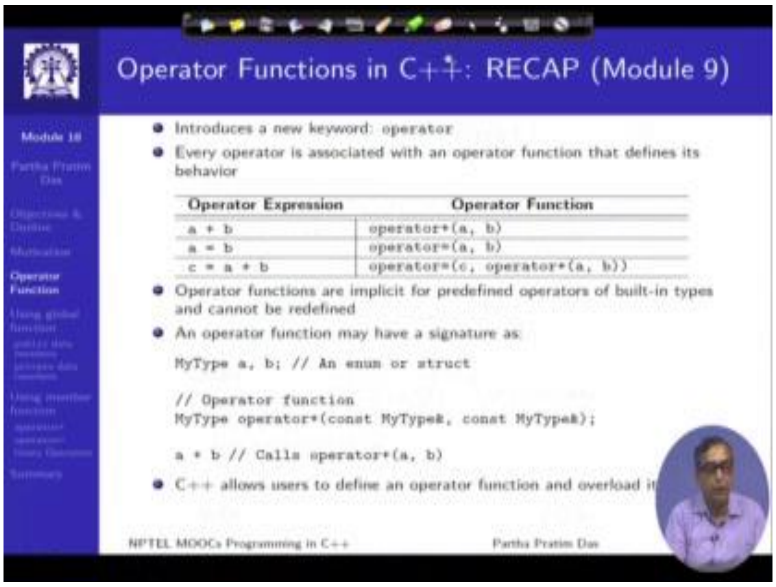
So, if I can overload the operators for a type that we are going to define; that is a user-defined type; much in the way that the operators are defined for the built in types. Then for user-defined type, also we can write expressions, we can write actually build a complete algebra based on the particular user-defined type or UDT that we are building.

So, here I have just mentioned and tried to give a very aboard outline, in terms of what can be done. For example, the C++ does not have a complex type. It has double type, but it does not have a complex type. But using operator overloading, we can define like the operator plus we can say is the add of complex numbers, this is subtraction of complex number, multiplication of complex number, division of complex number. All those can be defined in terms of the corresponding operators and something, which is specific for complex type like finding the complex conjugate. We can use the exclamatory operator; overload the exclamation operator to mean that the sense being that exclamation operation is kind of a negation. So, complex conjugate is kind of a negation of that operator and so on.

Similarly, we can have fraction types, we can have matrix types where we can have a operators for all matrix a algebra including inversion of matrices, we can have set types with union, difference, superset, subset relations and so on. So, all of these and variety of different types that you may want to need to define create for as UDT you can built complete types using operator overloading. And particularly, in terms of the IO’s, we can do very compact IO in terms of the operator output and operator input streaming. Those operators could also be overloaded for the UDT's to give very compact IO structures.

And I would just like to mention, but it is somewhat at the at an advance level. That some of the very nice designs in C++ of a smart pointer, of functions and so on are base significantly on operator overloading.

(Refer Slide Time: 05:45)



Operator Functions in C++: RECAP (Module 9)

Module 18

Partha Pratim Das

Objectives & Outline

Motivation

Operator Function

Using global function

public data members

private data members

Using member function

operator+

operator=

Unary Operators

Summary

- 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:

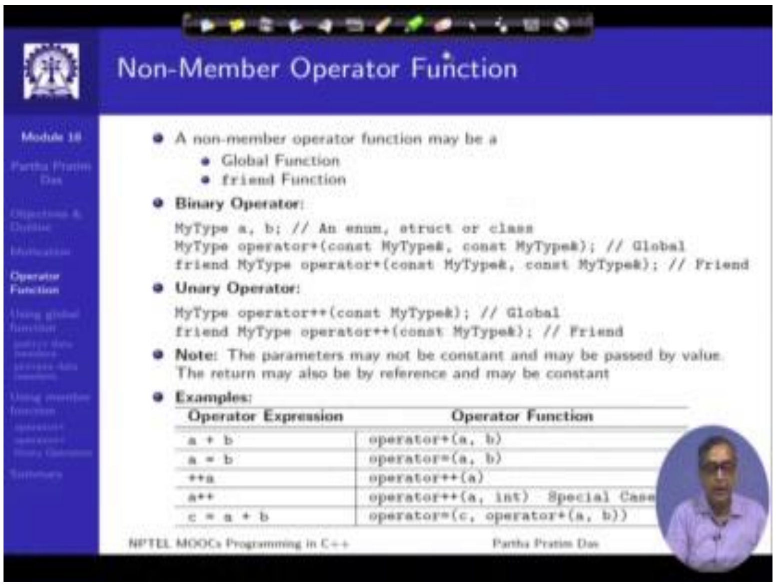
```
MyType a, b; // An enum or struct

// Operator function
MyType operator+(const MyType&, const MyType&);

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

So, overloading an operator is a critical requirement for building of a good user-defined type. So, let us just go forward this is a quick recap from module 9, we had observed that for every operator, there is an operator function, which we can define. And that operator function has certain a signature like the operator plus. In this case, we have already seen this; I will not waste a lot of time here.

(Refer Slide Time: 06:15)



Non-Member Operator Function

Module 16

Partha Pratim Das

- A non-member operator function may be a
  - Global Function
  - friend Function
- **Binary Operator:**

```
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&); // Global
friend MyType operator+(const MyType&, const MyType&); // Friend
```

- **Unary Operator:**

```
MyType operator++(const MyType&); // Global
friend MyType operator++(const MyType&); // Friend
```

- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

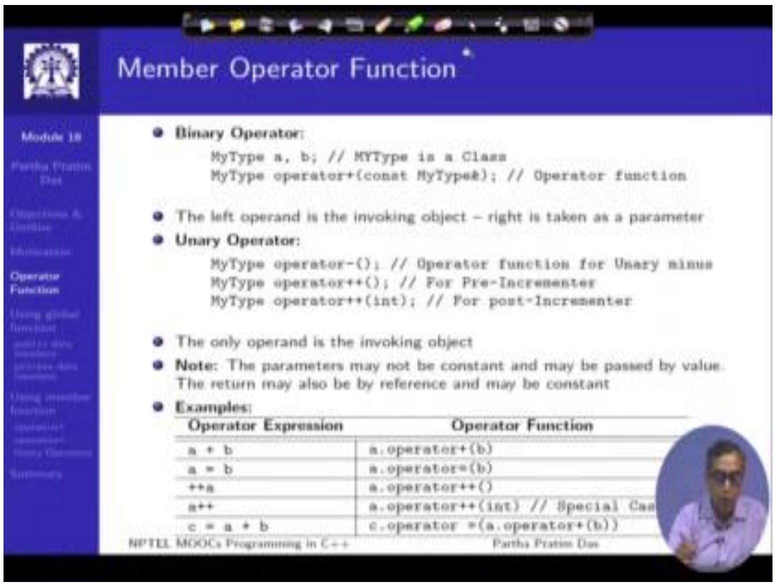
Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`++a`	`operator++(a)`
`a++`	`operator++(a, int)` Special Case
`c = a + b`	`operator=(c, operator+(a, b))`

NPTEL MOOCs Programming in C++ Partha Pratim Das

Just recap, in case it has become easy in your mind. Now this operator function could be a non-member function. Like it could be a global function and we know that there are friend functions also. So, if a operator function is a global function or a friend function then it could look something like, this is a global operator function for adding two objects of my type whatever that my type is. Similarly, for a - if it is friend, then it will look something like this only difference being the friend keyword coming in.

Similarly, we can have global or friend functions for unary operators these are for binary operators. So, you can have it for unary operators. We can have specific operators for a prefix operator and postfix operator and so on.

(Refer Slide Time: 07:10)



Member Operator Function

Module 18

Partha Pratim Das

- **Binary Operator:**

```
MyType a, b; // MYType is a Class
MyType operator+(const MyType&); // Operator function
```

- The left operand is the invoking object – right is taken as a parameter
- **Unary Operator:**

```
MyType operator-(); // Operator function for Unary minus
MyType operator++(); // For Pre-Incrementer
MyType operator++(int); // For post-Incrementer
```

- The only operand is the invoking object
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

Operator Expression	Operator Function
`a + b`	`a.operator+(b)`
`a = b`	`a.operator=(b)`
`++a`	`a.operator++()`
`a++`	`a.operator++(int) // Special Cas`
`c = a + b`	`c.operator =(a.operator+(b))`

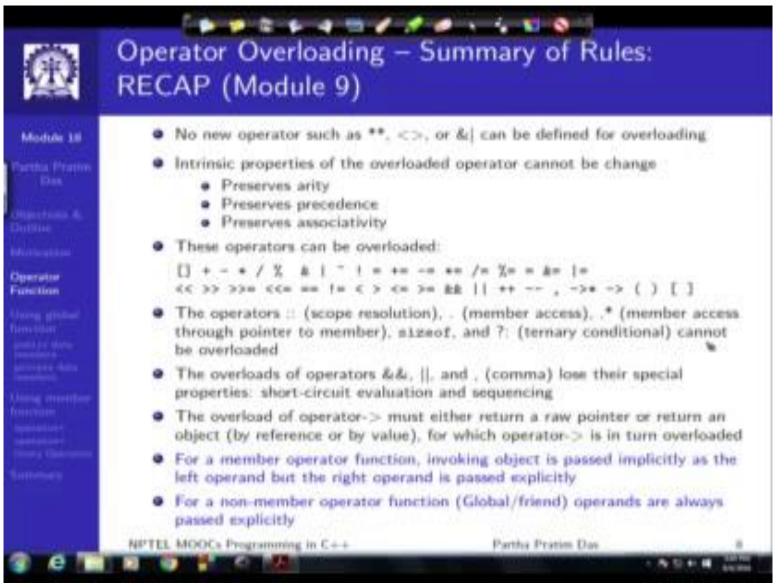
NPTEL MOOCs Programming in C++ Partha Pratim Das

In addition, we could also have operator functions, which are basically member functions. So, if it is a member function then the operator function will look something like this. A one major point to note here that plus is a binary operator, so it needs two operands. But here you are you will see only one parameter for operator plus, because since this a operator function is a member function the first operand or the left hand side operand the left operand of the operator is a object on which you are invoking the operator function. So, as you know that if I have a member function then in here I have the invisible this pointer for every member function.

So, this invisible this pointer actually means that my first operand is star this or the current object that I have, that always will be the left operands and the right operand is what is passed through here. So, in case of member function this is a difference from the global or friend function.

The unary operators can also be defined, in a way naturally unary operators will not have an operand, because they have only one and that is the object on which you are invoking it. With the exception that if it is a post fixed operator, then we will have an operand of type int that actually will not have an operand? But the signature has an additional type int this means that it is a post fix operators. So, that is used to distinguish between the prefix operators which is this one and the post fix operator, because both of these by name are plus plus. So, this additional type in the signature list designates that which one is the prefix operator, which one is the post fix operator. So, these are the different options through which we can actually define the operator functions and overload them.

(Refer Slide Time: 09:15)



Operator Overloading – Summary of Rules: RECAP (Module 9)

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

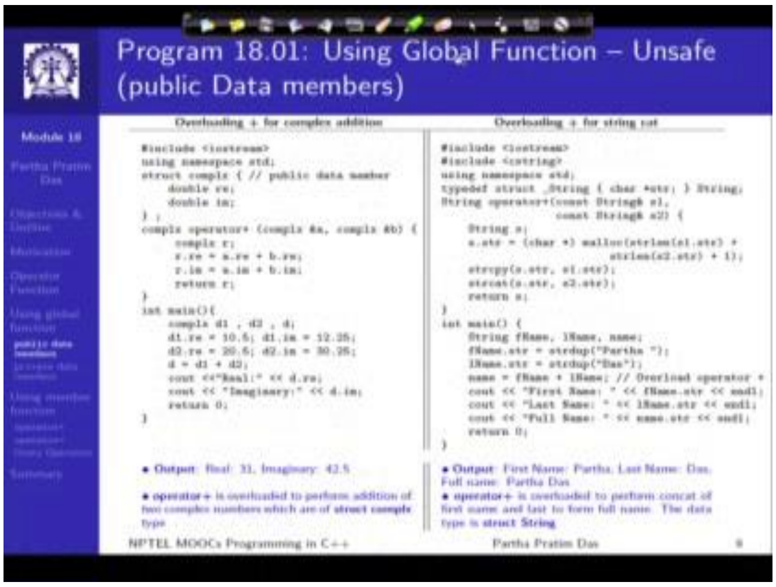
```
[] + - * / % & | ~ ! = += -= *= /= %= = &= |=
<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]
```

- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), sizeof, and ?: (ternary conditional) cannot be overloaded
- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator-> must either return a raw pointer or return an object (by reference or by value), for which operator-> is in turn overloaded
- For a member operator function, invoking object is passed implicitly as the left operand but the right operand is passed explicitly
- For a non-member operator function (Global/friend) operands are always passed explicitly

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

So, I would again by quickly refer you to module 9. Where, we saw the summary of rules of what we can do for operators. We cannot change the basic properties of the operators and there is a fixed list of operators which only can be overloaded there are some operators which cannot be overloaded, some that should not be overloaded, we discus is on depth on module 9, so I will skip that. Now what is additional is a these two points which have highlighted in blue line is for a member function invoking object is passed implicitly which I have just explained and that turns out to be the left operand, so only the right operand is explicitly mentioned. And in case of global function or friend function certainly both the operands are explicitly mentioned.

(Refer Slide Time: 10:10)



Program 18.01: Using Global Function – Unsafe (public Data members)

Module 18

Partha Pratim Das

Objectives & Outline

Motivation

Operator Function

Using global function

public data members

private data members

Using member function

operator+

operator=

Unary Operators

Summary

Overloading + for complex addition

```
#include <iostream>
using namespace std;
struct complx { // public data member
    double re;
    double im;
} ;
complx operator+ (complx &a, complx &b) {
     complx r;
     r.re = a.re + b.re;
     r.im = a.im + b.im;
     return r;
}
int main(){
    complx d1 , d2 , d;
    d1.re = 10.5; d1.im = 12.25;
    d2.re = 20.5; d2.im = 30.25;
    d = d1 + d2;
    cout <<"Real:" << d.re;
    cout << "Imaginary:" << d.im;
    return 0;
}
```

- **Output**: Real: 31, Imaginary: 42.5
- **operator+** is overloaded to perform addition of two complex numbers which are of **struct complx** type

Overloading + for string cat

```
#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;
}
```

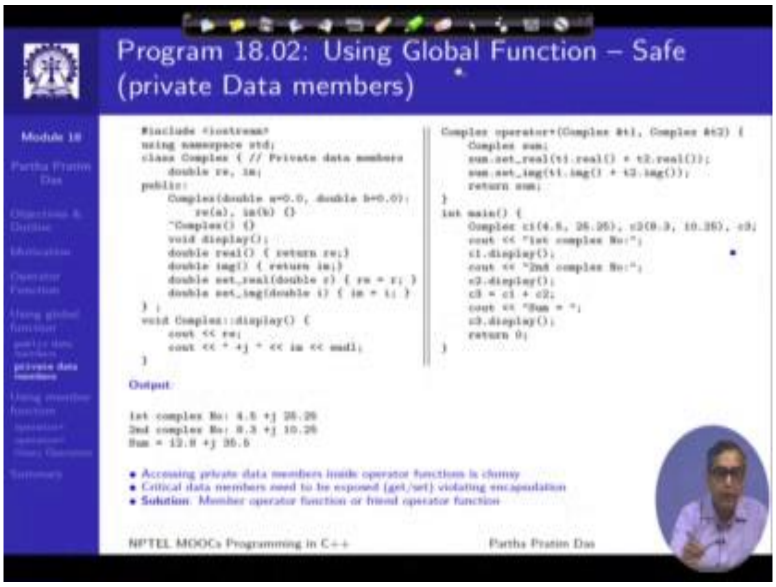
- **Output**: First Name: Partha, Last Name: Das, Full name: Partha Das
- **operator+** is overloaded to perform concat of first name and last to form full name. The data type is **struct String**

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

So, this is a basic of operator overloading. So, with this now let us a look back into some of the code that we had seen earlier is to try to overload an operator using a global function. So, I have a simple a complex class which I have written in term of a structure. So, which means that both it is members are publicly available and we have written operator plus addition operator for this complex structure, which is basically component wise addition of the real and imaginary parts. And if we use that in this expression d1 is being a complex, d2 being another complex we can write d1 plus 2. To mean addition of two complex numbers, which will actually this operator we will actually invoke this function the operations will happened and that result will get assigned to d. So, this is fine.

I am sure you will not have a much difficulty in understanding that. On the right hand side you have another example, where we are using a structure to rap a C string here and based on that structure type, the string structure type, we have defined and overloading for the operator plus again for the string. So, that given two strings here we can write and expression for the first name and the last name which actually goes to this function and performance a concatenation. So, depending on the kind of a type we have the same operator plus in some case is may being addition of complex numbers, in some case it means at the concatenation of strings and so on. This is quite straightforward to do and for this, we have used the global functions here.

(Refer Slide Time: 12:06)



Program 18.02: Using Global Function – Safe (private Data members)

Module 18

Partha Pratim Das

```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) {}
    ~Complex() {}
    void display();
    double real() { return re;}
    double img() { return im;}
    double set_real(double r) { re = r; }
    double set_img(double i) { im = i; }
} ;
void Complex::display() {
    cout << re;
    cout << " +j " << im << endl;
}
```

```
Complex operator+(Complex &t1, Complex &t2) {
    Complex sum;
    sum.set_real(t1.real() + t2.real());
    sum.set_img(t1.img() + t2.img());
    return sum;
}
int main() {
    Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
    cout << "1st complex No:";
    c1.display();
    cout << "2nd complex No:";
    c2.display();
    c3 = c1 + c2;
    cout << "Sum = ";
    c3.display();
    return 0;
}
```

Output:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Accessing private data members inside operator functions is clumsy
- Critical data members need to be exposed (get/set) violating encapsulation
- **Solution**: Member operator function or friend operator function

NPTEL MOOCs Programming in C++ Partha Pratim Das

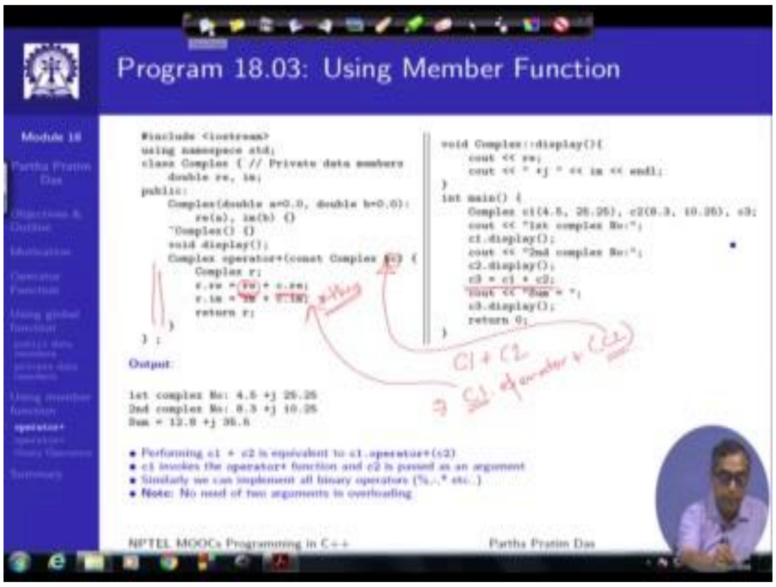
Now naturally, if you use a global function then we know that is not an advisable thing a in object oriented programming system, because the global functions, but to work they naturally needed that the data members the ma particularly the real and imaginary part of the complex number. How to be made public and are we have repeated taken this principle that the data member should actually be private. So, we should make them private. Only the different operations that the class supports, only those should be public.

Now if we do that then naturally a certainly with this we cannot write the operator plus overwrite the operator plus as a global function directly, because this global function will not able to access the private data members. So, what will have to do for making this work, we have to add a bunch of get set functions for the components. Where I can read the real component, read the imaginary component, write the real component write the imaginary component.

Therefore, then using this get set methods we can write a global operator overloading for the operator plus. This works this will work fine this will work as it is and we have been able to make the data members private, but this tell does not pretty much give a better solution. Because, in order to implement the global function we had to provide all these get set function. Then the more of the get set functions you provide your actually exposing your internal data to the external world.

Because, well people cannot directly access the data members and, but since the operator plus can access set and get a real and imaginary parts any other a external function any external class can also do that. So, in a way this solution is kind of breaking the encapsulation that we have. So, certainly we know with global functions we cannot preserve encapsulation.

(Refer Slide Time: 14:17)



Program 18.03: Using Member Function

```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) {}
    ~Complex() {}
    void display();
    Complex operator+(const Complex &c) {
        Complex r;
        r.re = re + c.re;
        r.im = im + c.im;
        return r;
    }
};
```

```
void Complex::display(){
    cout << re;
    cout << " +j " << im << endl;
}
int main() {
    Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
    cout << "1st complex No:";
    c1.display();
    cout << "2nd complex No:";
    c2.display();
    c3 = c1 + c2;
    cout << "Sum = ";
    c3.display();
    return 0;
}
```

Output:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 8.3 +j 10.25
Sum = 12.8 +j 35.5
```

- Performing c1 + c2 is equivalent to c1.operator+(c2)
- c1 invokes the operator+ function and c2 is passed as an argument
- Similarly we can implement all binary operators (%,-,* etc..)
- **Note:** No need of two arguments in overloading

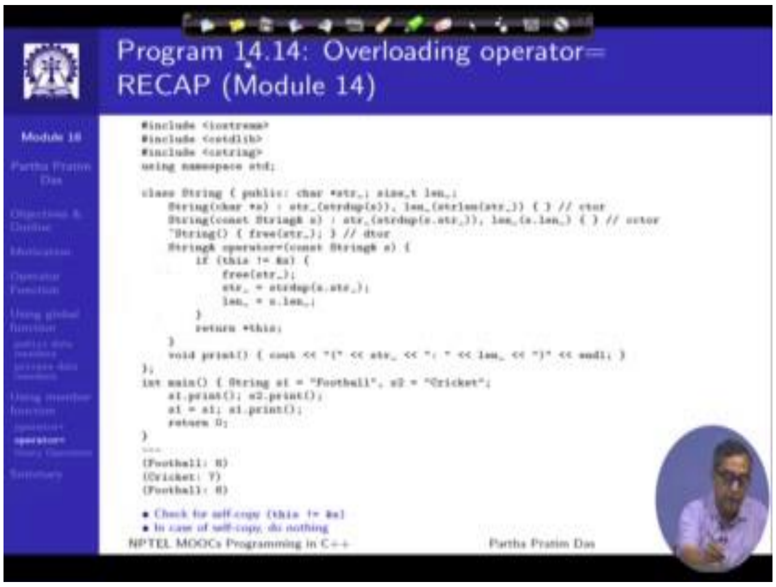
NPTEL MOOCs Programming in C++ Partha Pratim Das

The next option we will look into which is basically using member functions and for doing the same task. Let us now consider that the operator now is overloaded as a member function. So, you can see that it has moved inside the class. My data members are still private and since it is moved inside the class, it has only one parameter, which is the right hand side parameter. And the left hand side parameter would be the object itself. So, when I want to do add I, add re with c dot re. Where, c is the right hand operand and re as you know refers to the real component of the current object on which this method has been invoked, which has to be the left hand side operand.

So, as in a here if you have written c1 plus c2 as we write c one plus c2 this implies that the notation would be that this is equivalent to c one operator plus c2. So, which means that the c2 here is a right hand side operand which will become c and the c1 here is the current object which is star this. That is when we are referring to re we are actually referring to the real component of c1, when I referring to c dot re we are referring to the real component of c2 and that is how this computation will go through.

Certainly this again gives you the same a solution this gives you a same answer has advantage of a making a protecting the encapsulation that we want to preserve. So, we can a very easily work with this and have good operator overloading using the member functions.

(Refer Slide Time: 16:05)



Program 14.14: Overloading operator= RECAP (Module 14)

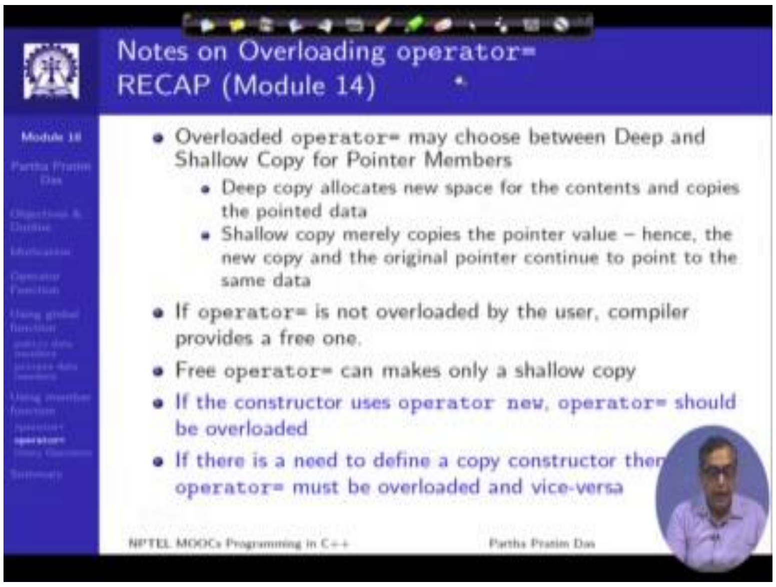
```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // cctor
    ~String() { free(str_); } // dtor
    String& operator=(const String& s) {
        if (this != &s) {
            free(str_);
            str_ = strdup(s.str_);
            len_ = s.len_;
        }
        return *this;
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket";
    s1.print(); s2.print();
    s1 = s1; s1.print();
    return 0;
}
---
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for self-copy (this != &s)
- In case of self-copy, do nothing

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now in the same way the as you have seen shown for the binary addition operator several, other operators can be overloaded. For example, if you recall in module 14 while we discussed about various opportunities in options of coping, we showed that operator assignment copy assignment operator. We had been talking about can be overloaded and this is we showed that is a safe way of doing this whole example is actually again repeated from module 14 for your reference.

(Refer Slide Time: 16:45)



Notes on Overloading operator= RECAP (Module 14)

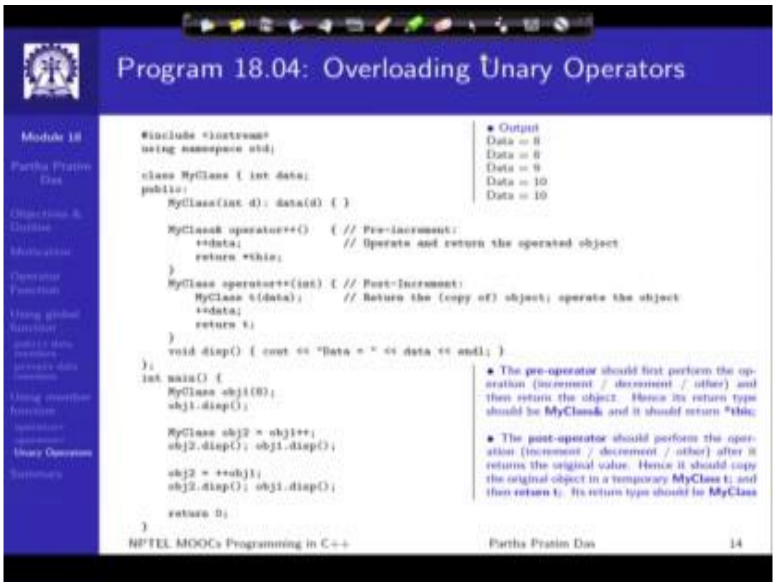
- Overloaded operator= may choose between Deep and Shallow Copy for Pointer Members
  - Deep copy allocates new space for the contents and copies the pointed data
  - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data
- If operator= is not overloaded by the user, compiler provides a free one.
- Free operator= can makes only a shallow copy
- If the constructor uses operator new, operator= should be overloaded
- If there is a need to define a copy constructor then operator= must be overloaded and vice-versa

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, in that we have already seen the how overloading works for the copy assignment operator and the same is also here and we had that noted that this overloading is very important because unless you overload the compiler provides a free copy assignment which does only a shallow copy. So, in the presence of pointer variables that could give you a wrong semantics and you may need to overload and do a deep copy in the copy assignment operator.

Now in addition in blue I have shown what are the additional factors that you should keep in mind, that if you are constructed users operator new. When will the constructor use operator new when you have some dynamically allocable pointer members. So, when if you are compiler is using operator new, then your operator assignment should be overloaded. Because then you have a situation of deciding between shallow copy and deep copy. And at the same time if there is a need to define a copy constructor then there usually must be a need to define a copy assignment operator also and vice versa. So, while you do the design of classes please keep this point in mind because copy assignment operator is a very critical operator which needs to be overloaded.

(Refer Slide Time: 17:54)



Program 18.04: Overloading Unary Operators

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

class MyClass { int data;
public:
    MyClass(int d): data(d) { }

    MyClass& operator++()    { // Pre-increment:
        ++data;                // Operate and return the operated object
        return *this;
    }
    MyClass operator++(int) { // Post-Increment:
        MyClass t(data);       // Return the (copy of) object; operate the object
        ++data;
        return t;
    }
    void disp() { cout << "Data = " << data << endl; }
};
int main() {
    MyClass obj1(8);
    obj1.disp();

    MyClass obj2 = obj1++;
    obj2.disp(); obj1.disp();

    obj2 = ++obj1;
    obj2.disp(); obj1.disp();

    return 0;
}
```

- Output

```
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10
```

- The **pre-operator** should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should return ***this;**
- The **post-operator** should perform the operation (increment / decrement / other) after it returns the original value. Hence it should copy the original object in a temporary **MyClass t;** and then **return t;**. Its return type should be **MyClass**

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

Just going for the ma unary operators can be overloaded the only reason i include them here is unary operators could be of two types prefix and postfix. So, this is a prefix operator and this is a postfix operator. I have already mentioned that the postfix operator is designated by an additional type in the signature, but this is just a placeholder this is just is used by the compiler to resolve between the prefix and the postfix operator. Of course, you do not expect to pass a second parameter for a unary operator, because unary operator already has it is parameter from the class on which the class on which this is invoked either it is this case or it is in this case.

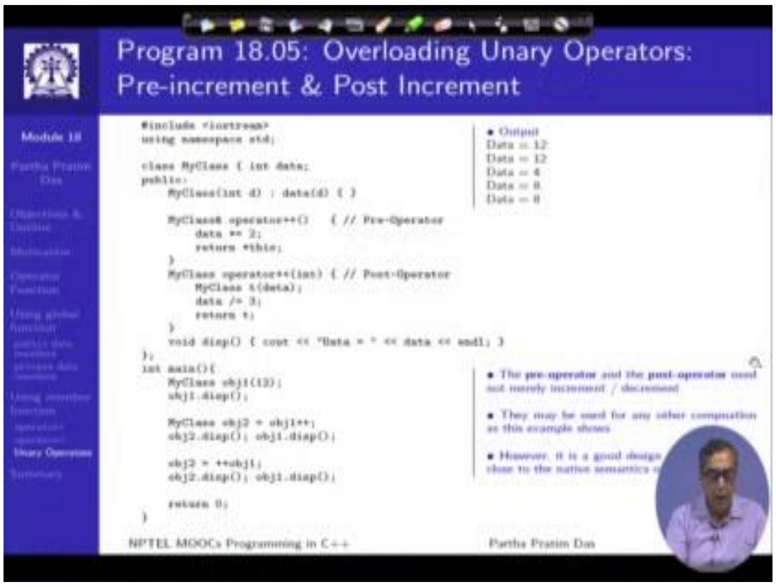
Now, if you looking to the implementation and the written type. Which I would like to little bit highlight you to. So, if it is a prefix increment return the same object on which invocation has happened. So, that tells us that the same object has to come back and naturally return has to be a start this, which basically returns the same object. And before that the incrimination of the operation has already happened.

On the other hand, if you look into the post increment operator then, what you are supposed to get yes the operator has already worked the result of post increment is the original value. The original value that you had before the operator was invoked you get that as a return value and then the object is actually incremented. So, it is done later on.

So, if you have to since the object will get incremented the value in the object will get incremented after this operation any way. The value that is returned must be a copy of the original, because the original object is going to change as soon as you return from this operator.

So, now you need to copy this in terms of a temporary return that temporary and since you are returning a temporary as you have already discussed earlier your return type will have to be a co returned by value it cannot be a return by reference. So, in terms of a post operator your return type will be a written by value type. So, these explanations I have written in here also in case you need to refer to them while you are preparing on this and that is how you can overload unary operators.

(Refer Slide Time: 20:28)



Program 18.05: Overloading Unary Operators: Pre-increment & Post Increment

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

class MyClass { int data;
public:
    MyClass(int d) : data(d) { }

    MyClass& operator++()    { // Pre-Operator
        data *= 2;
        return *this;
    }
    MyClass operator++(int) { // Post-Operator
        MyClass t(data);
        data /= 3;
        return t;
    }
    void disp() { cout << "Data = " << data << endl; }
};
int main(){
    MyClass obj1(12);
    obj1.disp();

    MyClass obj2 = obj1++;
    obj2.disp(); obj1.disp();

    obj2 = ++obj1;
    obj2.disp(); obj1.disp();

    return 0;
}
```

- Output

```
Data = 12
Data = 12
Data = 4
Data = 8
Data = 8
```

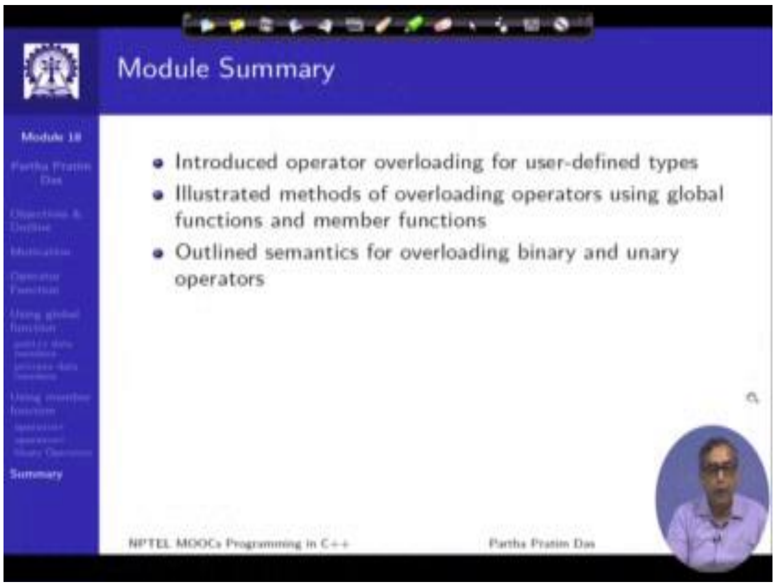
- The **pre-operator** and the **post-operator** need not merely increment / decrement
- They may be used for any other computation as this example shows
- However, it is a good design close to the native semantics

NPTEL MOOCs Programming in C++ Partha Pratim Das

Another point that I would like to highlight in terms of overloading not only unary, but many other operators as well; but I just show you with a unary operators that suppose this operator is unary increment operator, pre increment operator. But it does not necessarily mean that the functionality that you put in has to be an increment function. It could be some other functionality also, like here, I am showing that if I call this operator then my data field will get double and, but what is critical is the basic semantics of the operator that it does return you the object on which you have invoked it. So, this will remain same, this will remain same, but the actual functionality could be different.

And the same holds for post operator as well I have given some different functionality that your data is divided by three. It does not matter, whatever you need, you can put there. But the basic points remain same that you need to copy the original object in a temporary and actually return that temporary through return by value, because that is what a post operator post incrimination operator will lead and these observations. So, it will be similar for the pre increment, pre decrement operator or all of the other unary operators and binary operators that you may have.

(Refer Slide Time: 21:47)



Module Summary

- Introduced operator overloading for user-defined types
- Illustrated methods of overloading operators using global functions and member functions
- Outlined semantics for overloading binary and unary operators

NPTEL MOOCs Programming in C++ Partha Pratim Das

In this module, we have introduced the operator overloading for user-defined types. We have explained why it is important to overload the operators and we have illustrated the basic method of operator overloading by using global function and by using member functions of the classes. In the process, we also observed that you can use a friend function for overloading we will take that up in the next module. In this module, we have overall outline the basic semantics of overloading for binary and unary operators.