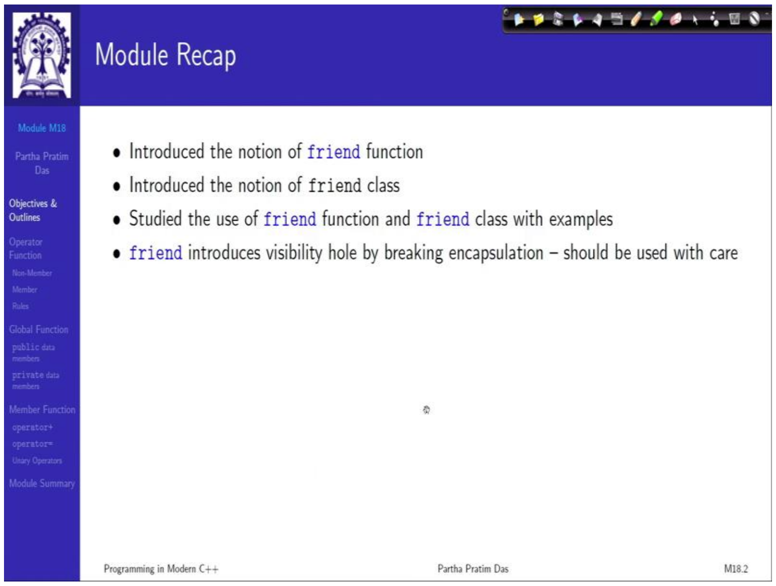
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecturer 18
Overloading Operator for User-Defined Types: Part 1

Welcome to Programming in Modern C++, we can we are in week 4, and we are going to discuss module 18.

(Refer Slide Time: 00:36)



Module Recap

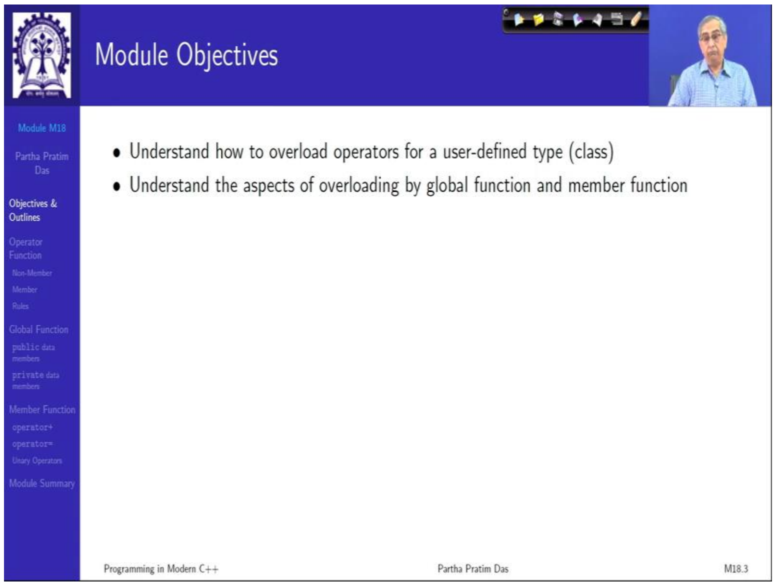
- Introduced the notion of friend function
- Introduced the notion of friend class
- Studied the use of friend function and friend class with examples
- friend introduces visibility hole by breaking encapsulation – should be used with care

Module M18
Partha Pratim Das
Objectives & Outlines
Operator Function
Non-Member
Member
Rules
Global Function
public data members
private data members
Member Function
operator+
operator=
Unary Operators
Module Summary

Programming in Modern C++ Partha Pratim Das M18.2

In the last module, we talked about the notion of friend function and friend classes. Particularly if I would talk about the previous one also we where we covered about static. So, with the static and friend function, we have extended the capabilities of the classes to a significant extent.

(Refer Slide Time: 01:02)

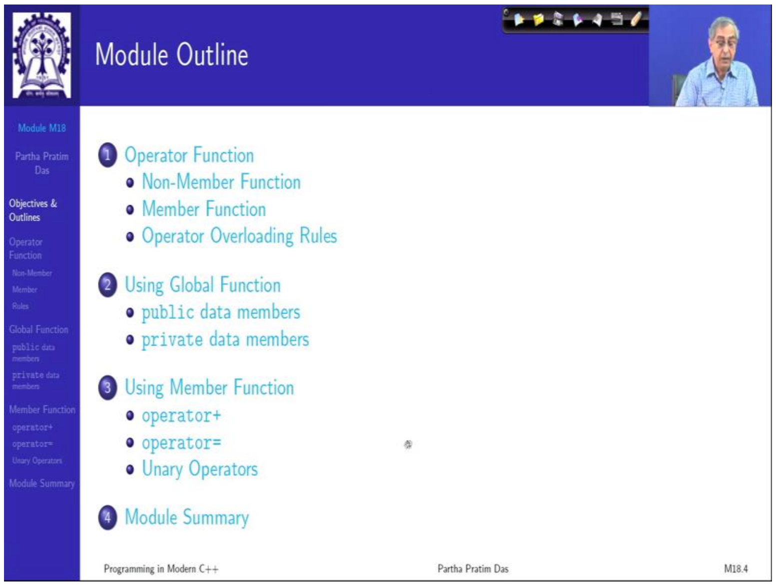


Module Objectives

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

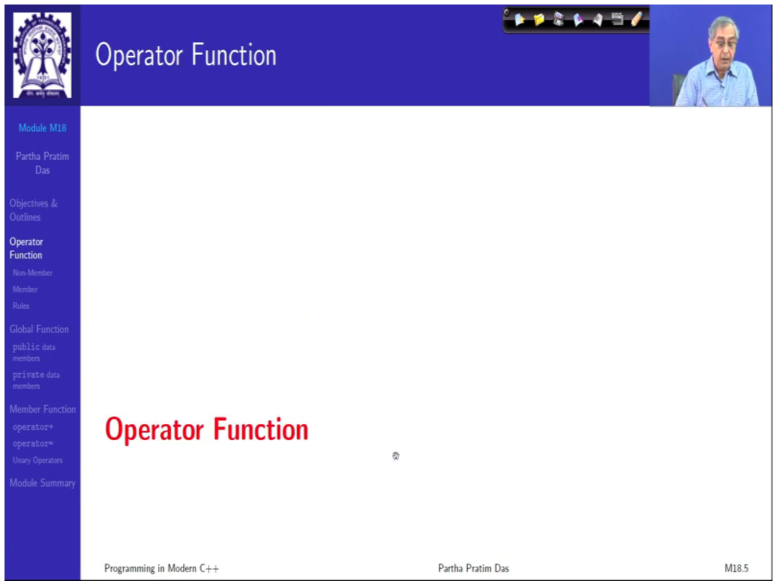
And in the current module, we will talk about overloading of operators, for user defined types. This is not the first time we are hearing about it, we have already talked about this in at least 2 modules earlier in module 9 and module 14 as far as I remember. And, but, here we are going to develop the operator overloading concept in a very structured and comprehensive way and continue that in the next module too.

(Refer Slide Time: 01:36)



Module Outline

1. Operator Function
   - Non-Member Function
   - Member Function
   - Operator Overloading Rules
2. Using Global Function
   - public data members
   - private data members
3. Using Member Function
   - operator+
   - operator=
   - Unary Operators
4. Module Summary


Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

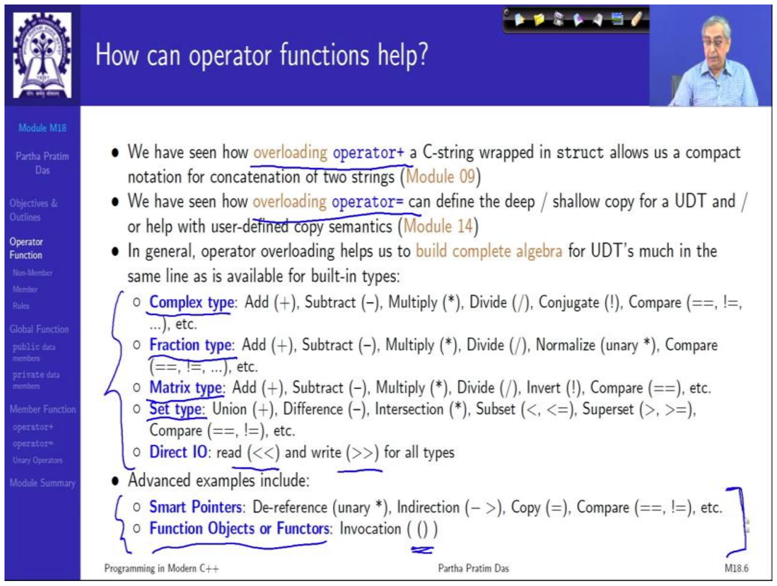
Unary Operators

Module Summary

Operator Function

Operator Function

Programming in Modern C++ Partha Pratim Das M18.5


Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

How can operator functions help?

- 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 seen 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 to 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 (())

Programming in Modern C++ Partha Pratim Das M18.6

Outline as you know, will be available on your left. So, I start with a quick recap of things we have already discussed. If you have further doubts on them, you can go back to the respective modules and look up the details. So, you have seen how to overload a operator plus let us say for allowing the concatenation of strings as a addition operation.

We have also seen overloading this was done in module 9, we have also seen the overloading of assignment operator copy assignment operator as we said in module 14, which can define different kinds of deep or shallow copy semantics for this assignment operator. But in general, so, these are two specific examples, we did. In general, what we want to do, one of the main reasons of operator overloading is we want to build up algebra, algebra of our types in the very same way as is available for the building type.

So, like we have int type we can write a complete algebra for that (a+b)*c, - all of those operators and so on. We can do that for double but, if I want to define a complex type. Now, I know how to define that type in terms of having private data members double, real double, imaginary and constructor, destructor all that.

But how do I build a complete complex type with addition, subtraction, multiplication, conjugation and comparison all of that so that I can just the kind of expressions I can write with int I will be able to write with complex as well, can I do that for proper fractional type rational numbers? Can I do that kind of? Can I have similar types built up for matrices for sets?

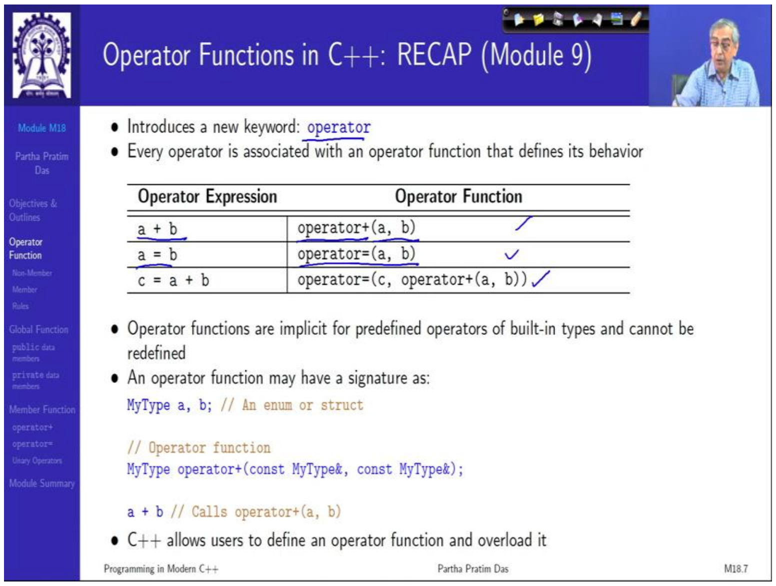
So, far we have been reading or writing values using separate member functions, can I extend the streaming operators for any of these types that I am defining and directly write stream a complex object to the c out or a rational object to the c out and so, on. Further, this will come this will not these will not come as a part of this module because these by each one of these is by themselves very deep.

One is can I have smart pointers, can I have operator overloading for the pointer operations which is dereferencing and indirection and copy and compare and so on. And there are several reasons of why I want to do that. I would like to have smart pointers, pointers they refer to other objects by address but behave in the way I want, not their default behaviour.

So, that is smart pointers. So, we will have a separate discussion on that. And very interestingly, we will talk about function objects or functors these are objects or these are classes, where the function call operator itself is overloaded. So, we know that functions are as defined and you can use the name of the function and call it, have you ever heard of that, you can have a class so that its instance could be used to call a function. The instance itself will be calling the function not a member function, but that itself.

So, there are several effects of that which lead to function objects or functors which are very very critical and particularly with an eye of going towards a modern C plus plus part the C plus plus 11 part where you have lambda functions and so, on, we need to have a good understanding of the functors. And operator overloading is the critical mechanism for all of these, but the present module and the next will focus on the on this part of requirements of operator overloading these two will come later on in separate modules.

(Refer Slide Time: 06:24)



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

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module 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:

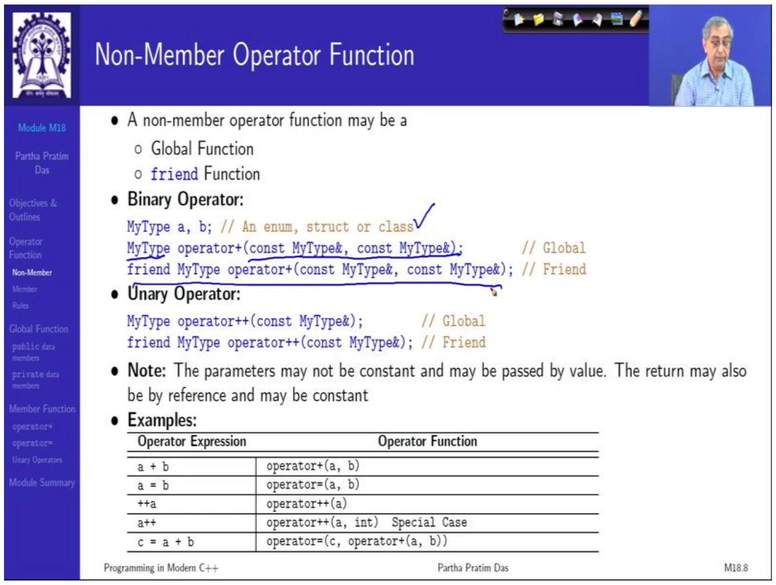
```
MyType a, b; // An enum or struct

// 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

Programming in Modern C++ Partha Pratim Das M18.7

So, with that quick recap, just to tell you that every operator can be thought to be associated with a function using the keyword operator. So, this is what happens a + b is operator+(a, b), a = b is operator=(a, b) and so on. So, this is in C++ you can think of these as these kinds of operators.

(Refer Slide Time: 06:56)



Non-Member Operator Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

- 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
```

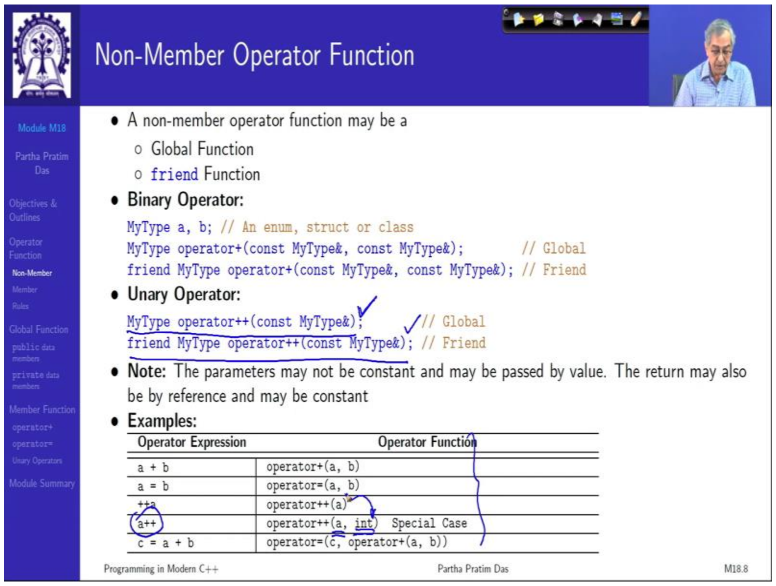
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

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))

Programming in Modern C++ Partha Pratim Das M18.8

Now, there could be several non member operator functions, that is a operator function could be a global function or it could be a friend function. They are not necessarily a part of the class. So, if I have a certain type a and b enumerate enum type, struct or class, then I can have a class operator +, which takes two parameters by reference of myType and gives me the added the result of the addition as a return type. I can also make that a friend function of a class each one of this possible, this is for the binary operator.

(Refer Slide Time: 07:58)



Non-Member Operator Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

- 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:**

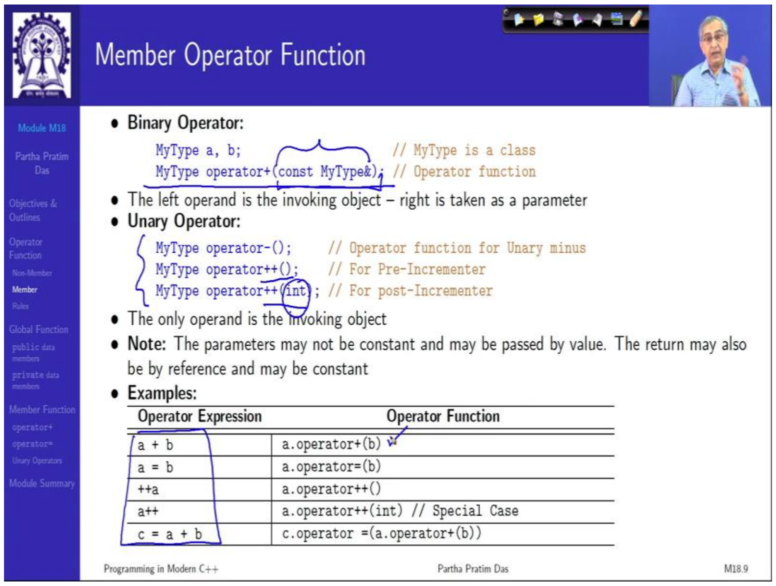
Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
++a	operator++(a)
a++	operator++(a, int) Special Case
c = a + b	operator=(c, operator+(a, b))

Programming in Modern C++ Partha Pratim Das M18.8

I could have for unary operator also. Unary operator, as you know, could be a prefix operator, or it could be a postfix operator. So, if I have a prefix operator, then this is what I am showing is, will be applicable, we will talk about the postfix operator as a special case later on. So, in a, this is a global function, and this is a friend function and here are examples of the same you can see that this is a postfix operators postfix unary operator.

So, here, in addition to the actual operate, you just have a dummy int just to differentiate these two whether it is prefix or postfix, we will have examples of that. So, this is the typical non member operator functions.

(Refer Slide Time: 08:57)



Member Operator Function

- **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 Case
c = a + b	c.operator =(a.operator+(b))

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

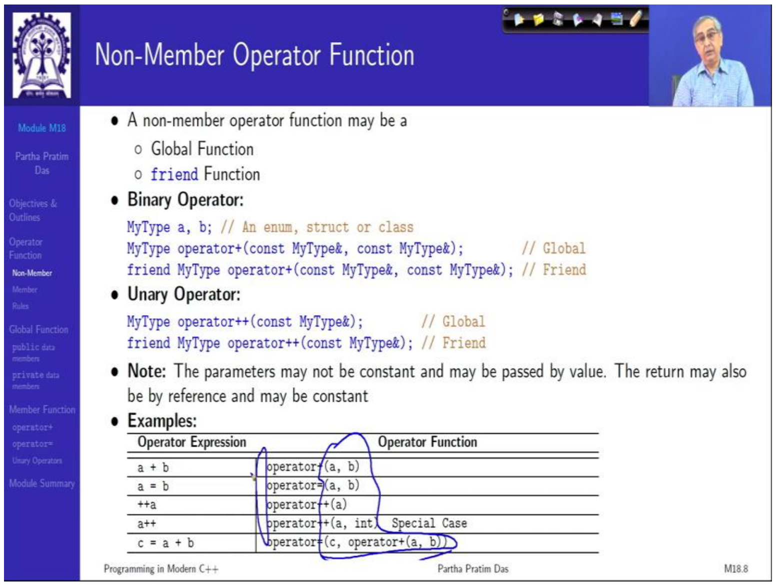
Module Summary

Programming in Modern C++ Partha Pratim Das M18.9

There could be member operator functions also that is the operator could be a part of a class. If it is a part of the class, then certainly it takes only one operand in case of a binary operator. Why? Because, being a member function of the class it is always invoked on an object and when it is invoked on an object, that object is by default, the left operand and the other right operand needs to be passed. So, you see only one operator, but actually there will be two insight.

Similarly, for a unary operator, you will not have any operator because it is invoked on the object. So, that object itself is operant. If it is pre increment kind of you write like this, if it is a post increment, then you need to have that dummy int type to designate it. So, now you see the equivalent expressions between the infix and with the operator this is these are not global, just to point it out to you see this was about the global.

(Refer Slide Time: 10:21)



Non-Member Operator Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function
- Non-Member
- Member
- Rules

Global Function
- public data members
- private data members

Member Function
- operator+
- operator=
- Unary Operators

Module Summary

- 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
```

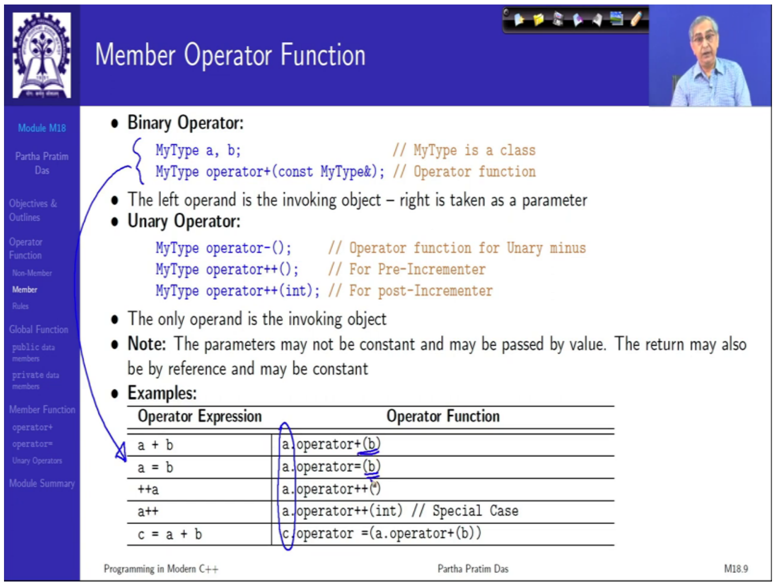
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

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))

Programming in Modern C++ Partha Pratim Das M18.8

So, you can see all of them were having 2 2, 2 or 1 operands and these are global calls not with respect to any particular object. But when you have that.

(Refer Slide Time: 10:42)



Member Operator Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function
- Non-Member
- Member
- Rules

Global Function
- public data members
- private data members

Member Function
- operator+
- operator=
- Unary Operators

Module Summary

- **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
```

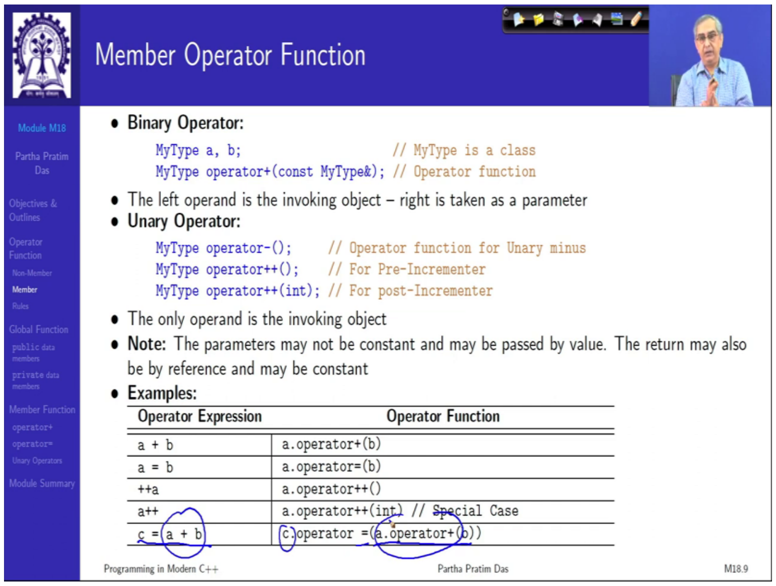
- The only operand is the invoking object
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

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

Programming in Modern C++ Partha Pratim Das M18.9

As in here, I am saying that this is my operator defined in this class, then I have the left operand coming as the object on which the operator is invoked, and the right operand being taken as an actual argument.

(Refer Slide Time: 11:04)



Member Operator Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

- **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
```

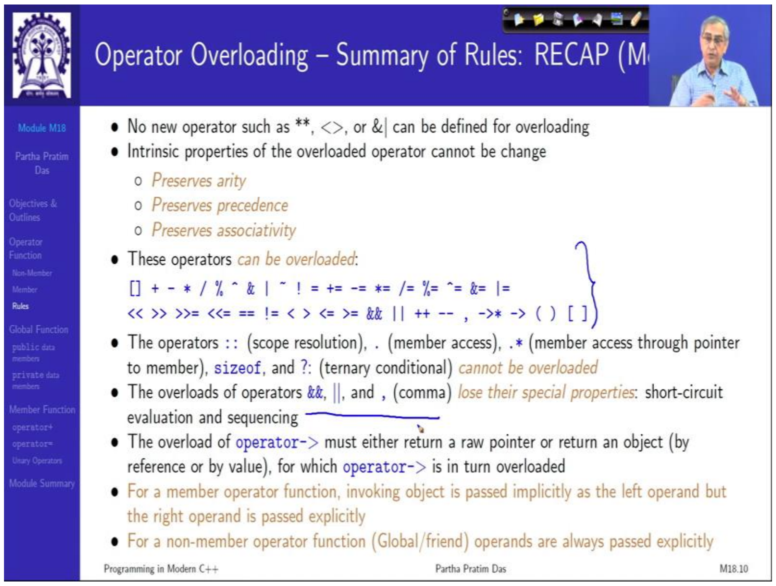
- The only operand is the invoking object
- **Note:** The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant
- **Examples:**

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

Programming in Modern C++ Partha Pratim Das M18.9

You can see the effect of this also. So, what is this this part will happen first, so, it is a.operator+(b) and that result will be assigned to c. So, c is the left hand operand, so, c.operator= of this value. So, these are the equivalent you know function like notation, which will actually get evaluated internally.

(Refer Slide Time: 11:28)



Operator Overloading – Summary of Rules: RECAP (M

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

- 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*:

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

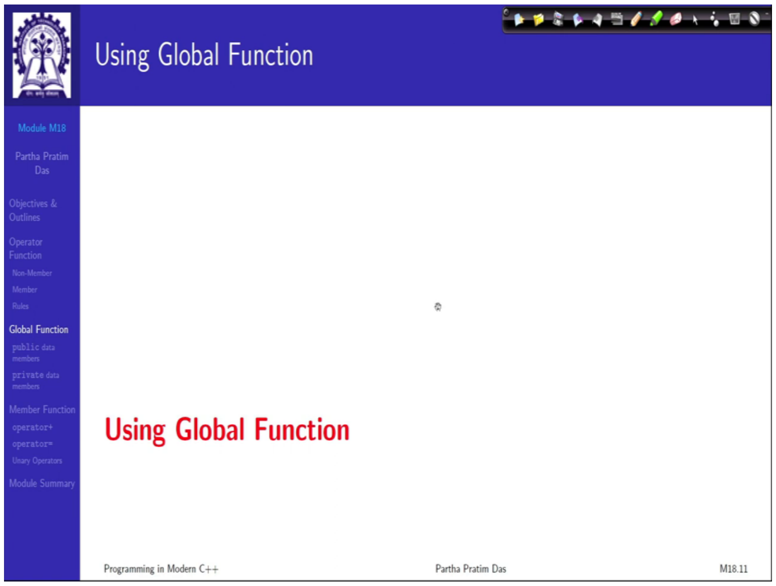
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) *cannot be overloaded*
- The overloads of operators `&&`, `||`, and `,` (comma) *lose their special properties*: short-circuit evaluation and sequencing
- 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

Programming in Modern C++ Partha Pratim Das M18.10

So, operator overloading rules we have discussed you cannot define a new operator, you cannot change the arity precedence or associativity of any operator and there is a list of operators that are available for overloading quite a few of them. Some operators like scope resolution, class membership, member access through pointer, size of question mark column, the ternary operator cannot be overloaded.

Some has difference in behaviour if you overload in terms of shortcutting or shortcut evaluation or sequencing, not working for them and so on so forth we have discussed this I have included here just for completeness, if you have doubts, go back to module 9 and listen to that part of the discussion.

(Refer Slide Time: 12:15)



Using Global Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

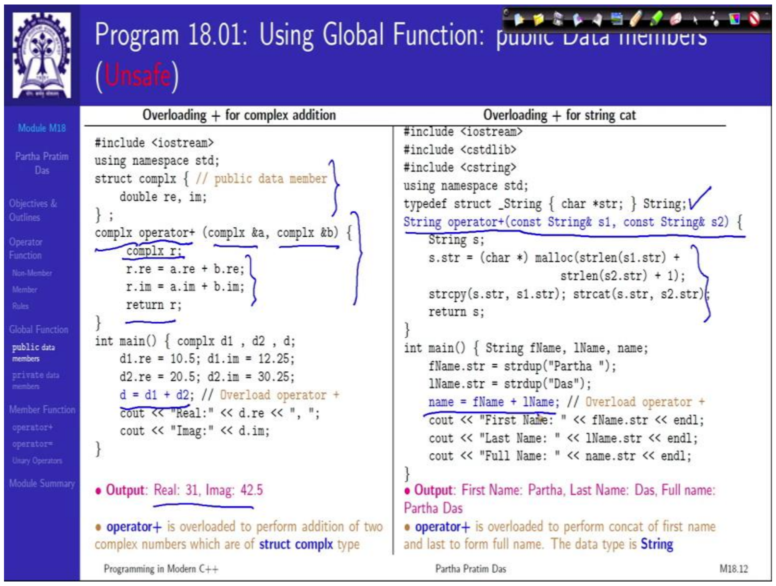
operator=

Unary Operators

Module Summary

Using Global Function

Programming in Modern C++ Partha Pratim Das M18.11



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

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

Overloading + for complex addition

```
#include <iostream>
using namespace std;
struct complx { // public data member
    double re, 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; // Overload operator +
    cout << "Real:" << d.re << ", ";
    cout << "Imag:" << d.im;
}
```

- **Output**: Real: 31, Imag: 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 <cstdlib>
#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;
}
```

- **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 **String**

Programming in Modern C++ Partha Pratim Das M18.12

Now, let us see how do we overload operators using global function. So, here two examples on left we are trying to overload for adding to complex number on right we are trying to overload for concatenating two strings. So, for simplicity I am using struct here, because struct as you know is a class in C++ with where everything is public.

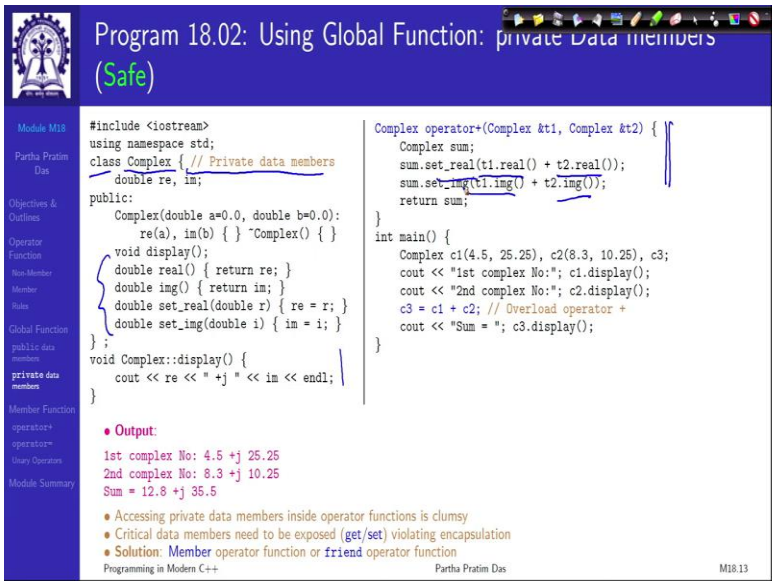
So, it has two data members which are necessarily public I define a global function operator plus it is a global function not defined within struct, which takes two complex numbers does the addition into a local temporary complex number and returns that. The return has to be by value, because for two reasons, one is you know, that local objects cannot be should not should never be returned as a reference, more importantly, it has to be a value because it is something new that is happening a and b existed, but the result the r did not exist. So, it will have to be a new object.

So, with that, you can write the addition of complex exactly in the same way as you write the addition of two integers and you will get that. Similar thing I can do for a string type where the typical c string character pointer is wrapped in the operator plus the global function overloading the concatenation, I do the standard concatenation code.

So that now I can easily write a fName + lName where these two are two strings to actually concatenate them and get the result. So, that is what I was meaning that these are the ways to kind of write them as an algebraic form than doing several function calls. We have seen these examples in a certain way before and we have seen the I mean kind of what are the limitations of this semantics and so on.

What I want to point out is certainly here, it is a global function and naturally it has to use public data members. It is using public data members and the moment we do that no encapsulation, it is unsafe.

(Refer Slide Time: 14:55)



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

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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 << " +j " << im << endl;
}
```

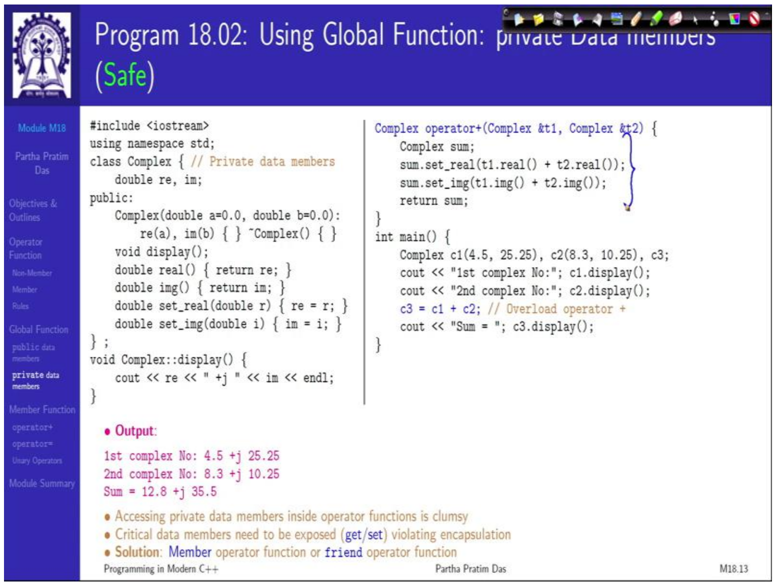
```
Complex operator+(Complex &t1, Complex &t2) {
    Complex sum;
    sum.set_real(t1.real() + t2.real());
    sum.set_img(t1.img() + t2.img());
    return sum;
}
int main() {
    Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
    cout << "1st complex No:"; c1.display();
    cout << "2nd complex No:";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

- **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

Programming in Modern C++ Partha Pratim Das M18.13



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

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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 << " +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; // Overload operator +
    cout << "Sum = "; c3.display();
}
```

- **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

Programming in Modern C++ Partha Pratim Das M18.13

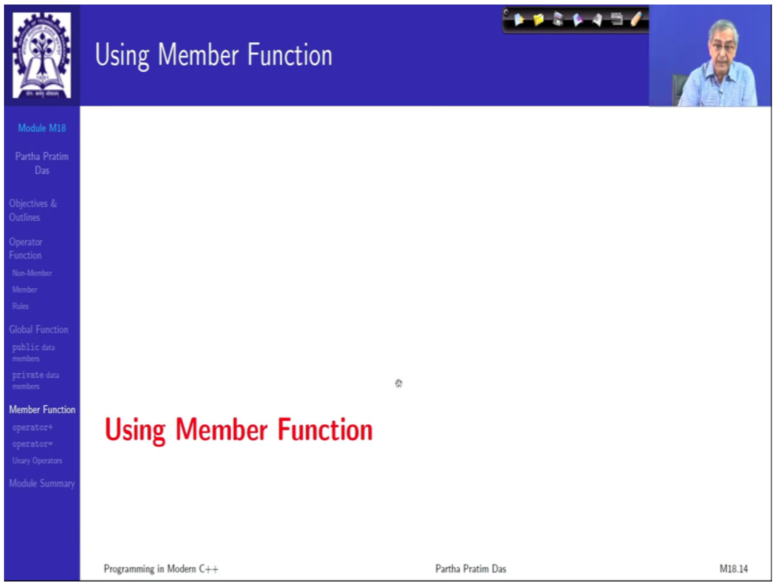
So, you know how to take care of this. Say, for the case of complex numbers, I can easily take care of this by making these data members private. I call it class now, make this private and provide the get set functions on the members as an interface. So, this a standard you know information hiding paradigm we have done and then define this define this operator plus as a global function.

So, as a global function, it has to use these get set functions to actually be able to define the operator plus good enough. The problem is it really gets clumsy if your operator function code is long, it gets really gets clumsy with this get set functions and so on. So, you can work out of that.

So, what is the problem that we are getting into we have a function global function, which is our operator function.

It needs to use the internal state of the object to be able to define that operator function and being a global function, it cannot do that. If you put the variables put those data members in public you have information leakage, it is not safe. If you put them into private, then you can follow the information hiding paradigm, but you have the difficulty that you have to do I mean, if you can see, how cluttered this code is becoming with the real real mg mg this that, lot of gets set calls.

(Refer Slide Time: 16:43)



Using Member Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

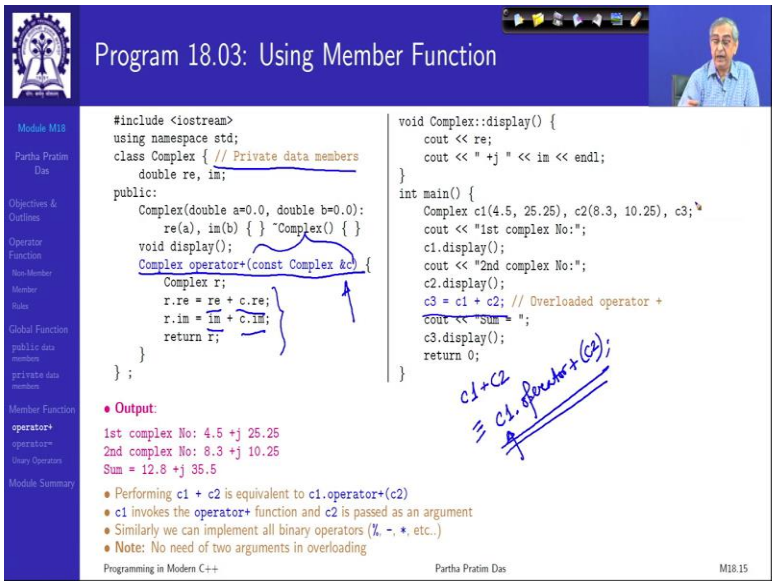
operator=

Unary Operators

Module Summary

Using Member Function

Programming in Modern C++ Partha Pratim Das M18.14



Program 18.03: Using Member Function

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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; // Overloaded operator +
    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

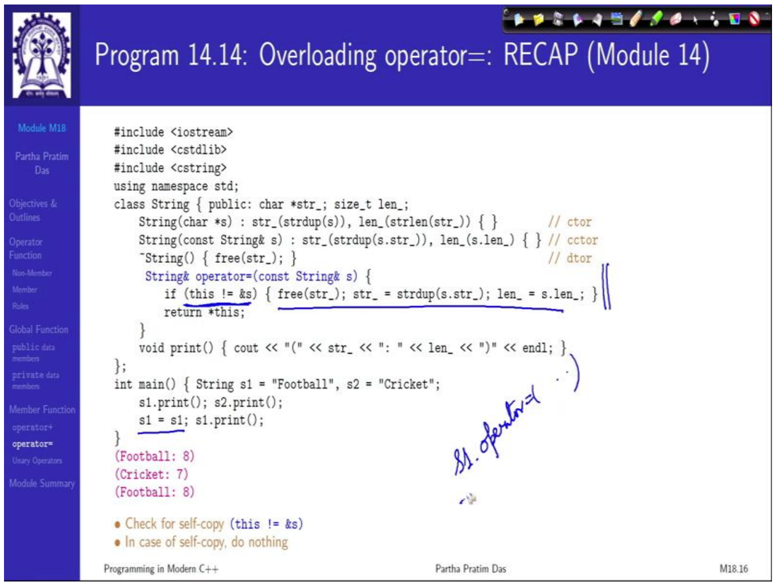
Programming in Modern C++ Partha Pratim Das M18.15

So, if we instead of using global function, if we use member functions, then certainly we can do easier things than this. So, let me do that same thing the private data members, but I have moved the overloaded operator plus inside the class. As I move it inside the class, I need only the right parameter, the left parameter is object on which this is getting called because, when I write it like this, it means c1 + c2 means c1.operator+(c2) this is what it actually means.

Since it means this, so, this is where c2 will come and c1 is the left hand side operand, which is the object on which the call is being met. So, these are components of c1 the current object, if you want to be if you want to make it very explicit, you can write them as this pointer re and this is these are component from the right hand side.

There is no issue of accessing these components because it is a member function now. So, with this basic problem of safety and also the clumsiness that we got in terms of using the global function is removed, I can very easily write this and write all sorts of operators for complex and have a very nice complex data type created in parallel to the built in types.

(Refer Slide Time: 18:34)



Program 14.14: Overloading operator=: RECAP (Module 14)

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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();
}
(Football: 8)
(Cricket: 7)
(Football: 8)
```

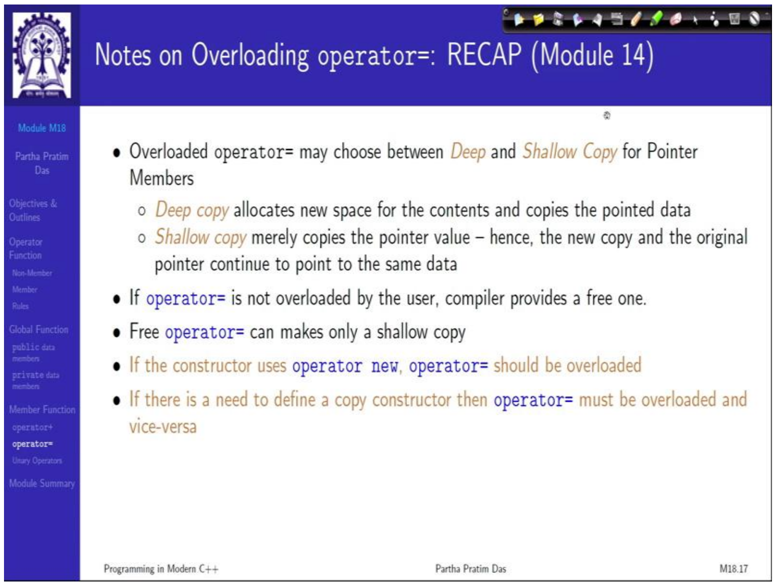
s1.operator=(. .)

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

Programming in Modern C++ Partha Pratim Das M18.16

Similarly, we have we studied about the assignment operator, this is the overloading of the assignment operator we studied that self copy needs to be guarded. So, if it is an assignment to itself, then you do not do it do nothing and this is what ensures that you do a deep copy that is if you the pointer you do a deep copy. So, when you do this assignment, the assignment like this, then you will have the function called as s1 dot operator assignment like this. We have seen all of these.

(Refer Slide Time: 19:22)



Notes on Overloading operator=: RECAP (Module 14)

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

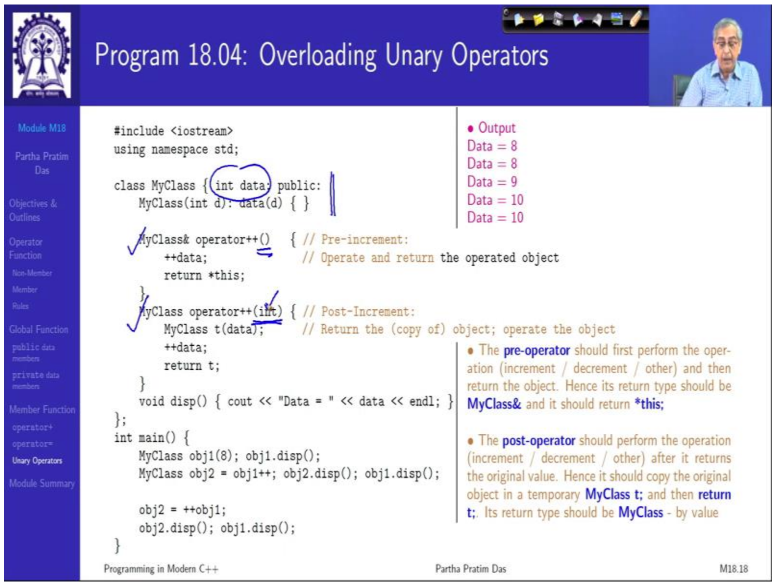
Module Summary

- 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

Programming in Modern C++ Partha Pratim Das M18.17

And we also noted that there is a difference between deep copy and shallow copy and if it is operated assignment if it is not overloaded by the user, the compiler provides a free one which does a shallow copy. Please remember that if the operate constructor uses operator new, then operator assignment should be overloaded. You must provide your version. Similarly, if the copy constructor, you need to define a copy constructor operator assignment must be overloaded and vice versa. Basic rules we have already seen.

(Refer Slide Time: 20:03)



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();
}
```

- Output

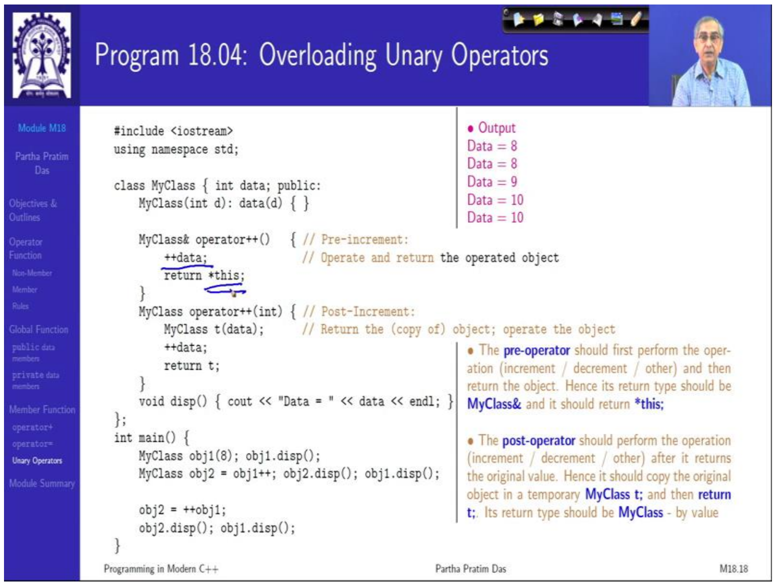
```
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10
```

- 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** - by value

Programming in Modern C++ Partha Pratim Das M18.18

Let us, move on to overloading the unary operators. So, I have a MyClass which has only one int data member. So, it is kind of a wrapper of an integer which is private, I want to define pre increment and post increment operators for that. So, for pre increment operator, I have no argument, because I am working on the object only and for the post argument, just to make it syntactically distinguishable from the previous argument, I have a dummy int parameter for which I will actually not pass any parameter and it is acceptable.

(Refer Slide Time: 20:59)



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();
}
```

- Output

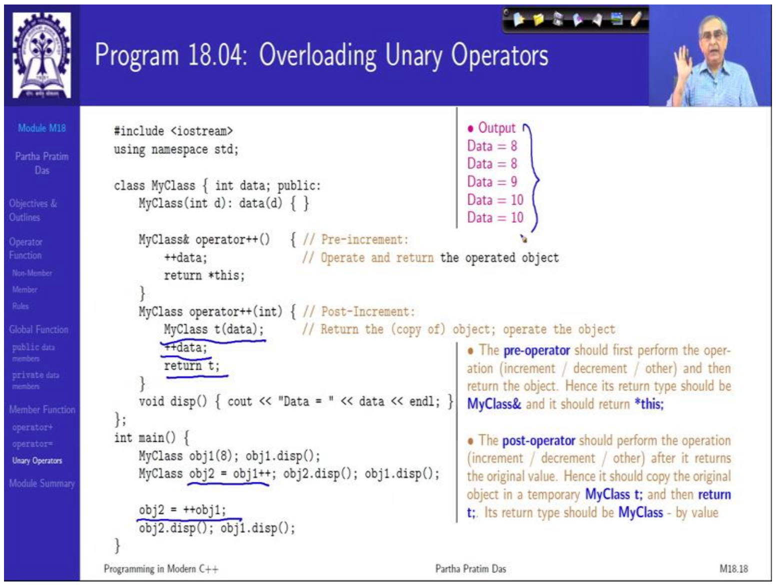
```
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10
```

- 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** - by value

Programming in Modern C++ Partha Pratim Das M18.18

Now, this what is this is pre increment, so it will first do the increment and return me the incremented value. So, as it does the increment, what should it return? It the incremented value that is that is the incremented object itself. So, I have to return start this much in the way we did in case of operator assignment.

(Refer Slide Time: 21:39)



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();
}
```

- Output

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

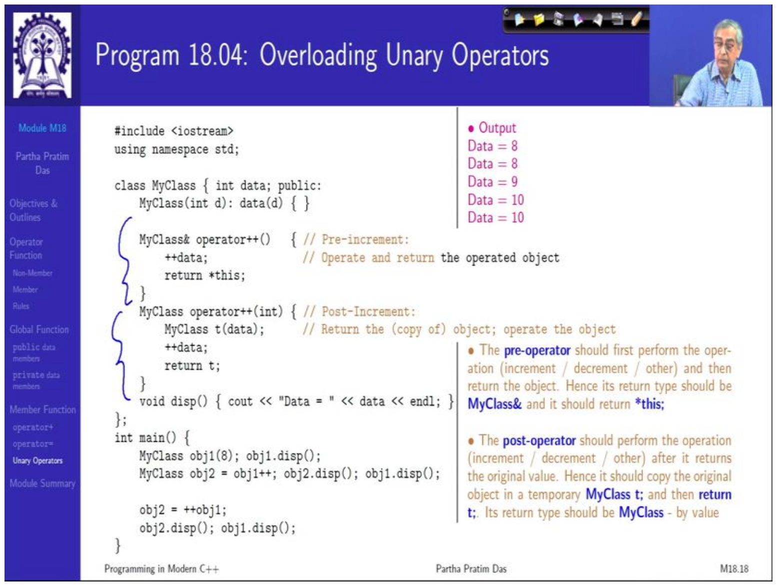
- The **pre-operator** should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should return ***this;**
- The **post-operator** should perform the operation (increment / decrement / other) after it returns the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

Programming in Modern C++ Partha Pratim Das M18.18

What if I do a post increment? What do I want? I want the current object to change be incremented, but the object returned must be the old object. So, I have to remember the old object, this is where I remember the old object I use a, I will need a copy constructor for that. So that t becomes my old objects, then I increment data, so my current object is incremented and then I return that object.

Here, I am not returning the object itself, because object itself has changed, but you want the that change to take effect separately and you want the original value back. So, if we do this in case of here, and here the post increment and pre increment, you will see the effects taking place. That is straightforward, that is there is nothing no specific fun in that you could have done it with int.

(Refer Slide Time: 22:39)



Program 18.04: Overloading Unary Operators

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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();
}
```

- Output

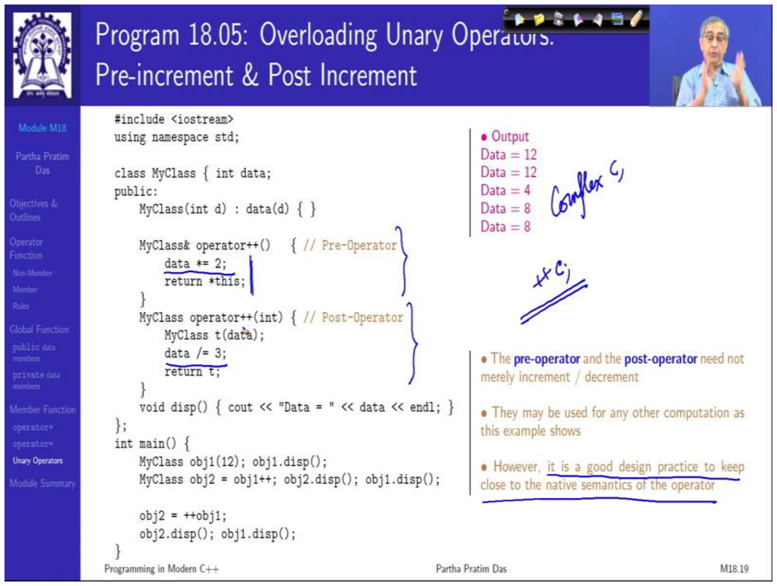
```
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10
```

- 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** - by value

Programming in Modern C++ Partha Pratim Das M18.18

But the what we want to really learn is how to write this pre increment and post increment operators.

(Refer Slide Time: 22:53)



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

Module M18

Partha Pratim Das

Objectives & Outlines

Operator Function

Non-Member

Member

Rules

Global Function

public data members

private data members

Member Function

operator+

operator=

Unary Operators

Module Summary

```
#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();
}
```

- Output

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

Complex c;

++c;

- 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 practice to keep close to the native semantics of the operator

Programming in Modern C++ Partha Pratim Das M18.19

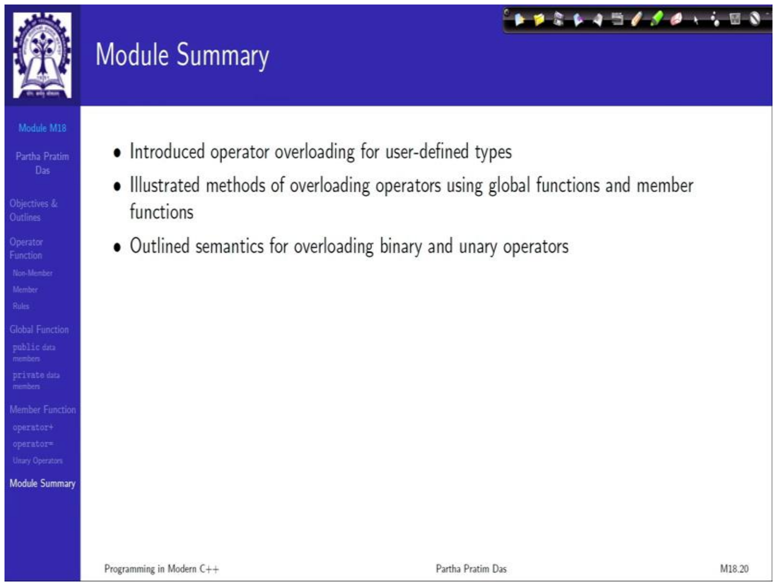
Now, while we call them as pre increment post increment operator, it is not necessary that when you overload you will have to do incremental you can do something else. For example, here I have a pre operator ++ where I have doubled my data. Here I have a post operator ++ where I have divided the data by 3 anything I can do, that is up to me as to what I want.

So, when I do that, it becomes a map to that particular operation, but it is usually a good design practice that you keep the semantics of the object a semantics of the type as close to the built in type as possible. So, if I have a pre increment operator, which actually makes it double for a complex, then it probably is not something which is very very good.

For example, if for a rational if I want to do a pre increment, that would mean that I add 1 to that fraction and return that value. If I do a post increment, it will mean that I return that value and then add 1 to that function and so on so forth. But it is possible I mean programming wise it is possible to do this design wise you may be more choosy as to, because what happened, what will happen is, is if I do this for a say my complex type, I have complex C, and then I write ++ C, anybody who reads the code would instinctively think that this is incrementing the complex number.

But you are actually maybe doubling the components or doing something. So, though you are building up an algebra, that algebra deviates from the normal semantics that most of us understand. So, that deviation makes the programme much less readable and confusing, and potential for error. So, these are but this is design choice, I am telling you get, programming wise you can write anything that you want to write in terms of these unary operators.

(Refer Slide Time: 25:26)



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

So, that brings us to the end of the first part of operator overloading, where we have seen how I can overload operators, recapitulating on what we did on module 9 and module 14 and noted that

I can do it as a global function and I can do it as a member function as well. We will going forward in the next module we will see how we can make use of the friend function and how we can do some very useful overloading particularly for the input output operator. Thank you very much for your attention. See you in the next module.