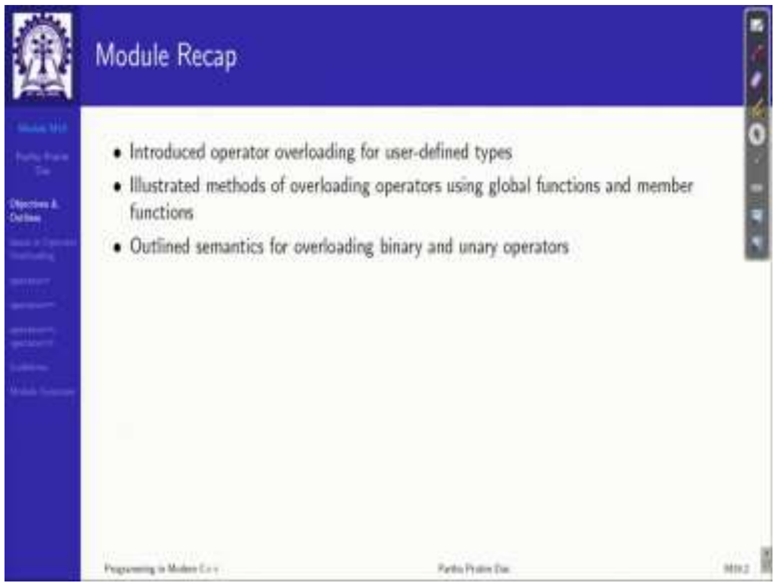
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecturer 19
Overloading Operator for User-Defined Types - Part 2

Welcome to programming in modern C++. We are in week 4 and we are going to discuss module 19.

(Refer Slide Time: 00:43)

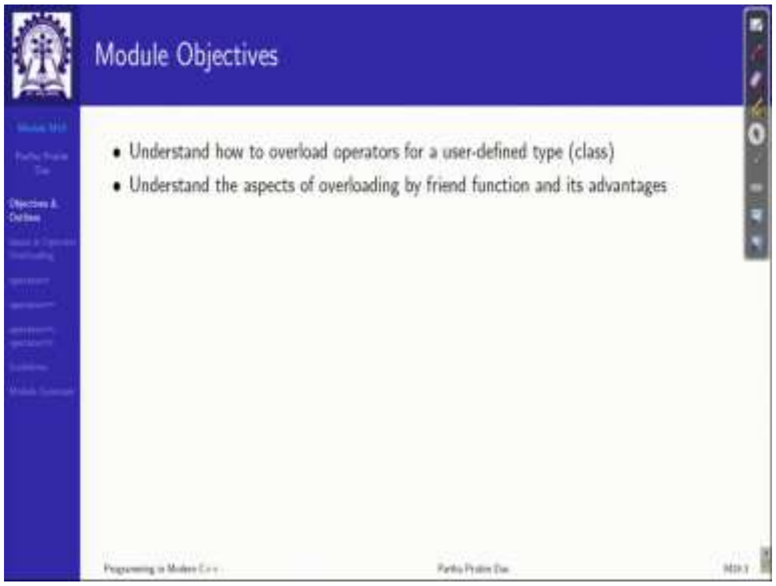


Module Recap

- 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

In the last module 18, we started discussing about operator overloading, that is how to define operator functions for user defined types so that we can build complete algebra with our user defined types much in the same way the algebra is available for built in types like int like double and so on. We have looked at overloading operators using global functions and member functions of classes and we have also outlined the semantics for overloading binary and unary operators.

(Refer Slide Time: 01:26)

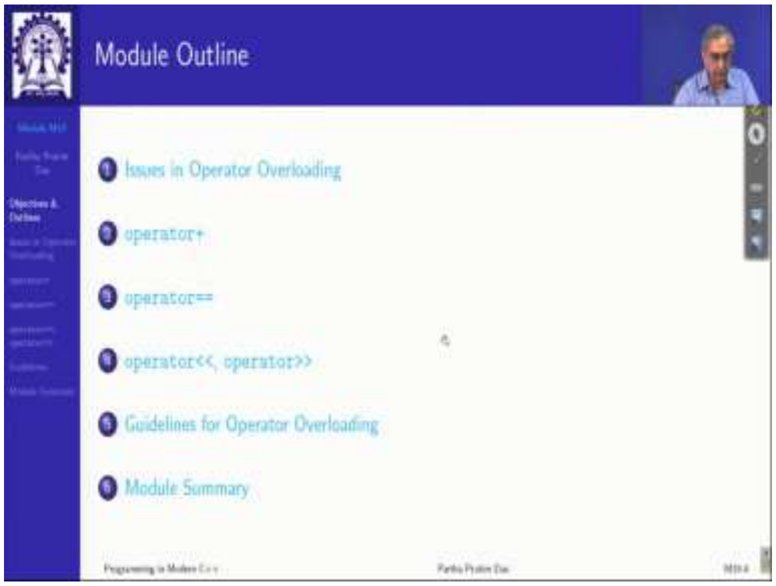


Module Objectives

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

Building a further on that, we will now try to understand overloading all types of operators for a user defined type or a class and particularly focus on the aspect of overloading by friend function and what advantages does that specifically provide.

(Refer Slide Time: 01:45)

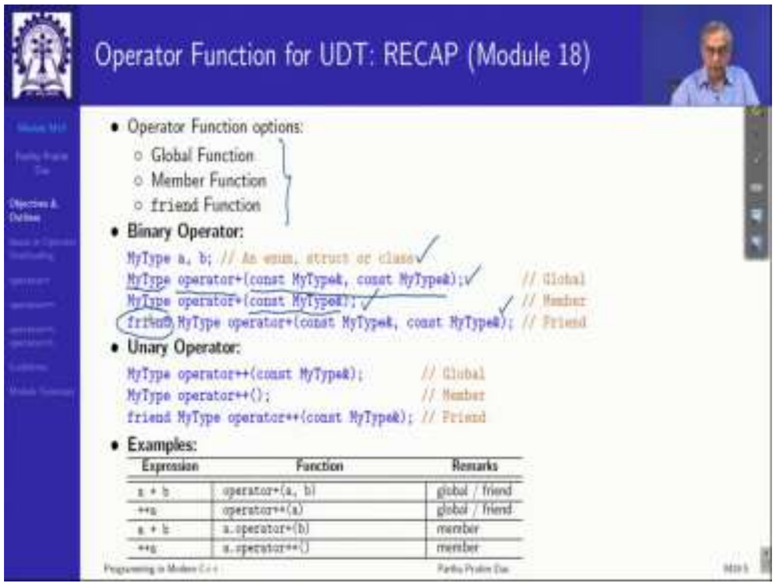


Module Outline

1. Issues in Operator Overloading
2. operator+
3. operator==
4. operator<<, operator>>
5. Guidelines for Operator Overloading
6. Module Summary

This is the outline will be available on your left panel.

(Refer Slide Time: 01:51)



Operator Function for UDT: RECAP (Module 18)

- Operator Function options:
  - Global Function
  - Member Function
  - friend Function
- **Binary Operator:**

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

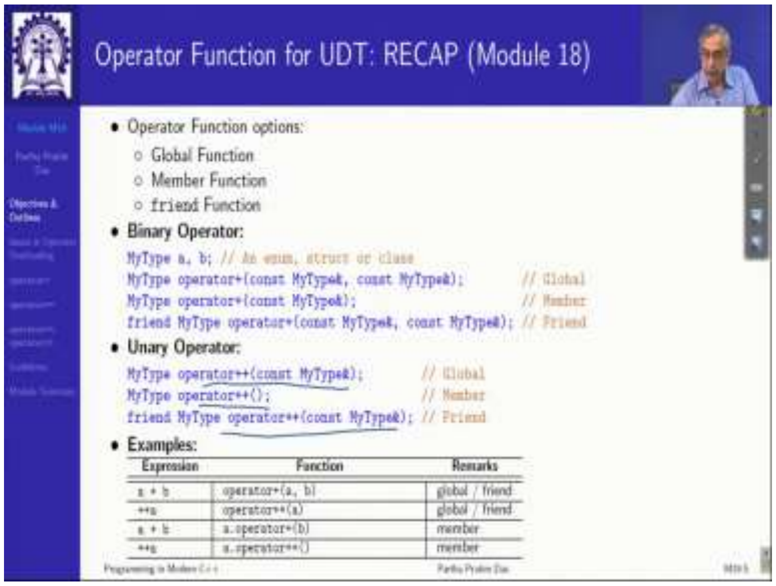
- **Unary Operator:**

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

- **Examples:**

Expression	Function	Remarks
a + b	operator+(a, b)	global / friend
++a	operator++(a)	global / friend
a + b	a.operator+(b)	member
++a	a.operator++()	member

Programming in Modern C++ Partha Pratim Das



Operator Function for UDT: RECAP (Module 18)

- Operator Function options:
  - Global Function
  - Member Function
  - friend Function
- **Binary Operator:**

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

- **Unary Operator:**

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

- **Examples:**

Expression	Function	Remarks
a + b	operator+(a, b)	global / friend
++a	operator++(a)	global / friend
a + b	a.operator+(b)	member
++a	a.operator++()	member

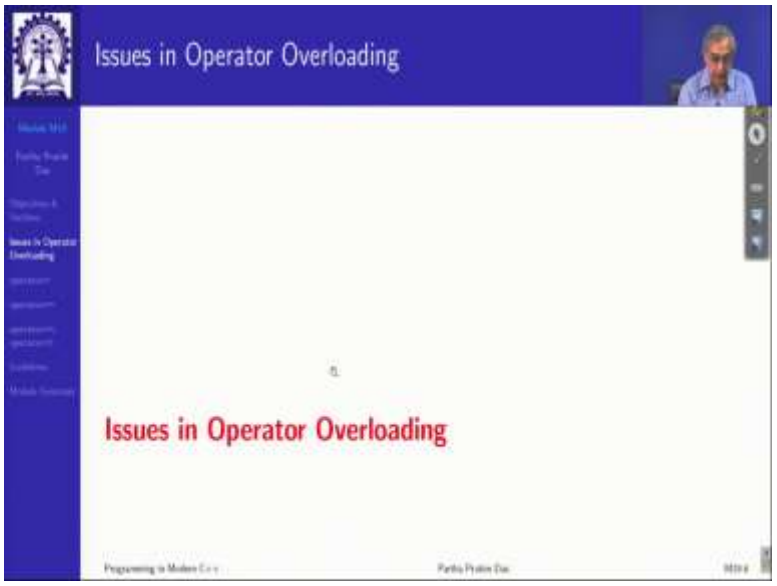
Programming in Modern C++ Partha Pratim Das

So, a quick recap from the last module, we have three options to overload using global function, using member function or using friend function. So, if a and b are of MyType or MyType is sum, enum, struct or class type, then I could write the operator function say for operator plus, which takes two parameters left and right and returns me the final result.

I can use a member function of the MyType class and in that case, my operated function takes only one parameter which is the right argument because the left argument is the implicit parameter which is the object on which this operator is being invoked. Doing it with a friend function is very similar because it is like just like another global function, but it is prefixed with the keyword friend and its prototype is declared within the scope of the class declaring that this is the friend function.

This is what holds for the binary operators for unary operator similarly, I can have global version with one parameter, member function version with no parameter or the friend version, which is also with one parameter. What is to be noted here, which we saw at that time is the fact that if the unary operator is prefix or postfix, if it is postfix particularly, we will need to pass a we will need to declare it not pass declare it with a dummy int parameter.

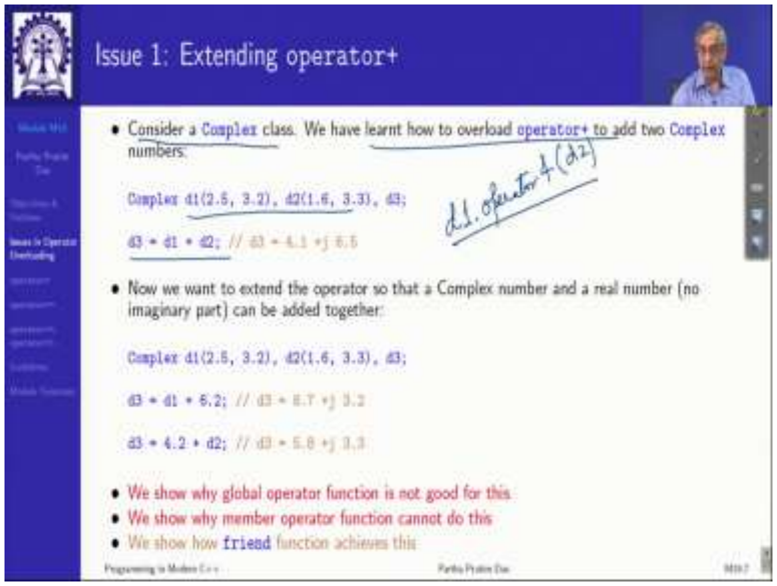
(Refer Slide Time: 03:47)



Issues in Operator Overloading

Issues in Operator Overloading

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



Issue 1: Extending operator+

- Consider a Complex class. We have learnt how to overload operator+ to add two Complex numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + d2; // d3 = 4.1 +j 6.5
```

$d1.operator+(d2)$

- Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

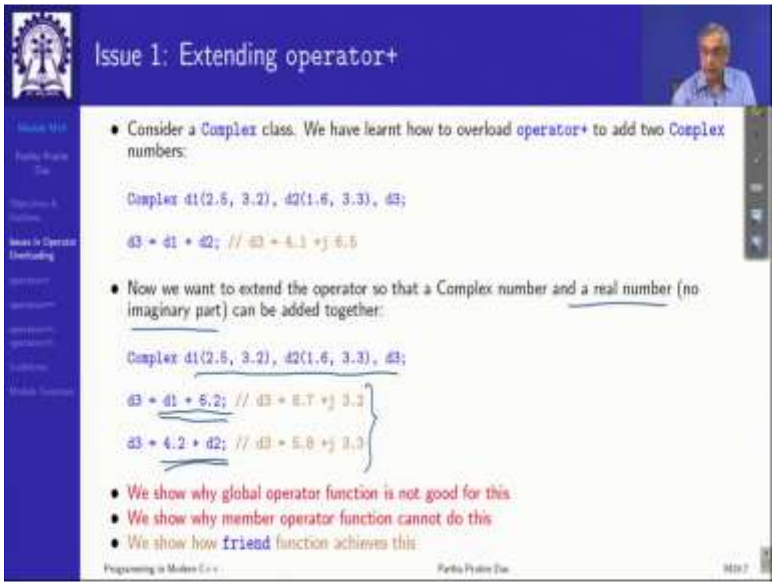
```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + 6.2; // d3 = 8.7 +j 3.2

d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how friend function achieves this

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

So, equipped with this, let us now, look into the issues in operator overloading and we will present two specific cases where we have issues. So, consider a Complex class and we have learned how to overload the operator for two complex numbers. So, we have two complex numbers we can overload it like this to get the result. So, actually what we get is d1.operator+ and that takes d2 or it is called as a global function whatever we it is it will work.

(Refer Slide Time: 04:32)



Issue 1: Extending operator+

- Consider a Complex class. We have learnt how to overload operator+ to add two Complex numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + d2; // d3 = 4.1 +j 6.5
```

- Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

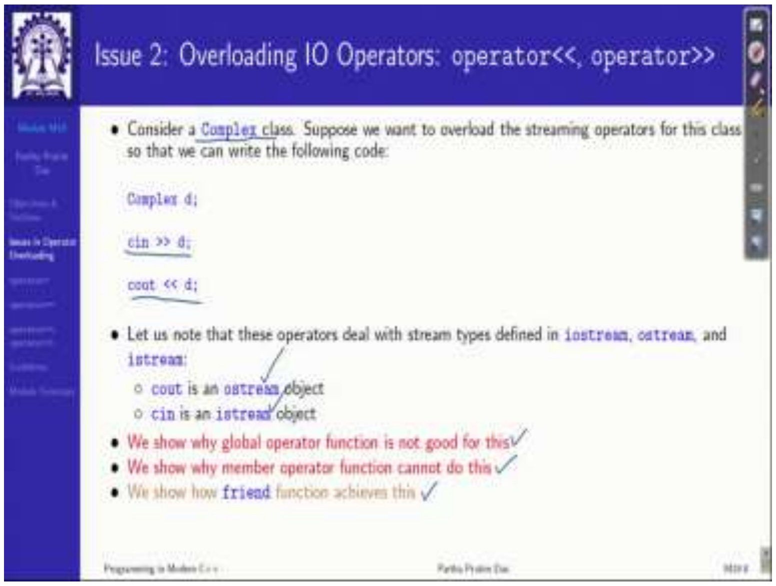
```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;

d3 = d1 + 6.2; // d3 = 8.7 +j 3.2

d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

- We show why global operator function is not good for this
- We show why member operator function cannot do this
- We show how friend function achieves this

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

Now, suppose we want to extend that operator, so, that the complex number can be added to a real number also that is it which has no imaginary part. So, if we have these complex numbers, I want to make this addition that is d1 plus 6.2 which actually means 6.2 plus j0 or 4.2 plus d2 which means 4.2 plus j0 to be added to d2, if I want these also to be valid for my operator plus. Now, we have to see why global function is not good for this and why member operator function cannot also do this and only friend function can achieve this. So, this is the context in terms of which we are trying to refine the operator function.

(Refer Slide Time: 05:27)



Issue 2: Overloading IO Operators: operator<<, operator>>

- Consider a Complex class. Suppose we want to overload the streaming operators for this class so that we can write the following code:

```
Complex d;

cin >> d;

cout << d;
```

- Let us note that these operators deal with stream types defined in iostream, ostream, and istream:
  - cout is an ostream object
  - cin is an istream object
- We show why global operator function is not good for this ✓
- We show why member operator function cannot do this ✓
- We show how friend function achieves this ✓

Programming in Modern C++ Partha Pratim Das

The other context that we will deal with is of the io operators, streaming operators. So, we have a Complex class. So, I want to write these kinds of streaming expressions to print or read the complex values. So, we know that cout is the ostream object, we know cin is an istream object predefined.

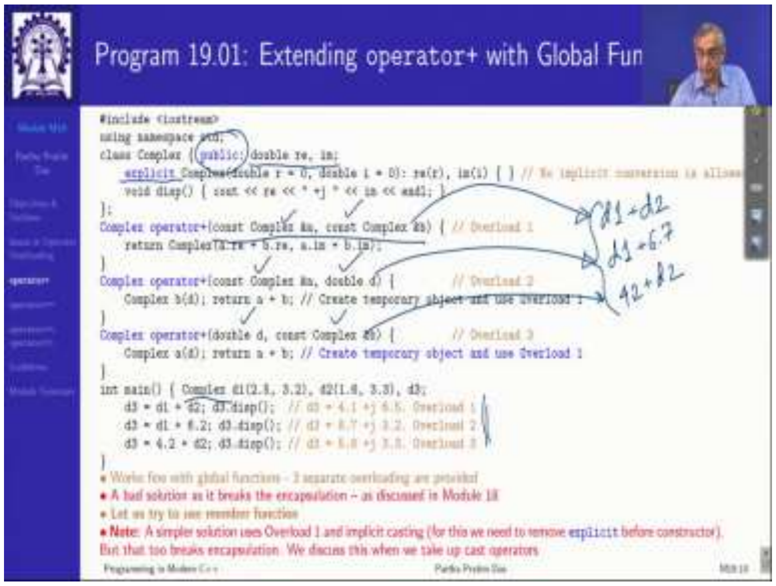
Again, we show why global function is not good for this purpose, why member function also cannot achieve this task and we finally show that how friend function will provide a solution. So, two different contexts that we are highlighting here where our the solution we have looked at so, far is not going to extend and we will see the specific difficulties that we will come up with.

(Refer Slide Time: 06:31)

Extending operator+

Extending operator+

Programming in Modern C++ Partha Pratim Das M19.9



Program 19.01: Extending operator+ with Global Fun

```
#include <iostream>
using namespace std;
class Complex { public: double re, im;
    explicit Complex(double r = 0, double i = 0): re(r), im(i) { } // No implicit conversion is allowed
    void disp() { cout << re << " +j " << im << endl; }
};
Complex operator+(const Complex &a, const Complex &b) { // Overload 1
    return Complex(a.re + b.re, a.im + b.im);
}
Complex operator+(const Complex &a, double d) {       // Overload 2
    Complex b(d); return a + b; // Create temporary object and use Overload 1
}
Complex operator+(double d, const Complex &b) {       // Overload 3
    Complex a(d); return a + b; // Create temporary object and use Overload 1
}
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

d1 + d2
d1 + 6.7
42 + d2

- Works fine with global functions - 3 separate overloading are provided
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- Let us try to use member function
- **Note:** A simpler solution uses Overload 1 and implicit casting (for this we need to remove explicit before constructor). But that too breaks encapsulation. We discuss this when we take up cast operators

Programming in Modern C++ Partha Pratim Das M19.10

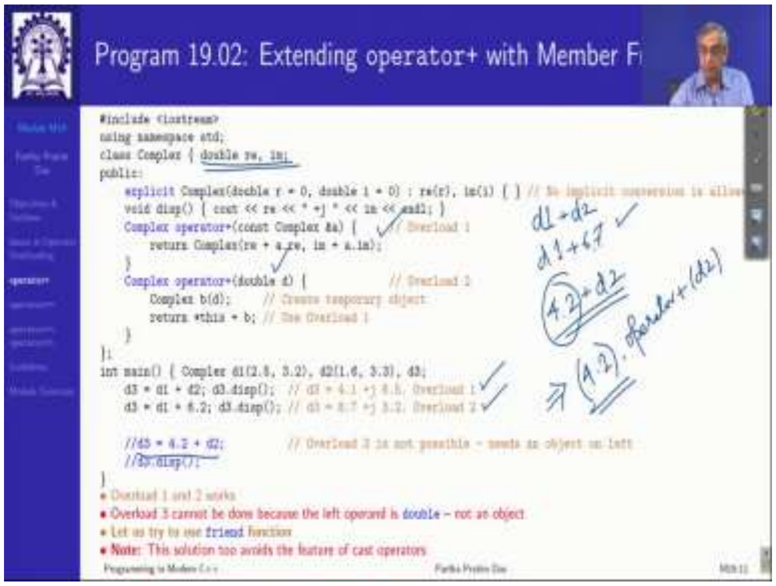
So, first let us try to address the first issue try to extend the operator plus. So, if I have to extend this, then what I can do is I can define using global function. So, since we are looking at three forms, that is d1 plus d2 is one form, d1 plus say 6.7 is another form, 4.2 plus d2 is another form. So, since we are looking at three forms, and you can see that they have different types of the parameters.

So, we can overload this operator plus thrice one for a pair of Complex which will work for this, one for a Complex and a double which will work for this, and one for a double and a Complex which will work for this. This solution apparently will not have any difficulties and you can try out with these examples and it will work fine.

The only difficulty that arise is with the fact that being global function, this will not be able to access the data members of this class. So, I have to make the data members public the moment I do that, for do it this operator overloading I have to break the encapsulation that I have created with so much of care. So, it is this is not a very good approach.

And we have also provided explicit we will I will talk about explicit explicitly later on, but all that it means that if you are constructed is explicit then it has to be directly called either through this or through new it cannot be used implicitly for doing a conversion. So, the global function cannot work because it is breaking the encapsulation, so, that the whole paradigm falls apart.

(Refer Slide Time: 08:34)



Program 19.02: Extending operator+ with Member Fi

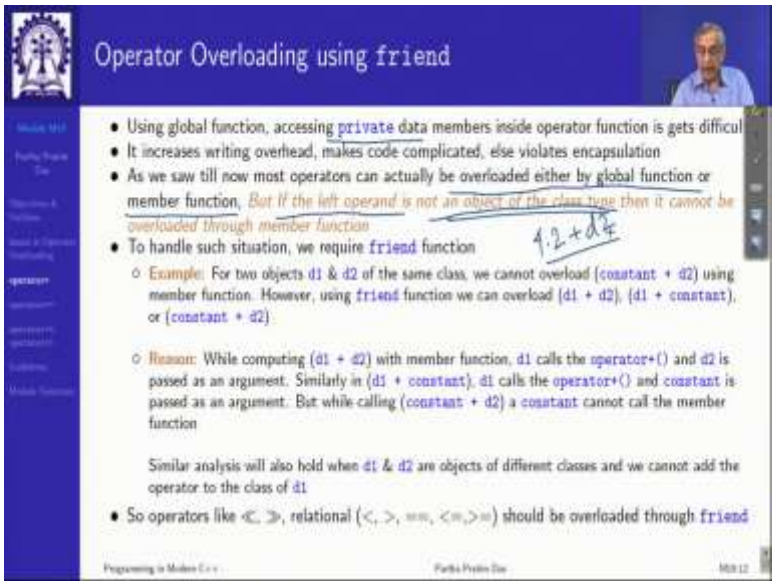
```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allo
    void disp() { cout << re << " +j " << im << endl; }
    Complex operator+(const Complex &a) {         // Overload 1
        return Complex(re + a.re, im + a.im);
    }
    Complex operator+(double d) {                  // Overload 2
        Complex b(d);       // Create temporary object
        return *this + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();   // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2

    //d3 = 4.2 + d2;              // Overload 3 is not possible - needs an object on left
    //d3.disp();
}
```

- Overload 1 and 2 works
- Overload 3 cannot be done because the left operand is double – not an object
- Let us try to use friend function
- Note: This solution too avoids the feature of cast operators

So, let us look at member function. So, as we look at member function, then I have one member function, which takes a Complex takes care of d1 plus d2. I have another member function which takes a double which takes care of d1 plus 6.7. So, the first expression works the second expression works, but how about the third one. If I have 4.2 plus d2, I cannot write a member function to do this because a member function always binds with the left hand side parameter left parameter.

So, 4.2 plus d2 basically in member function notation means 4.2.operator+(d2) which means that operator plus has to be a member function of the double type and you know built in types like double cannot let their operators be overloaded. So, this is not going to, this is going to partly work for this, this of course restores the this restores the encapsulation, the data members are back to private, but this does not solve the problem because the third form of the expression is not permissible.

(Refer Slide Time: 09:59)



Operator Overloading using friend

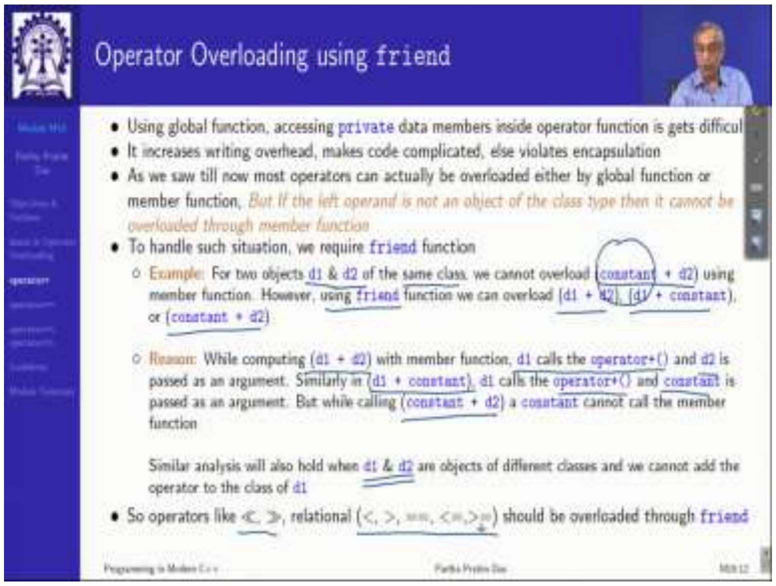
- Using global function, accessing private data members inside operator function is gets difficul
- It increases writing overhead, makes code complicated, else violates encapsulation
- As we saw till now most operators can actually be overloaded either by global function or member function, *But if the left operand is not an object of the class type then it cannot be overloaded through member function*
- To handle such situation, we require friend function
  - Example: For two objects d1 & d2 of the same class, we cannot overload (constant + d2) using member function. However, using friend function we can overload (d1 + d2), (d1 + constant), or (constant + d2)
  - Reason: While computing (d1 + d2) with member function, d1 calls the operator+() and d2 is passed as an argument. Similarly in (d1 + constant), d1 calls the operator+() and constant is passed as an argument. But while calling (constant + d2) a constant cannot call the member function

    Similar analysis will also hold when d1 & d2 are objects of different classes and we cannot add the operator to the class of d1
- So operators like ≪, ≫, relational (<, >, ==, <=, >=) should be overloaded through friend

So, let us look at using a friend function. So, how does friend help friend on one side is like a global function in terms of function call and all that and it has the ability to access private data member of the class because it is a friend. So, while writing global if I want to protect the encapsulation and use a global function for working on the class, then I will have to give get into all that get set mechanism lot of extra code has to be written, but with friend function, I do not need to do that.

Now, the operators can actually be overloaded either by a global function or by a member function is what we have seen, but, the question is if the left operand is not an object of the class type, then it cannot be overloaded through member function that is precisely the case we are getting into when we have expression like 4.2 plus d2. So, to handle such situation, we can make use of the friend.

(Refer Slide Time: 11:14)



Operator Overloading using friend

- Using global function, accessing private data members inside operator function is gets difficul
- It increases writing overhead, makes code complicated, else violates encapsulation
- As we saw till now most operators can actually be overloaded either by global function or member function, *But If the left operand is not an object of the class type then it cannot be overloaded through member function*
- To handle such situation, we require friend function
  - Example: For two objects d1 & d2 of the same class, we cannot overload (constant + d2) using member function. However, using friend function we can overload (d1 + d2), (d1 + constant), or (constant + d2)
  - Reason: While computing (d1 + d2) with member function, d1 calls the operator+() and d2 is passed as an argument. Similarly in (d1 + constant), d1 calls the operator+() and constant is passed as an argument. But while calling (constant + d2) a constant cannot call the member function

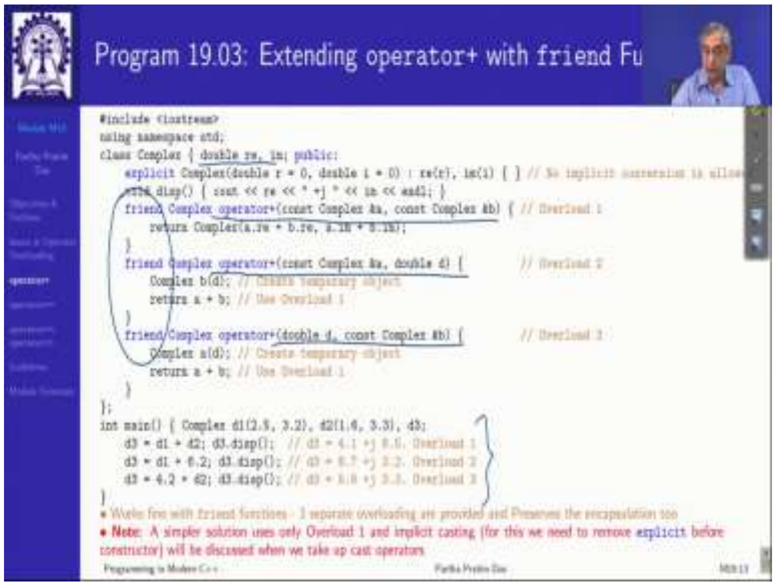
    Similar analysis will also hold when d1 & d2 are objects of different classes and we cannot add the operator to the class of d1
- So operators like ≪, ≫, relational (<, >, ==, <=, >=) should be overloaded through friend

Programming in Modern C++ Partha Pratim Das M09.12

Now, let us say there are two objects d1 and d2 of the same class we cannot overload constant plus d2 using the member function. However, using friend I can do all three of them, because I could do them with global function, friend is just I mean in this case it is like the global except that it does not break the encapsulation for everybody. So, the reason is while computing d1 plus d2 the member function d1 with member function, if I tried to do with member function d1 calls operator plus and d2 is an argument.

Similarly, for d1 plus constant we have seen that, it calls operator plus and constant is a second argument, but when we have this, we cannot do it because the constant cannot call a member function. So, similar analysis will also hold not only when this other mixed parameter is a constant, but when it is an object of a different class, which is what is happening in case of the iostream also we will come to that. So, operators like streaming like relational should be overloaded using the friend function.

(Refer Slide Time: 12:32)



Program 19.03: Extending operator+ with friend Fu

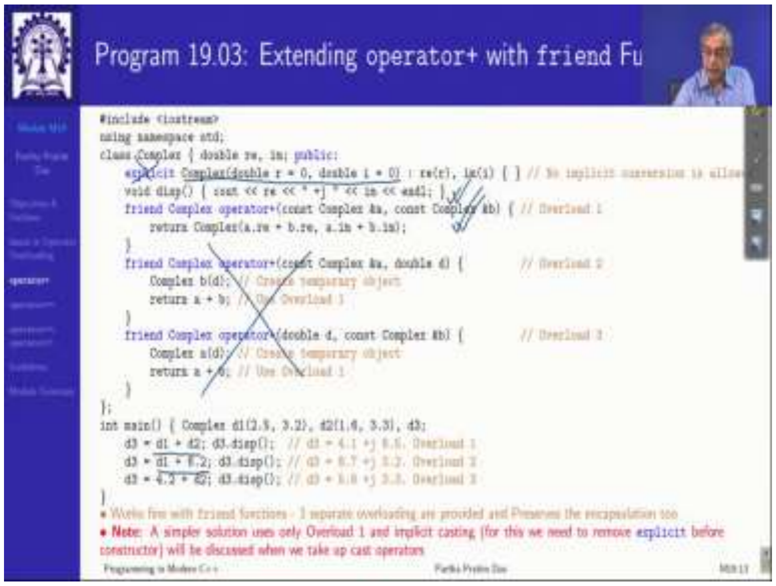
```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allow
    void disp() { cout << re << " +j " << im << endl; }
    friend Complex operator+(const Complex &a, const Complex &b) { // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
    }
    friend Complex operator+(const Complex &a, double d) {        // Overload 2
        Complex b(d); // Create temporary object
        return a + b; // Use Overload 1
    }
    friend Complex operator+(double d, const Complex &b) {        // Overload 3
        Complex a(d); // Create temporary object
        return a + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- Works fine with friend functions - 3 separate overloading are provided and Preserves the encapsulation too
- **Note:** A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before constructor) will be discussed when we take up cast operators

Programming in Modern C++ Partha Pratim Das M19.13

So, let us try to do that for operator plus. So, now, what we do is, we have again three overloads Complex Complex, Complex double, double Complex instead of global they are all friend function the encapsulation is perfectly maintained and all three become correct.

(Refer Slide Time: 13:05)

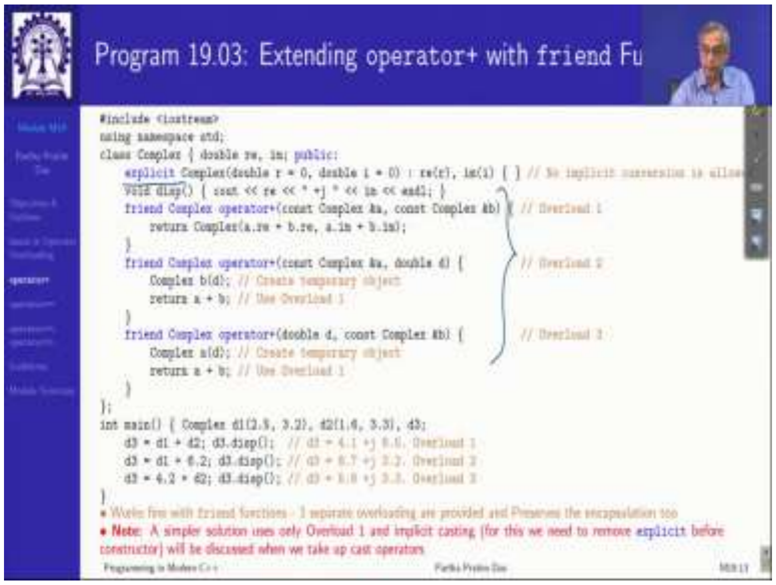


Program 19.03: Extending operator+ with friend Fu

```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allow
    void disp() { cout << re << " +j " << im << endl; }
    friend Complex operator+(const Complex &a, const Complex &b) { // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
    }
    friend Complex operator+(const Complex &a, double d) {        // Overload 2
        Complex b(d); // Create temporary object
        return a + b; // Use Overload 1
    }
    friend Complex operator+(double d, const Complex &b) {        // Overload 3
        Complex a(d); // Create temporary object
        return a + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- Works fine with friend functions - 3 separate overloading are provided and Preserves the encapsulation too
- **Note:** A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before constructor) will be discussed when we take up cast operators

Programming in Modern C++ Partha Pratim Das M19.13



Program 19.03: Extending operator+ with friend Fu

```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit conversion is allo
    void disp() { cout << re << " +j " << im << endl; }
    friend Complex operator+(const Complex &a, const Complex &b) { // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
    }
    friend Complex operator+(const Complex &a, double d) {         // Overload 2
        Complex b(d); // Create temporary object
        return a + b; // Use Overload 1
    }
    friend Complex operator+(double d, const Complex &b) {         // Overload 3
        Complex a(d); // Create temporary object
        return a + b; // Use Overload 1
    }
};
int main() { Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp();  // d3 = 4.1 +j 6.5. Overload 1
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- Works fine with friend functions - 3 separate overloading are provided and Preserves the encapsulation too
- **Note:** A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before constructor) will be discussed when we take up cast operators

Programming in Modern C++ Partha Pratim Das M19.13

I could have done a different solution also, which we will discuss further in a much later module when we talk about casting is that do not provide these do not make the constructor explicit and just provide one operator mind do this will also work. But, it will work differently. The way it will work is since d1 plus d2 is fine, but when I do say d1 plus 6.2, 6.2 is a double and what I need as a second argument is a Complex fit does not match.

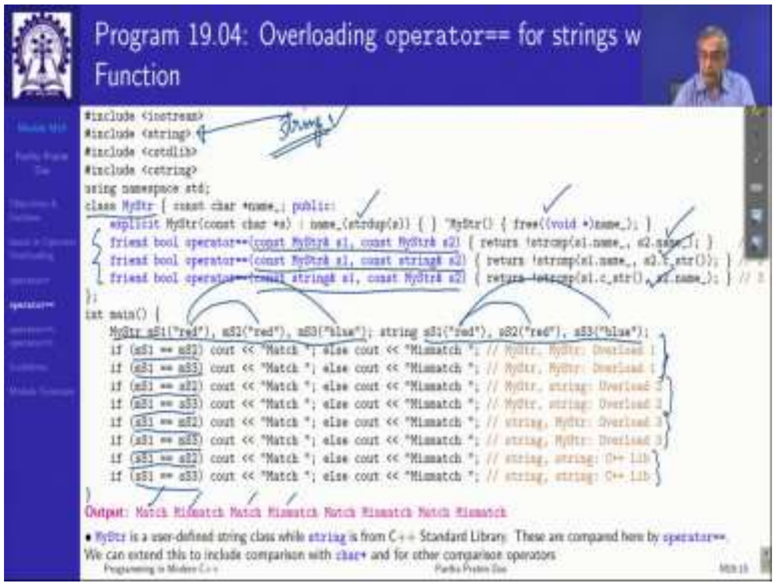
But what the compiler will see is, is there a way to construct a Complex object from a double, well it is there. Therefore it will construct a temporary double object and call this function that is a different way the mechanism will work which is often not very advisable because then it will be able to possibly cast integer also. It depends on what I want to define. When I make the constructor explicit, I actually stop all of these irritating silent conversions from happening and I exactly know what are the data types that I am going to add through my operator. So, the friend function friend operator function really provides a good solution.

(Refer Slide Time: 14:28)

Overloading Comparison Operators

Overloading Comparison Operators

Programming in Modern C++ Partha Pratim Das M19.14



Program 19.04: Overloading operator== for strings w Function

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <cstring>
using namespace std;
class MyStr { const char *name_; public:
    explicit MyStr(const char *s) : name_(strdup(s)) { } ~MyStr() { free((void *)name_); }
    friend bool operator==(const MyStr& s1, const MyStr& s2) { return !strcmp(s1.name_, s2.name_); }
    friend bool operator==(const MyStr& s1, const string& s2) { return !strcmp(s1.name_, s2.c_str()); }
    friend bool operator==(const string& s1, const MyStr& s2) { return !strcmp(s1.c_str(), s2.name_); } // 3
};
int main() {
    MyStr mS1("red"), mS2("red"), mS3("blue"); string sS1("red"), sS2("red"), sS3("blue");
    if (mS1 == mS2) cout << "Match "; else cout << "Mismatch "; // MyStr, MyStr: Overload 1
    if (mS1 == mS3) cout << "Match "; else cout << "Mismatch "; // MyStr, MyStr: Overload 1
    if (mS1 == sS2) cout << "Match "; else cout << "Mismatch "; // MyStr, string: Overload 2
    if (mS1 == sS3) cout << "Match "; else cout << "Mismatch "; // MyStr, string: Overload 2
    if (sS1 == mS2) cout << "Match "; else cout << "Mismatch "; // string, MyStr: Overload 3
    if (sS1 == mS3) cout << "Match "; else cout << "Mismatch "; // string, MyStr: Overload 3
    if (sS1 == sS2) cout << "Match "; else cout << "Mismatch "; // string, string: C++ Lib
    if (sS1 == sS3) cout << "Match "; else cout << "Mismatch "; // string, string: C++ Lib
}
```

```
Output: Match Mismatch Match Mismatch Match Mismatch Match Mismatch
```

- MyStr is a user-defined string class while string is from C++ Standard Library. These are compared here by operator==.

We can extend this to include comparison with char* and for other comparison operators

Programming in Modern C++ Partha Pratim Das M19.15

Just as an illustration take a look at overloading a comparison operator say equal to operator. So, what am I having? I have included string here because I want to use the string type of the library. And I am defining a class MyStr which is kind of a simple wrapper string for me which just copies takes a cstring copies and makes an object later on at the time of destruction frees that up.

I want to compare a library defined stream, string object and a MyStr object for say equality. So, there are 3 possibilities, 4 possibilities actually because there are two classes. So, there are four ways I may compare. So, one is my string with my string which is these two cases. So, I have defined objects mS1, mS2, mS3 which are MyStr objects sS1, sS2, sS3 are library string object.

So, I compare mS1 with mS2, mS1 with mS3 one have kept equal another I have kept separate. These two are equal these two are separate just to show that comparison outcome being true and outcome being false both work. So, this will work with the first overload. Because both are MyStr.

In the second, I have mS1 and sS2, mS1 and sS3. So, it is MyStr and string MyStr and string. So, the second overload will work and I will have correct response based on that. The third is string and MyStr string and MyStr that is a third overload this will work to give result in this these two cases all are correct you can see match mismatch match mismatch.

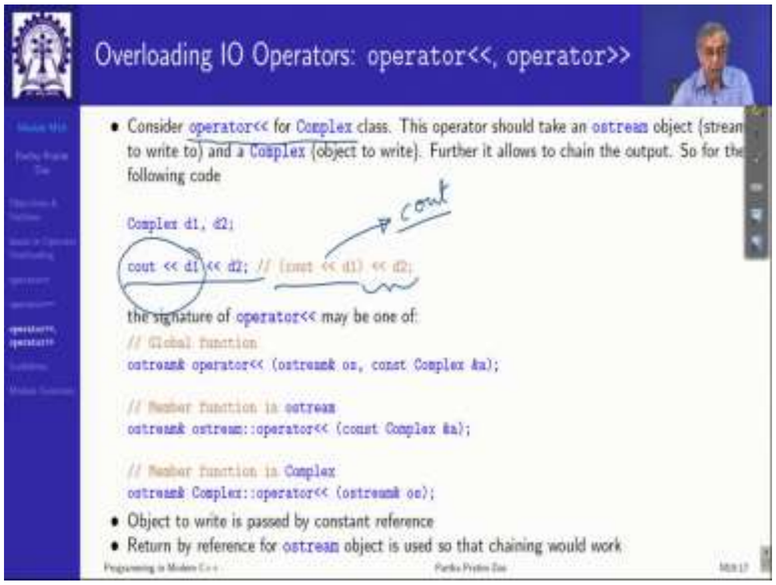
And finally, the fourth possibility is I compare a library string object with the library string object that is sS1 sS2, sS1 sS3. Now, in this case, I am not providing anything here because my class is not even involved and library has already provided a overloaded comparison equal to operator for this string class.

So, here it will also work with the comparison operator from the C++ standard library. So, you can see that I can define operators, I can mix operators from other class obviously with judicious design, and this was possible without breaking the encapsulation or anything by using simply the print function.

(Refer Slide Time: 17:39)

Overloading IO Operators

Overloading IO Operators



Overloading IO Operators: operator<<, operator>>

- Consider operator<< for Complex class. This operator should take an ostream object (stream to write to) and a Complex (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

cout

the signature of operator<< may be one of:

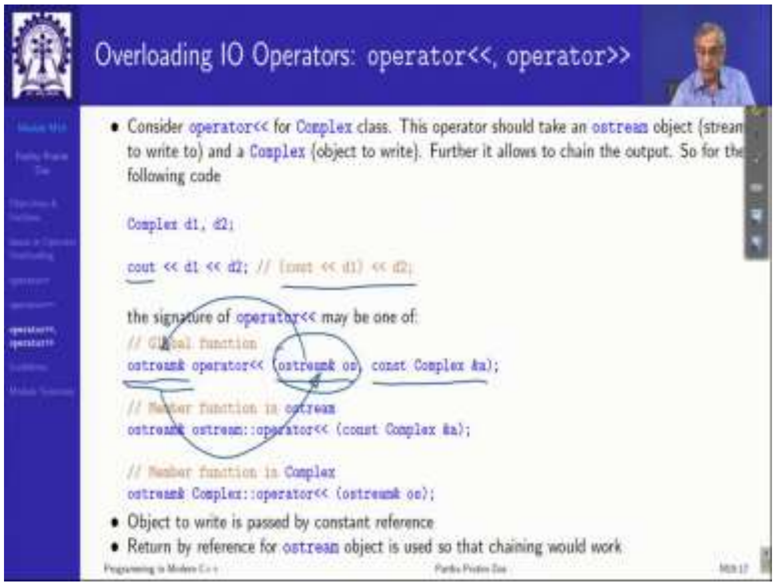
```
// Global function
ostream& operator<< (ostream& os, const Complex &a);

// Member function in ostream
ostream& ostream::operator<< (const Complex &a);

// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for ostream object is used so that chaining would work

So, let me move on to discuss the io operators. The other issue the second issue I talked of. So, we want to say create an output streaming operator for Complex class. So, that any complex number I have I should be able to just write this two Complex number d1, d2 as you know this will be done first. It is left associative this will be done first, which will which must return the cout again because I want to change that this will be done next this is.

So, if I want to this is the behaviour that I get for integer character double these kinds of built in types. So, I want that behaviour to be present. So, let me see of the three options what will be the different signatures of the overloaded function

(Refer Slide Time: 18:43)



Overloading IO Operators: operator<<, operator>>

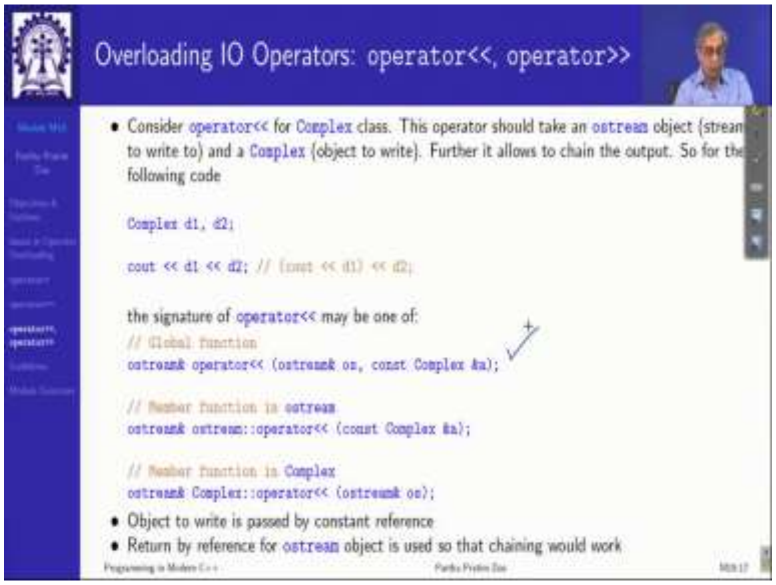
- Consider operator<< for Complex class. This operator should take an ostream object (stream to write to) and a Complex (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

the signature of operator<< may be one of:

```
// Global function
ostream& operator<< (ostream& os, const Complex &a);

// Member function in ostream
ostream& ostream::operator<< (const Complex &a);

// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for ostream object is used so that chaining would work



Overloading IO Operators: operator<<, operator>>

- Consider operator<< for Complex class. This operator should take an ostream object (stream to write to) and a Complex (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

the signature of operator<< may be one of:

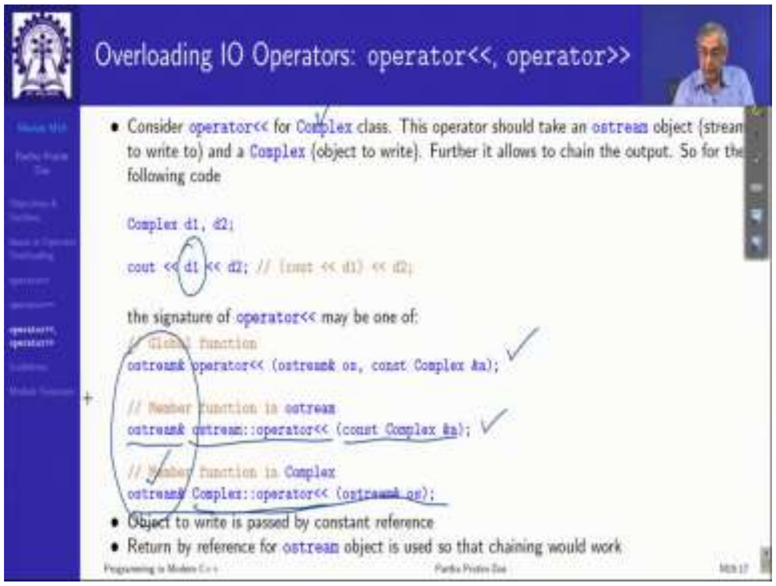
```
// Global function
ostream& operator<< (ostream& os, const Complex &a);

// Member function in ostream
ostream& ostream::operator<< (const Complex &a);

// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for ostream object is used so that chaining would work

If I have a global function, then my first operate is this one which is ostream is passed on as a reference because certainly you do not want to copy the entire output stream and also pass the Complex object as a constant reference. This is not to be made const the ostream reference is not to be made const because I am going to write to it. So, it is going to get changed.

And after I have written I must get back after I have written I must get back the same ostream object. So, that when I do the chaining, this is what I have got back will be able to go into the second instance of the operator. So, that is the basic style that we will have to maintain. So, this is if we do it by global function.

(Refer Slide Time: 19:38)



Overloading IO Operators: operator<<, operator>>

- Consider operator<< for Complex class. This operator should take an ostream object (stream to write to) and a Complex (object to write). Further it allows to chain the output. So for the following code

```
Complex d1, d2;

cout << d1 << d2; // (cout << d1) << d2;
```

the signature of operator<< may be one of:

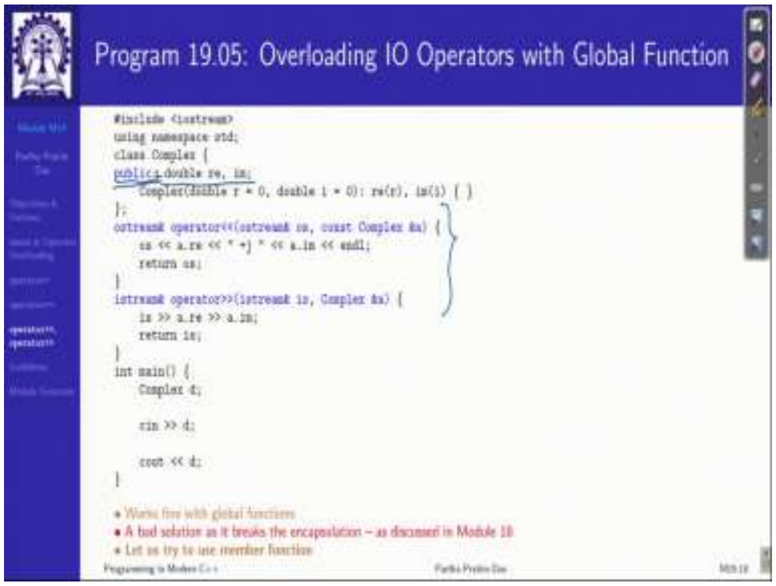
```
// Global function
ostream& operator<< (ostream& os, const Complex &a);

// Member function in ostream
ostream& ostream::operator<< (const Complex &a);

// Member function in Complex
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for ostream object is used so that chaining would work

What if we do by a member function of the operator class member function of the operator class. So, I am sorry, not the operator class member function of the ostream class that is there in the library. So, the operator now is ostream colon colon operator output. It takes one parameter which is a Complex the ostream is already there, and it returns the reference to the ostream object, which should be fun.

The third option is the other class involved here is my Complex class. So, I can make it an operator of the Complex class, if I make it the operator of the Complex class. So, basically the operator will be invoked on this, then the parameter it has to take is the ostream object, and it has to return that ostream object, it has to return the ostream object in every case.

(Refer Slide Time: 20:39)



Program 19.05: Overloading IO Operators with Global Function

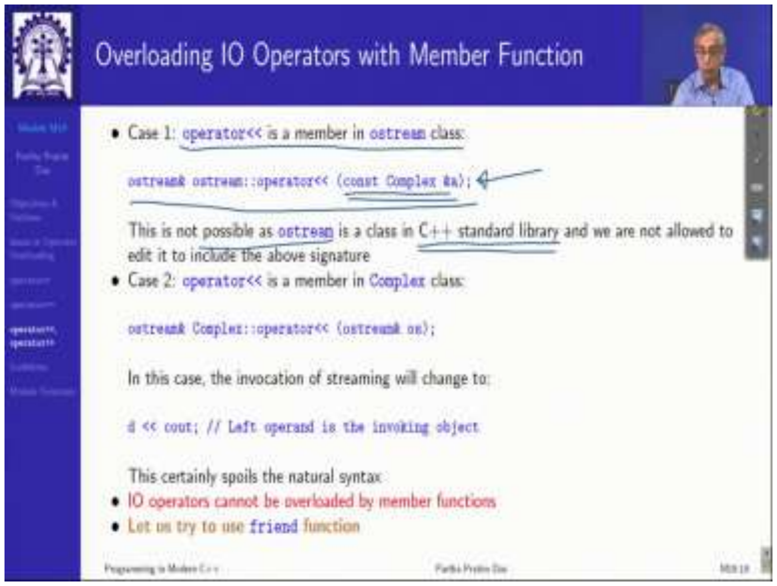
```
#include <iostream>
using namespace std;
class Complex {
public: double re, im;
    Complex(double r = 0, double i = 0): re(r), im(i) { }
};
ostream& operator<<(ostream& os, const Complex &a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}
istream& operator>>(istream& is, Complex &a) {
    is >> a.re >> a.im;
    return is;
}
int main() {
    Complex d;

    cin >> d;

    cout << d;
}
```

- Works fine with global functions
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- Let us try to use member function

Programming in Modern C++ Partha Pratim Das M19.18

Now, let us see what happens in these cases. So, with a global function, if I do it with a global function, if I do it with the global function, then it will work fine, except for the fact that I have to break the encapsulation, the good old problem. So, breaking this encapsulation is not going to be a good thing to do. So, global functions are not preferred.

(Refer Slide Time: 21:05)



Overloading IO Operators with Member Function

- Case 1: operator<< is a member in ostream class:

```
ostream& ostream::operator<< (const Complex &a);
```

This is not possible as ostream is a class in C++ standard library and we are not allowed to edit it to include the above signature
- Case 2: operator<< is a member in Complex class:

```
ostream& Complex::operator<< (ostream& os);
```

In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

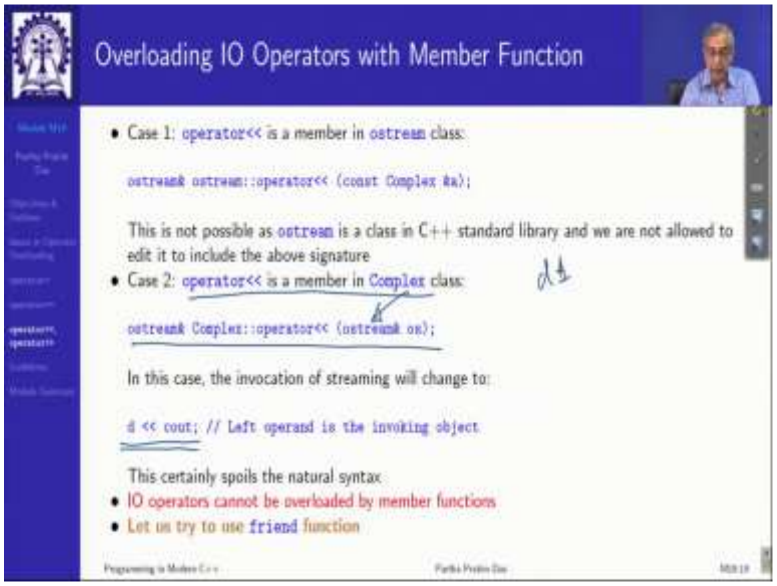
This certainly spoils the natural syntax
- IO operators cannot be overloaded by member functions
- Let us try to use friend function

Programming in Modern C++ Partha Pratim Das M19.18

So, my next option is to use a member function. Now, let us say it is a member function of what suppose it is a member function of the ostream class, which is has this signature. Now, this is a perfect solution. But the reason it will not be possible because as a developer, I am not allowed to make changes to my C++ standard library. So, ostream object is a part of ostream class is a part of the C++ standard library.

So, I cannot make a change, I cannot add a new operator overloaded operator for my Complex class argument to get this implemented. So, this is technically a possible feasible solution, but this is not allowed according to the rules of the library.

(Refer Slide Time: 22:06)



Overloading IO Operators with Member Function

- Case 1: `operator<<` is a member in `ostream` class:

```
ostream& ostream::operator<< (const Complex &a);
```

This is not possible as `ostream` is a class in C++ standard library and we are not allowed to edit it to include the above signature

- Case 2: `operator<<` is a member in `Complex` class:

```
ostream& Complex::operator<< (ostream& os);
```

In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

This certainly spoils the natural syntax

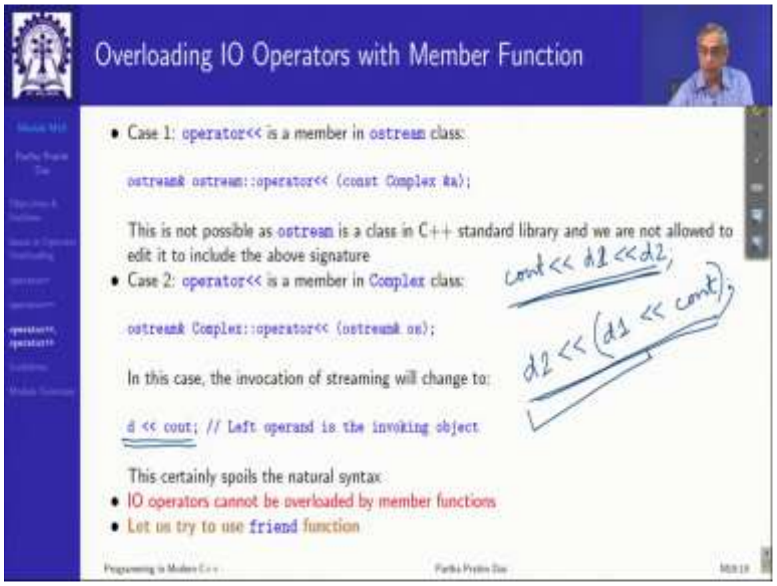
- IO operators cannot be overloaded by member functions
- Let us try to use `friend` function

d1

Programming in Modern C++ Partha Pratim Das M19.19

So, I have to then look at as a member function, I have to look at as a member function for the Complex class. So then, this is the signature that will have it is Complex colon colon operator output, which takes ostream as an object as an argument. So, ostream becomes the second argument or the right argument. Therefore, when I want to output this, I have to write it as d output cout like, it gets inverted, which is not neither it is good for streaming. So, if I have to.

(Refer Slide Time: 22:45)



Overloading IO Operators with Member Function

- Case 1: `operator<<` is a member in `ostream` class:

```
ostream& ostream::operator<< (const Complex &a);
```

This is not possible as `ostream` is a class in C++ standard library and we are not allowed to edit it to include the above signature

- Case 2: `operator<<` is a member in `Complex` class:

```
ostream& Complex::operator<< (ostream& os);
```

In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

This certainly spoils the natural syntax

- IO operators cannot be overloaded by member functions
- Let us try to use `friend` function

cout << d1 << d2;

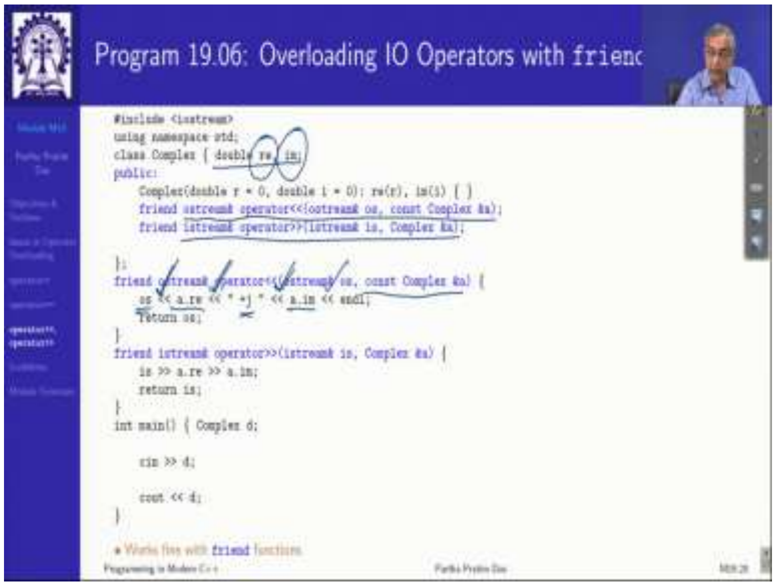
d2 << (d1 << cout);

Programming in Modern C++ Partha Pratim Das M19.19

Actually, I would have liked to write cout say there are two objects d1 and d2, I would like to write this, this is what the built in types write. But here what I will have to write, I will have to write because unless my object d of the Complex class come on the left of the operand, I will not be able to invoke this operator and pass the cout to it. So, I have to this is completely unnatural, and then I have another problem is that this operator is left associative.

So, if I write this and do not do anything, it will try to first execute this instance, which takes two complex and does not make any sense. So, it is not only this, I have to actually put a parenthesis around it. So, while this is the form of the expression for a built in type, my overloaded operator with my overloaded operator the expression will look something like this, which is really really annoying, this is not something that that is making the natural extension of the type as I wanted to do.

(Refer Slide Time: 24:09)



Program 19.06: Overloading IO Operators with frienc

```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    Complex(double r = 0, double i = 0): re(r), im(i) { }
    friend ostream& operator<<(ostream& os, const Complex &a);
    friend istream& operator>>(istream& is, Complex &a);

};
friend ostream& operator<<(ostream& os, const Complex &a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}
friend istream& operator>>(istream& is, Complex &a) {
    is >> a.re >> a.im;
    return is;
}
int main() { Complex d;

    cin >> d;

    cout << d;
}
```

- Works fine with friend functions

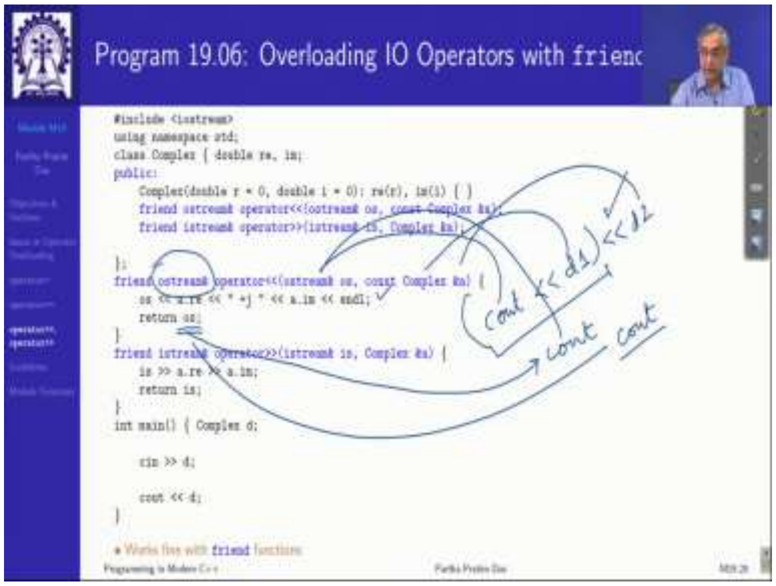
Programming in Modern C++ Partha Pratim Das M19.28

So, the only choice left is to have a friend function. So, now what I do is it is like the global itself, take the ostream and the Complex istream and the Complex return the ostream for output operator, return the istream for the input operator and you can make the encapsulation re, in again, because you have friend functions.

And the way you write this operator is very simple this you have got this a. So, as a friend, it can access re and im. So, you take re you take im and I am assuming that to mean it is a complex number I am writing it in the form of real component plus j imaginary component. So, I stream that two ways this is this operators will work according to the respective built in type.

For example, here the output streaming operator for double will work here the output streaming operator for constant string or string will work. Here again the double will work and so on. So, ostream is now got whatever I have outputted. So, what I will have to do I will have to give it back as a result.

(Refer Slide Time: 25:38)



Program 19.06: Overloading IO Operators with friend

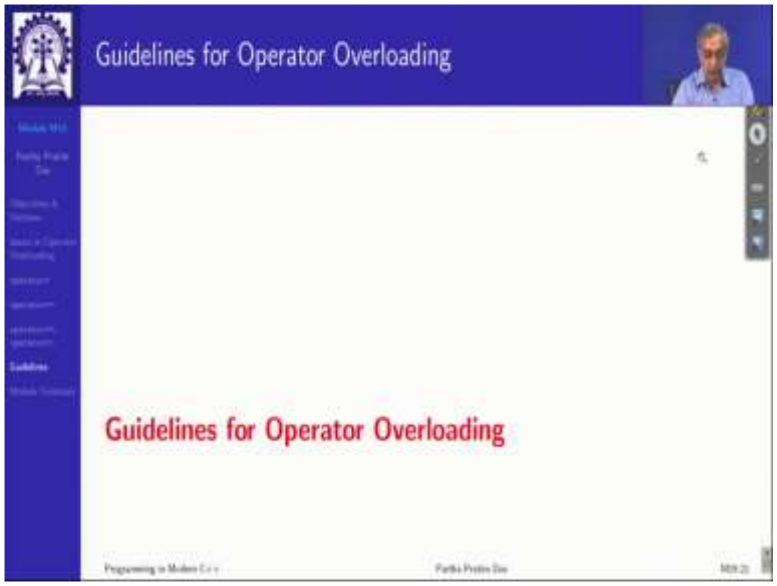
```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    Complex(double r = 0, double i = 0): re(r), im(i) { }
    friend ostream& operator<<(ostream& os, const Complex &a);
    friend istream& operator>>(istream& is, Complex &a);
};
friend ostream& operator<<(ostream& os, const Complex &a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}
friend istream& operator>>(istream& is, Complex &a) {
    is >> a.re >> a.im;
    return is;
}
int main() { Complex d;

    cin >> d;

    cout << d;
}
```

- Works fine with friend functions

So, I returned ostream which actually comes back as a as my stream reference result. So, when I now write cout d1, d2 and chain it by left associativity this is done first, this goes as os, this goes as a and into that cout this is written and what comes back is again cout. So, the result of this is cout.

So, again when the second instance is happening cout goes in as ostream, d2 goes in and a and what comes out is again cout to which I can stream anything else. So, it perfectly falls in order in terms of the syntax of the expression of output, you will not be able to see any difference from the ordinary built in types. So, the same thing will also hold for the istream.

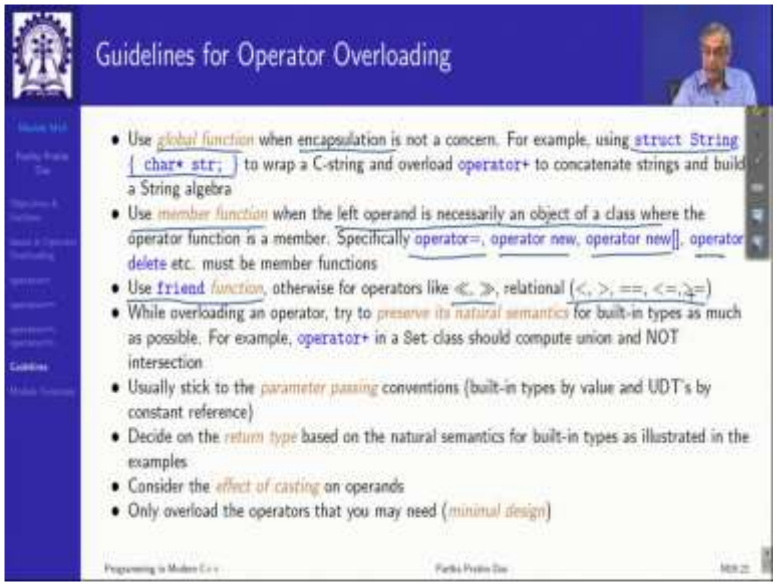
(Refer Slide Time: 28:49)



Guidelines for Operator Overloading

Guidelines for Operator Overloading

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



Guidelines for Operator Overloading

- Use global function when encapsulation is not a concern. For example, using `struct String { char* str; }` to wrap a C-string and overload `operator+` to concatenate strings and build a String algebra
- Use member function when the left operand is necessarily an object of a class where the operator function is a member. Specifically `operator=`, `operator new`, `operator new[]`, `operator delete` etc. must be member functions
- Use `friend` function, otherwise for operators like ≪, ≫, relational (<, >, ==, <=, >=)
- While overloading an operator, try to preserve its natural semantics for built-in types as much as possible. For example, `operator+` in a Set class should compute union and NOT intersection
- Usually stick to the parameter passing conventions (built-in types by value and UDT's by constant reference)
- Decide on the return type based on the natural semantics for built-in types as illustrated in the examples
- Consider the effect of casting on operands
- Only overload the operators that you may need (minimal design)

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

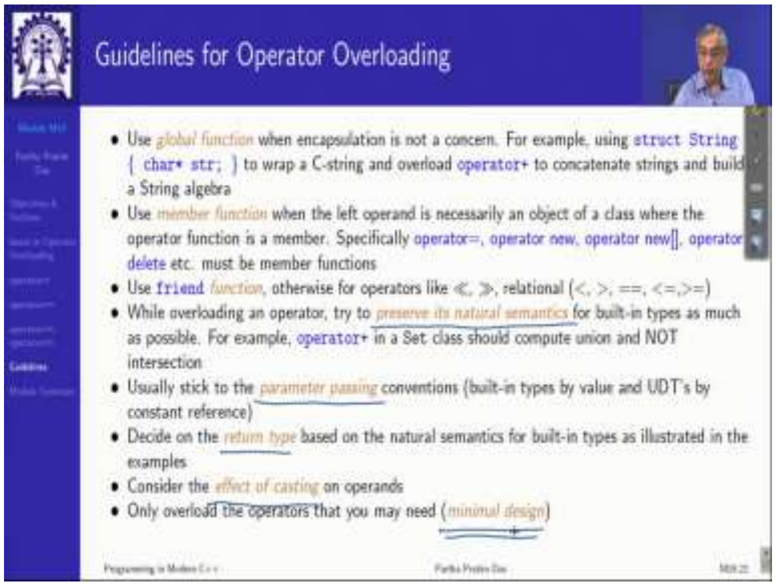
So, this was about operator overloading to summarise here has some general guidelines. So, out of the three options, what will you choose when? So, you can choose a global function, when you do not care about encapsulation, it is not a concern. For example, you are trying to write something for a struct in C++ which means there is no hiding, no encapsulation.

So, then you can just use global function to keep things simple. You will have to use member function when the left operand is necessarily an object of the same class for which you are overloading the operator then you can use a member function. And specifically there are operators which will have to be member function overloaded as member function like operator assignment, you have already seen that operator new, operator array new, operator delete and so, on they all will have to be necessarily member.

So, it is not a generalisation that you will always do friend, you will do global if you do not if you want to keep things simple and to not have encapsulation concerned you will have to do member function if it fundamentally defines the behaviour of your class and always has will have the left operand as an object of the same class that that is a simple solution.

Otherwise, for example, other operators like your increment operator and post increment pre increment could also be done as member functions. Otherwise, you use friend function for operators like streaming for relational and so on. So, this is the basic structuring rule of operator overloading.

(Refer Slide Time: 28:56)



Guidelines for Operator Overloading

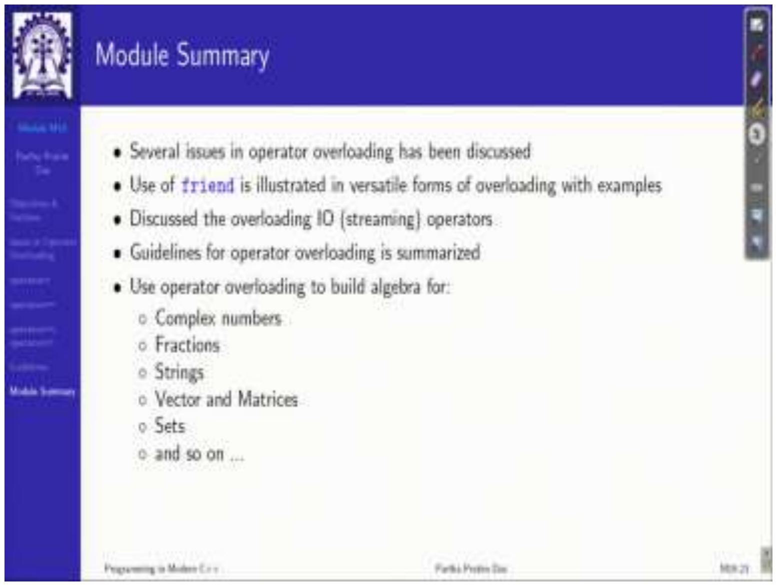
- Use global function when encapsulation is not a concern. For example, using `struct String { char* str; }` to wrap a C-string and overload `operator+` to concatenate strings and build a String algebra
- Use member function when the left operand is necessarily an object of a class where the operator function is a member. Specifically operator=, operator new, operator new[], operator delete etc. must be member functions
- Use `friend` function, otherwise for operators like ≪, ≫, relational (<, >, ==, <=,>=)
- While overloading an operator, try to preserve its natural semantics for built-in types as much as possible. For example, `operator+` in a Set class should compute union and NOT intersection
- Usually stick to the parameter passing conventions (built-in types by value and UDT's by constant reference)
- Decide on the return type based on the natural semantics for built-in types as illustrated in the examples
- Consider the effect of casting on operands
- Only overload the operators that you may need (minimal design)

In terms of semantics, it is good to preserve the natural semantics. For example, if we are overloading operator plus for say set, then you should do that for union not for intersection. So, that same sense so similar sense must be prevailed. Similarly, for usually for parameter passing, follow the usual convention that pass the built in types by value and the user defined types by constant reference.

The return type will decide based on whether you want to have a reference if what you are returning is an existing object like cout or whether you will return by value because it did not exist and you are creating like in operator plus for Complex and so on. So, based on that choose the return type appropriately should consider the effect of casting as well.

We will look at this more when we deal with casting operators. And last but not the least is do not overload an operator which you do not need to that is keep to minimal design. Because the more you overload operators you are adding semantics to your design and it may be confusing for someone who is not well acquainted with your design. So, only add operators that you really need for your class.

(Refer Slide Time: 30:52)



Module Summary

- Several issues in operator overloading has been discussed
- Use of `friend` is illustrated in versatile forms of overloading with examples
- Discussed the overloading IO (streaming) operators
- Guidelines for operator overloading is summarized
- Use operator overloading to build algebra for:
  - Complex numbers
  - Fractions
  - Strings
  - Vector and Matrices
  - Sets
  - and so on ...

So, we have talked about several issues of operator overloading discuss that and particularly highlighted the context and the mechanism by which friend functions can be used for operator overloading to get a really nice solution for IO operators and so on. These operator overloading can be used to build algebra for various different types as we will see more and more.

Thank you all very much. Thank you for your attention. I hope this will this concludes our discussion on operator overloading. I think you have got a good grasp to that. I will record a tutorial to show you how using this operator overloading and other features, you can create Complex like data type much in the to behave and to lexically, syntactically look very similar to the built in type and you can get to the creation of algebra.