



## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

# Module 19: Programming C++

## Overloading Operator for User-Defined Types: Part 2

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ernet.in*

Tanwi Mallick  
Srijoni Majumdar  
Himadri B G S Bhuyan



# Module Objectives

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

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



# Module Outline: Lecture-34

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Issues in Operator Overloading
- Extending operator+
- Overloading IO Operators
- Guidelines for Operator Overloading



# Operator Function for UDT

## RECAP (Module 18)

### Module 19

Partha Pratim  
Das

### Objectives & Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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



# Issue 1: Extending operator+

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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



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

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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



# Program 19.01: Extending operator+ with Global Function

## Module 19

Partha Pratim Das

Objectives & Outline

Issues in Operator Overloading

Extending operator+

Overloading IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { public: double re, im;
    explicit Complex(double r = 0, double i = 0): re(r), im(i) { }
    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
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3
    return 0;
}
```

- 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



# Program 19.02: Extending operator+ with Member Function

## Module 19

Partha Pratim Das

Objectives & Outline

Issues in Operator Overloading

Extending operator+

Overloading IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { }
    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); return *this + 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
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2

    //d3 = 4.2 + d2;          // Overload 3 is not possible - needs an object of left
    //d3.disp();
    return 0;
}
```

- **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





# Program 19.03: Extending operator+ with friend Function

## Module 19

Partha Pratim Das

Objectives & Outline

Issues in Operator Overloading

Extending operator+

Overloading IO Operators

Guidelines

Summary

```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) { }
    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); return a + b; // Create temporary object and use Overload 1
    }
    friend 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
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3
    return 0;
}
```

- Works fine with friend functions - 3 separate overloading are provided
- 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



# Overloading IO Operators: operator<<, operator>>

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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



# Program 19.04: Overloading IO Operators with Global Function

## Module 19

Partha Pratim Das

Objectives & Outline

Issues in Operator Overloading

Extending operator+

Overloading IO Operators

Guidelines

Summary

```
#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;

    return 0;
}
```

- 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



# Overloading IO Operators with Member Function

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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**



# Guidelines for Operator Overloading

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- 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 a class where the operator function is a member and multiple types of operands are not involved
- Use **friend function**, otherwise
- 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. For example, as in pre-increment and post-increment operators
- Consider the **effect of casting** on operands
- Only overload the operators that you may need (**minimal design**)



# Module Summary

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

- Several issues 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 ...



# Instructor and TAs

## Module 19

Partha Pratim  
Das

Objectives &  
Outline

Issues in  
Operator  
Overloading

Extending  
operator+

Overloading  
IO Operators

Guidelines

Summary

Name	Mail	Mobile
Partha Pratim Das, <i>Instructor</i>	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, <i>TA</i>	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, <i>TA</i>	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, <i>TA</i>	himadribhuyan@gmail.com	9438911655