# Module 18: Programming in C++

## Overloading Operator for User-Defined Types: Part 1

Sourangshu Bhattacharya

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

*sourangshu@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

# Module Objectives

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

# Module Outline

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

# Motivation

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

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

- 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

# Non-Member Operator Function

- 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))` |

# 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))` |

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

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

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

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

**Overloading + for complex addition**

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

- **Output**: Real: 31, Imaginary: 42.5

- **operator+** is overloaded to perform addition of two complex numbers which are of **struct complx** type

**Overloading + for string cat**

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1,
                 const String& s2) {
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
                         strlen(s2.str) + 1);
    strcpy(s.str, s1.str);
    strcat(s.str, s2.str);
    return s;
}
int main() {
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");
    name = fName + lName; // Overload operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
    return 0;
}
```

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

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

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

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

**Output**:

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

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

# Program 18.03: Using Member Function

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

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

**Output**:

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

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

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

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

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

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

# Notes on Overloading operator= RECAP (Module 14)

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

# Program 18.04: Overloading Unary Operators

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

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

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

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

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

    return 0;
}
```

- Output

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

- The **pre-operator** should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should return ***this;**

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

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

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

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

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

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

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

    return 0;
}
```

- Output

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

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

# 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