Module 19: Programming in C++

Overloading Operator for User-Defined Types: Part 2

Intructors: Abir Das and Sourangshu Bhattacharya

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

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern 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 friend function and its advantages

Module Outline

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

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

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

Program 19.01: Extending operator+ with Global Function


```
#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
}
```

- 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


```
#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 allowed
    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

Operator Overloading using `friend`

- Using global function, accessing `private` data members inside operator function is gets difficult
- 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`

Program 19.03: Extending operator+ with friend Function


```
#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 allowed
    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

Program 19.04: Overloading operator== for strings with friend 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_); }    // 1
    friend bool operator==(const MyStr& s1, const string& s2) { return !strcmp(s1.name_, s2.c_str()); } // 2
    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

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

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**

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**

Program 19.06: Overloading IO Operators with `friend` Function


```
#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

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 (`<`, `>`, `==`, `!=`, `<=`, `>=`) should be overloaded through `friend`
- 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*)

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 ...