Object Oriented Programming with C++

9. Operator Overloading

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

Operator Overloading

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
        double re, im;
public:
        Complex(double re = 0, double im = 0) {
                this->re = re;
                this->im = im;
        }
        void print() {
                cout << re << " + j" << im << endl;
        }
        Complex addm(Complex &c) {
                cout << "Inside method" << endl;
                Complex result;
                result.re = this->re + c.re;
                result.im = this->im + c.im;
                return result;
        }
        // declared as friend function because re and im are private
        // if they were public, we dont need to declare it as friend
        friend Complex addf(Complex &c1, Complex &c2);
};
```

```
Complex addf(Complex &c1, Complex &c2) {
        cout << "Inside friend function" << endl;
        Complex result;
        result.re = c1.re + c2.re;
        result.im = c1.im + c2.im;
        return result;
}

int main() {
        Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4;
        c3 = c1.addm(c2);
        c4 = addf(c1, c2);
        // error: no match for 'operator+'
        // (operand types are 'Complex' and 'Complex
        // c3 = c1 + c2;
        c3.print();
        c4.print();
        return 0;
}
```

This is a normal function, it is not a method. Hence it needs to be declared as friend function because it accesses private data memebers

This results in error as + operator has not yet been overloaded to operate on two complex numbers. Hence compiler does not know which function or method to call

```
Inside method
Inside friend function
3.3 + j3.3
3.3 + j3.3
```

Operator Overloading

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex addm(Complex &c) {
		cout << "Inside method" << endl;
		Complex result;
		result.re = this->re + c.re;
		result.im = this->im + c.im;
		return result;
	}
	// declared as friend function because re and im are private
	// if they were public, we dont need to declare it as friend
	friend Complex operator+(Complex &c1, Complex &c2);
};
```

```
Complex operator+(Complex &c1, Complex &c2) {
	cout << "Inside friend function" << endl;
	Complex result;
	result.re = c1.re + c2.re;
	result.im = c1.im + c2.im;
	return result;
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4;
	c3 = c1.addm(c2);
	c4 = c1 + c2;



	c3.print();
	c4.print();
	return 0;
}
```

```
Inside method
Inside friend function
3.3 + j3.3
3.3 + j3.3
```

Operator Overloading

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method" << endl;
		Complex result;
		result.re = this->re + c.re;
		result.im = this->im + c.im;
		return result;
	}
	// declared as friend function because re and im are private
	// if they were public, we dont need to declare it as friend
	friend Complex operator+(Complex &c1, Complex &c2);
};

Complex operator+(Complex &c1, Complex &c2) {
	cout << "Inside friend function" << endl;
	Complex result;
	result.re = c1.re + c2.re;
	result.im = c1.im + c2.im;
	return result;
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4;
	c3 = c1 + c2;
	c4 = c1 + c2;
	// Above two operators will result in error
	// error: ambiguous overload for ‘operator+’
	// (operand types are ‘Complex’ and ‘Complex’)
	c3.print();
	c4.print();
	return 0;
}
```

Operator Overloading

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method" << endl;
		Complex result;
		result.re = this->re + c.re;
		result.im = this->im + c.im;
		return result;
	}
	// declared as friend function because re and im are private
	// if they were public, we dont need to declare it as friend
	friend Complex addf(Complex &c1, Complex &c2);
};
```

```
Complex addf(Complex &c1, Complex &c2) {
	cout << "Inside friend function" << endl;
	Complex result;
	result.re = c1.re + c2.re;
	result.im = c1.im + c2.im;
	return result;
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4;
	c3 = c1 + c2;
	C4 = addf(c1 ,c2);



	c3.print();
	c4.print();
	return 0;
}
```

```
Inside method
Inside friend function
3.3 + j3.3
3.3 + j3.3
```

Operator Overloading

- New keyword – **operator**
- In order to overload operator, operator function/method needs to be defined
  - Name of the operator function/method starts with keyword **operator,** followed by operator symbol to be overloaded
    - For example, in order to overload operator '**+**' to add two complex numbers (**c1 + c2**)
      - Complex **operator+**(Complex &c1, Complex &c2); // overload using function
      - Complex Complex::**operator+**(Complex &c); // overload using method
  - To overload binary operator – function takes two parameters, while method takes one
  - To overload unary operator – function takes one parameters, while method takes zero

- Operator functions are implicit for predefined operators of built-in types and can not be redefined

- Operator functions/methods can be defined for struct and union types as well
- Operator functions can be defined for enum type too
- At least one argument of an operator function must be of a user-defined type (struct, class, union, enum), or a reference to one.

Operator Overloading

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method" << endl;
		Complex result;
		result.re = this->re + c.re;
		result.im = this->im + c.im;
		return result;
	}
	friend Complex operator*(Complex c1, int num);
};

Complex operator*(Complex c1, int num) {
	cout << "Inside friend function" << endl;
	Complex result;
	result.re = c1.re * num;
	result.im = c1.im * num;
	return result;
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4;
	c3 = c1 + c2;
	c4 = c3 * 2;
	c3.print();
	c4.print();
	return 0;
}
```

```
Inside method
Inside friend function
3.3 + j3.3
6.6 + j6.6
```

- Operator functions/methods can be defined for two operands of different types too
- Parameters can be passed by value or reference to the operator functions/methods

Operator Overloading

- Only existing operators could be overloaded. We can not create new operators of our own (e.g. **, <>)
- Intrinsic properties of overloaded operators can not be changed
  - Preserves arity (number of operands it takes)
  - Preserves precedence
  - Preserves associativity
- Both unary prefix and postfix could be oveloaded
  - void class_name::operator++() // prefix
  - void class_name::operator++(int) // postfix, **int** is used to distinguish it from prefix
  - Like overload of other operators, they can also return value and could be overloaded using functions
- Some operators could not be overloaded
  - :: . .* sizeof ?:
- Operators && || and , could be overloaded but they lose their special properties: short-circuit evaluation and sequencing (they will not be a sequence point for new types)

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};

Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}

// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	c3 = c1 + c2;
	c4 = c3 * 2;
	c5 = 2 * c4;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method I
Inside method II
Inside friend function
3.3 + j3.3
6.6 + j6.6
13.2 + j13.2
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};
```

```
Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}

// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}

int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	c3 = c1.operator+(c2);  // c1 + c2
	c4 = c3.operator*(2);   // c3 * 2
	c5 = operator*(2, c4);  // 2 * c4
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method I
Inside method II
Inside friend function
3.3 + j3.3
6.6 + j6.6
13.2 + j13.2
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};
```

```
Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}
// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}
int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	// error: cannot bind non-const lvalue
	// reference of type ‘Complex&’ to an rvalue
	// of type ‘Complex’
	// Temporary object can not be assigned to
	// non-const ref
	c3 = c1 + c2 * 2;
	// No error here as temporary object can
	// call method
	c4 = (c1 + c2) * 2;
	c5 = 2 * c1 * 3;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};
```

```
Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}
// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}
int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	// error: cannot bind non-const lvalue
	// reference of type ‘Complex&’ to an rvalue
	// of type ‘Complex’
	// Temporary object can not be assigne to
	// non-const ref
	// c3 = c1 + c2 * 2;
	// No error here as temporary object can
	// call method
	c4 = (c1 + c2) * 2;
	c5 = 2 * c1 * 3;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method I
Inside method II
Inside friend function
Inside method II
0 + j0
6.6 + j6.6
6.6 + j6.6
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(Complex c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};
```

```
Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}
// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}
int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	// No error, as operator+ accepts arg by value
	c3 = c1 + c2 * 2;
	c4 = (c1 + c2) * 2;
	c5 = 2 * c1 * 3;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method II
Inside method I
Inside method I
Inside method II
Inside friend function
Inside method II
5.5 + j5.5
6.6 + j6.6
6.6 + j6.6
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(const Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, Complex);
};
```

```
Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}
// This can not be achieved through operator method
Complex operator*(int num, Complex c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}
int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	// No error, as operator+ accepts arg by
	// ref to const
	c3 = c1 + c2 * 2;
	c4 = (c1 + c2) * 2;
	c5 = 2 * c1 * 3;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method II
Inside method I
Inside method I
Inside method II
Inside friend function
Inside method II
5.5 + j5.5
6.6 + j6.6
6.6 + j6.6
```

```
#include<iostream>

using std::cout;
using std::endl;

class Complex {
	double re, im;
public:
	Complex(double re = 0, double im = 0) {
		this->re = re;
		this->im = im;
	}
	void print() {
		cout << re << " + j" << im << endl;
	}
	Complex operator+(const Complex &c) {
		cout << "Inside method I" << endl;
		// No need of result object
		return Complex(this->re + c.re, this->im + c.im);
	}
	Complex operator*(int num);
	friend Complex operator*(int, const Complex &);
};

Complex Complex::operator*(int num) {
	cout << "Inside method II" << endl;
	return Complex(this->re * num, this->im * num);
}
// Its good practice to pass objects by ref to const
Complex operator*(int num, const Complex &c1) {
	cout << "Inside friend function" << endl;
	return Complex(c1.re * num, c1.im * num);
}
int main() {
	Complex c1(1.1, 1.1), c2(2.2, 2.2), c3, c4, c5;
	// No error, as operator+ accepts arg by
	// ref to const
	c3 = c1 + c2 * 2;
	c4 = (c1 + c2) * 2;
	c5 = 2 * c1 * 3;
	c3.print();
	c4.print();
	c5.print();
	return 0;
}
```

```
Inside method II
Inside method I
Inside method I
Inside method II
Inside friend function
Inside method II
5.5 + j5.5
6.6 + j6.6
6.6 + j6.6
```

```
#include<iostream>

using std::cout;
using std::endl;

class Number {
	int num;
public:
	Number(int num) {
		this->num = num;
	}
	Number operator++() {
		cout << "Pre-increment:" << num << endl;
		num++;
		return *this;
	}
	Number operator++(int) {
		cout << "Post-increment:" << num << endl;
		Number temp(*this);
		num++;
		return temp;
	}
	void print() {
		cout << num << endl;
	}
};
```

```
int main() {
	Number n1(10), n2(20);
	(++++n1).print();
	n1.print();
	n2++++.print();
	n2.print();
	return 0;
}
```

```
Pre-increment:10
Pre-increment:11
12
11
Post-increment:20
Post-increment:20
20
21
```

```
#include<iostream>

using std::cout;
using std::endl;

class Number {
	int num;
public:
	Number(int num) {
		this->num = num;
	}
	Number operator++() {
		cout << "Pre-increment: " << this << ": ";
		cout << num << endl;
		num++;
		return *this;
	}
	void print() {
		cout << "print function: " << this << ": ";
		cout << num << endl;
	}
};
```

```
int main() {
	Number n1(10);
	cout << "n1 address: " << &n1 << endl;
	n1 = ++++n1;
	n1.print();
	return 0;
}
```

```
n1 address: 0x7fff40a75210
Pre-increment: 0x7fff40a75210: 10
Pre-increment: 0x7fff40a75214: 11
print function: 0x7fff40a75210: 12
```

```
#include<iostream>

using std::cout;
using std::endl;

class Number {
	int num;
public:
	Number(int num) {
		this->num = num;
	}
	Number &operator++() {
		cout << "Pre-increment: " << this << ": ";
		cout << num << endl;
		num++;
		return *this;
	}
	void print() {
		cout << "print function: " << this << ": ";
		cout << num << endl;
	}
};
```

```
int main() {
	Number n1(10);
	cout << "n1 address: " << &n1 << endl;
	++++n1;
	n1.print();
	return 0;
}
```

```
n1 address: 0x7fff92af33d4
Pre-increment: 0x7fff92af33d4: 10
Pre-increment: 0x7fff92af33d4: 11
print function: 0x7fff92af33d4: 12
```

```
// 15_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    friend void operator*(float num, Complex &second_op);
};
// It is not good coding practice to alter operands of * operator
// Because for fundamental types * does not change its operands
void operator*(float num, Complex &second_op) {
    second_op.real *= num;
    second_op.imaginary *= num;
}
int main() {
    Complex c1(1.1, 2.2);
    2 * c1; // This will change members of c1
    c1.print();
    return 0;
}
```

2.20 4.40

```
// 16_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    // error: 'void Complex::operator*(float, Complex&)'
    // must have either zero or one argument
    void operator*(float num, Complex &second_op) {
        second_op.real *= num;
        second_op.imaginary *= num;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // error: no match for 'operator*'
    // (operand types are 'int' and 'Complex')
    2 * c1;
    c1.print();
    return 0;
}
```

```
// 17_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    void operator*(float num) {
        this->real *= num;
        this->imaginary *= num;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // While using operator method,
    // first operand must be object of class
    // in which method is declared
    c1 * 2;
    c1.print();
    return 0;
}
```

```
2.20 4.40
```

```
// 18_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    void operator*(float num) {
        this->real *= num;
        this->imaginary *= num;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // error: invalid operands of types
    // 'void' and 'int' to binary 'operator*'
    // will be evaluated as (c1 * 2) * 3
    // c1 * 2 will return void
    c1 * 2 * 3;
    c1.print();
    return 0;
}
```

```
// 19_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    Complex operator*(float num) {
        this->real *= num;
        this->imaginary *= num;
        return *this;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // will be evaluated as (c1 * 2) * 3
    // c1 * 2 will return Complex object
    c1 * 2 * 3;
    c1.print();
    return 0;
}
```

```
2.20 4.40
```

```
// 20_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    // This is proper behaviour as it does not alter operands
    Complex operator*(float num) {
        Complex result(0, 0);
        result.real = this->real * num;
        result.imaginary = this->imaginary * num;
        return result;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // c1 will remain unchanged
    Complex c2 = c1 * 2;
    c1.print();
    c2.print();
    return 0;
}
```

```
1.10 2.20
2.20 4.40
```

```cpp
// 21_q.cpp
#include<iostream>
#include<iomanip>
using std::cin;
using std::cout;
using std::endl;
class Complex {
    float real, imaginary;
public:
    Complex(float real, float imaginary) {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() {
        cout << std::fixed << std::setprecision(2);
        cout << real << " " << imaginary << endl;
    }
    Complex operator*(float num) {
        Complex result(0, 0);
        result.real = this->real * num;
        result.imaginary = this->imaginary * num;
        return result;
    }
};
int main() {
    Complex c1(1.1, 2.2);
    // c1 will remain unchanged
    Complex c2 = c1 * 2 * 3;
    c1.print();
    c2.print();
    return 0;
}
```

```
1.10 2.20
6.60 13.20
```

Interesting reads

- Operator Overloading
  - https://en.cppreference.com/w/cpp/language/operators
- Why operator function for **=** operator should return reference of compatible type
  - https://stackoverflow.com/questions/42335200/assignment-operator-overloading-returning-void-versus-returning-reference-param
- Why binding temporary object to local non-const lvalue reference is not allowed while calling of non-const member function by temporary object is allowed
  - https://stackoverflow.com/questions/51338287/c-whats-the-design-philosophy-of-allowing-temporary-object-to-call-non-const