

C++ Programming

11th Study: Object-Oriented Programming (7/8)

- Operator Overloading



C++ Korea 옥찬호 (utilForever@gmail.com)

Operator overloading

Customizes the C++ operators for operands of user-defined types

Operator Overloading

- 복소수(Complex Number)는, 실수(Real Part)와 허수(Imaginary Part)의 합으로 이루어지는 수
 - 임의의 복소수는 2개의 실수 a, b 를 사용한 $a + bi$ 형식으로 표현
 - 복소수 $\alpha = a + bi$ 에 대해 복소수 $a - bi$ 를 α 의 켤레복소수라고 함
 - 연산 ($\alpha = a + bi$ 를 (a, b) 라고 한다면)
 - 항등(Equality) : $(a, b) = (c, d)$
 - 덧셈(Addition), 뺄셈(Subtraction) : $(a, b) \pm (c, d) = (a \pm c, b \pm d)$
 - 곱셈(Multiplication) : $(a, b)(c, d) = (ac - bd, ad + bc)$
 - 나눗셈(Division) : $(c, d) \neq (0, 0)$ 이라면, $\frac{(a, b)}{(c, d)} = \left(\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2} \right)$

Operator Overloading

- 복소수와 덧셈 연산 구현

```
#include <iostream>

class Complex
{
private:
    double real;
    double imaginary;

public:
    Complex()
        : real(0.0), imaginary(0.0) { }
    Complex(double _real, double _imaginary)
        : real(_real), imaginary(_imaginary) { }

    void print() const;
};
```

```
void Complex::print() const
{
    std::cout << real;
    std::cout << "+";
    std::cout << imaginary;
    std::cout << "i";
}

int main()
{
    Complex c1(10.0, 5.0), c2(4.0, 3.0);
    Complex c3 = c1 + c2;

    c1.print(); std::cout << " + ";
    c2.print(); std::cout << " = ";
    c3.print();
    std::cout << std::endl;
}
```

Operator Overloading

- 문제는, int나 double과 같은 기본 타입과는 달리 Complex 클래스는 + 연산에 대한 정의가 되어 있지 않음

```
Complex c3 = c1 + c2;
```



```
error C2676: binary '+': 'Complex' does not  
define this operator or a conversion to  
a type acceptable to the predefined operator
```

- 따라서, 별도의 함수를 만들어서 덧셈에 대한 연산을 제공

```
const Complex add(const Complex& rhs) const;
```

```
const Complex Complex::add(const Complex& rhs) const  
{  
    return Complex(real + rhs.real, imaginary + rhs.imaginary);  
}
```

Operator Overloading

- add 함수를 구현하면 다음과 같이 덧셈 연산을 할 수 있음

```
Complex c1(10.0, 5.0), c2(4.0, 3.0);  
Complex c3 = c1.add(c2);
```

- add 함수도 꽤 쓸만하지만, 뭔가 지저분하고 부자연스러움
- 복소수도 int나 double과 같은 기본 타입처럼
+ 기호를 이용해서 더할 수 있다면 더 편리할텐데...

```
Complex c1(10.0, 5.0), c2(4.0, 3.0);  
Complex c3 = c1 + c2;
```

- C++에는 프로그래머가 자신만의 + 기호를 정의할 수 있는
기능을 제공함 → 연산자 오버로딩(Operator Overloading)!

Operator Overloading

- 덧셈 연산자(+)를 사용자가 만든 클래스와 연동하도록 커스터마이즈(Customize) 할 수 있음

- 먼저, operator+를 클래스 정의에서 선언함

```
const Complex operator+(const Complex& rhs) const;
```

- operator+에 대한 구현부 정의는 add() 함수와 같음

```
const Complex Complex::operator+(const Complex& rhs) const
{
    return Complex(real + rhs.real, imaginary + rhs.imaginary);
}
```

- 이제 두 복소수를 + 기호를 이용해서 더할 수 있음

```
Complex c1(10.0, 5.0), c2(4.0, 3.0);
Complex c3 = c1 + c2;    // 14.0, 8.0
```

Operator Overloading

- 꼭 Complex 객체가 아니더라도 (a, b) 형태의 복소수도 + 연산을 할 수 있게 만들고 싶다면...?



- 우선, `std::pair`를 통해 (a, b) 를 받을 수 있게 만들

```
using tempC = std::pair<double, double>;  
Complex(tempC c)  
    : real(c.first), imaginary(c.second) { }
```

- 그 뒤에 + 연산을 시도

```
Complex c4 = c3 + tempC(2.0, 4.0); // 16.0, 12.0
```

- tempC용 operator+를 정의하지 않았는데도 덧셈을 할 수 있는 이유?
→ 컴파일러가 operator+를 처리할 때 적합한 타입을 찾는 것이 아니라
적합한 타입으로 변환할 방법이 있는지도 찾기 때문 (묵시적인 변환)

```
tempC(2.0, 4.0)  Complex(2.0, 4.0)  operator+(const Complex& rhs)
```


Operator Overloading

- 하지만 + 연산인데, 교환 법칙이 성립되지 않음

```
Complex c4 = c3 + tempC(2.0, 4.0);    // 16.0, 12.0  
Complex c5 = tempC(2.0, 4.0) + c3;    // Error
```

- 묵시적인 변환은 연산자의 좌항이 Complex 객체인 경우에만 동작하고 우항에 위치하면 동작하지 않음
- 문제는 Complex 멤버 함수로 operator+를 정의하면 객체가 우항에 있을 때만 호출된다는 데 있음
- 하지만 전역 함수로서 operator+를 정의해 특정 객체에 종속되지 않도록 하면 문제가 해결됨

Operator Overloading

```
const Complex operator+(const Complex& lhs, const Complex& rhs) {  
    Complex addedComplex;  
    addedComplex.real = lhs.real + rhs.real;  
    addedComplex.imaginary = lhs.imaginary + rhs.imaginary;  
    return addedComplex;  
}
```

- 이제 좌항 / 우항에 상관없이 + 연산을 할 수 있음

```
Complex c4 = c3 + tempC(2.0, 4.0);    // 16.0, 12.0  
Complex c5 = tempC(2.0, 4.0) + c3;    // 16.0, 12.0
```

- 참고 : 전역 함수 operator+는 Complex 클래스의 private 멤버에 접근해야 하기 때문에 friend로 선언해야 함

(friend 키워드는 다른 클래스 또는 다른 클래스의 멤버 함수에서 private나 protected로 선언된 멤버 변수 / 함수에 접근할 수 있음)

```
friend const Complex operator+(const Complex& lhs, const Complex& rhs);
```

Operator Overloading

- 나머지 연산에 대해서도 구현해 보면...

```
friend const Complex operator-(const Complex& lhs, const Complex& rhs);
friend const Complex operator*(const Complex& lhs, const Complex& rhs);

const Complex operator-(const Complex& lhs, const Complex& rhs) {
    Complex subtractedComplex;
    subtractedComplex.real = lhs.real - rhs.real;
    subtractedComplex.imaginary = lhs.imaginary - rhs.imaginary;
    return subtractedComplex;
}

// (a, b)(c, d) = (ac-bd, ad+bc)
const Complex operator*(const Complex& lhs, const Complex& rhs) {
    Complex multipliedComplex;
    multipliedComplex.real = lhs.real * rhs.real - lhs.imaginary * rhs.imaginary;
    multipliedComplex.imaginary = lhs.real * rhs.imaginary + lhs.imaginary * rhs.real;
    return multipliedComplex;
}
```

Operator Overloading

- / 연산의 경우 $(c, d) \neq (0, 0)$ 조건을 먼저 검사

```
friend const Complex operator/(const Complex& lhs, const Complex& rhs);  
  
// If (c, d) is not equal (0, 0),  
// (a, b)/(c, d) = ((ac+bd)/(c^2+d^2), (bc-ad)/(c^2+d^2))  
const Complex operator/(const Complex& lhs, const Complex& rhs)  
{  
    if (rhs.real == 0 && rhs.imaginary == 0)  
        throw std::invalid_argument("Divide by zero.");  
    Complex multipliedComplex;  
    multipliedComplex.real = lhs.real * rhs.real - lhs.imaginary * rhs.imaginary;  
    multipliedComplex.imaginary = lhs.real * rhs.imaginary + lhs.imaginary * rhs.real;  
    return multipliedComplex;  
}
```

Operator Overloading

- 축약형 산술 연산자(+=, -= 등)도 오버로딩할 수 있음
- 축약형 연산자는 기본 연산자와 두 가지가 다름
 - 첫 번째는 임시 객체를 만들지 않고 좌항 객체를 변경한다는 점
 - 두 번째는 변경된 객체에 대한 참조 타입을 결과값으로 생성한다는 점
- 선언부

```
Complex& operator+=(const Complex& rhs);  
Complex& operator-=(const Complex& rhs);  
Complex& operator*=(const Complex& rhs);  
Complex& operator/=(const Complex& rhs);
```

Operator Overloading

- 구현부

```
Complex& Complex::operator+=(const Complex& rhs)
{
    real += rhs.real;
    imaginary += rhs.imaginary;
    return *this;
}
```

```
Complex& Complex::operator-=(const Complex& rhs)
{
    real -= rhs.real;
    imaginary -= rhs.imaginary;
    return *this;
}
```

Operator Overloading

- 구현부

```
Complex& Complex::operator*=(const Complex& rhs)
{
    Complex temp;
    temp.real = real * rhs.real - imaginary * rhs.imaginary;
    temp.imaginary = real * rhs.imaginary + imaginary * rhs.real;
    real = temp.real;
    imaginary = temp.imaginary;
    return *this;
}
```

Operator Overloading

- 구현부

```
Complex& Complex::operator/=(const Complex& rhs)
{
    if (rhs.real == 0 && rhs.imaginary == 0)
        throw std::invalid_argument("Divide by zero.");
    Complex temp;
    temp.real = real * rhs.real - imaginary * rhs.imaginary;
    temp.imaginary = real * rhs.imaginary + imaginary * rhs.real;
    real = temp.real;
    imaginary = temp.imaginary;
    return *this;
}
```


Operator Overloading

- 축약형 연산자 오버로딩을 해두면 다음과 같이 이용 가능

```
Complex c6(3.0, 2.0), c7(8.0, 3.0);  
c7 -= c6;           // 5.0, 1.0  
c7 += tempC(5.0, 4.0); // 10.0, 5.0
```

- 항등에 대해서는 비교 연산자를 오버로딩하면 됨
 - 클래스의 내부 멤버에 접근 가능하도록 클래스에서 friend로 선언
 - 비교 연산자는 모두 bool 타입 결과 값을 반환하는 것이 바람직

- 선언부

```
friend bool operator==(const Complex& lhs, const Complex& rhs);
```

- 구현부

```
bool operator==(const Complex& lhs, const Complex& rhs) {  
    return ((lhs.real == rhs.real) && (lhs.imaginary == rhs.imaginary));  
}
```

Operator Overloading

- 그 외 다양한 연산자들에 대해 오버로딩을 사용할 수 있음
- 연산자 오버로딩의 자세한 내용은 다음 사이트를 참고
 - <http://en.cppreference.com/w/cpp/language/operators>
- friend 키워드의 자세한 내용은 다음 사이트를 참고
 - <http://en.cppreference.com/w/cpp/language/friend>
- 이번 시간에 만든 파일의 완성본은 다음 사이트를 참고
 - <http://www.github.com/utilForever/ModernCpp>