Seminar 08

SOLID principles. Conversion. Operator overloading.

1. SOLID.

- A mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable.

- **S**ingle-responsibility principle.
  "There should never be more than one reason for a class to change."
- **O**pen-closed principle.
  "Software entities … should be open for extension, but closed for modification"
- ***L**iskov substitution principle.*
- ***I**nterface segregation principle.*
- ***D**ependency inversion principle.*

2. Conversion constructors.

- Single parameter constructors are essentially conversion constructors. They construct an object of your class, given another type.
- They can be marked as **explicit** (which is preferred).
- Example:

```
class MyClass {
public:
    explicit MyClass (int num) : num(num) {}

 ...
};

...

MyClass obj1(42);         // OK: Uses the constructor
MyClass obj2 = 42;        // Cannot convert 42 to MyClass implicitly!
MyClass obj3 = static_cast<MyClass>(42);  // OK

...

void fun(const MyClass& obj);

...

fun(42);  // Cannot convert 42 to MyClass implicitly!
fun(static_cast<MyClass>(42));  // OK
```

3. Operator overloading.

- Changing how the operators work with objects from our classes.

- In C++ *almost* every operator can be overloaded.
- <return type> operator<operator>([parameters]);

- Operators as methods (member functions) of a class are used when the left hand-side (lhs) of the expression is an object of your class.
  Example:

```
int operator+(int num) const;  |   MyClass obj;     int res = obj + 2;
```

  Or when a unary operator is being used.
  Example:

```
bool operator!() const;        |   MyClass obj;     bool res = !obj;
```

- Operators as functions or friend functions. Mostly used when the object of your class is the right hand-size (rhs) of the expression.
  Example:

```
int operator+(int num, const MyClass& obj);
MyClass obj;     int res = 2 + obj;
```

- Most operators can be written as a method or as a function as well.

  Note: Postfix operators (++ and --) require a dummy parameter so they can be identified. Also the prefix operator returns a reference.

```
MyClass& operator++();     // Prefix operator++
MyClass  operator++(int);  // Postfix operator++
```

- Example:

```
class Complex {
public:
    Complex(int real, int imaginary)
        : real(real)
        , imag(imaginary)
    {}

    bool operator==(const Complex& other) const
    {
        return real == other.real && imag == other.imag;
    }

    Complex& operator+=(const Complex& other)
    {
        real += other.real;
        imag += other.imag;
        return *this;
    }

    Complex operator+(const Complex& other) const
```

```
        {
            return Complex(*this) += other;
        }

        friend
        std::ostream& operator<<(std::ostream& out, const Complex& obj)
        {
            return out << obj.real << " + " << obj.imag << "i";
        }

    private:
        int real;
        int imag;
    }
```

Source.cpp

```
    Complex c1(5, 6);
    Complex c2(3, 1);
    Complex c3(8, 7);
    cout << (c1 == c2);      // false
    cout << endl;
    cout << (c1 + c2 == c3); // true
    cout << endl;
    cout << c1 << endl;      // 5 + 6i
```

4. Conversion operators.

- Methods that convert objects of your class to another type.
- Can be **implicit** (that's by default) or **explicit**, which must be marked with the `explicit` keyword.
- **Explicit**, means that the compiler will NOT convert objects of your type to another type automatically. You MUST state that you want to convert your object explicitly. **Explicit is the preferred way of writing conversion operators!**

- operator <type>() const;
- Example:

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1)
        : numerator(numerator)
        , denominator(denominator)
    {}
```

```
    explicit operator double() const
    {
        return static_cast<double>(numerator) / denominator;
    }
 ...
};

...

Rational  rat(1, 2);
double num1 = rat;                        // Implicit conversion ☹
double num2 = static_cast<double>(rat);   // Explicit conversion 😁
```