Seminar 05

Static members and methods, =default, =delete and more on overloading operators

1. = default and = delete.

- = default marks a constructor or operator= to be defined by the compiler using the **default implementation** (shallow copy).
  Example:
  ```
  ClassName() = default;
  ```

- = delete marks a constructor or operator= as **non existent** for this class. I.e. this class will not have the marked constructor/operator=.
  Example:
  ```
  // This prohibits the class from being copied
  ClassName(const ClassName& other) = delete;
  ClassName& operator=(const ClassName& other) = delete;
  ```

2. Static members and methods.

- Members and methods marked as static are linked to the **class** and not to an object of that class. And thus are accessed by specifying the class first, followed by :: and then the name of the member/method. *(ClassName::member)*
- We can think of static members as global variables for the class.
  Example:

Person.h

```
class Person
{
public:
    static int publicMember;
    static const int MAX_SOMETHING;
    Person();
    static int getNumOfPeople() { return Person::numOfPeople; }
private:
    static int numOfPeople;    // Used as people counter
};
```

Person.cpp

```
#include "Person.h"

int Person::publicMember = 42; // Default values for the data members
int Person::numOfPeople  = 0;  // are defined in the source file

Person::Person()
{
    numOfPeople++;
}
```

Source.cpp

```cpp
#include "Person.h"
int main()
{
    Person p1;
    std::cout << Person::getNumOfPeople();  // 1
    Person p2;
    std::cout << Person::getNumOfPeople();  // 2
    Person p3;
    Person p4;
    Person p5;
    std::cout << Person::getNumOfPeople();  // 5

    // This static member cannot be accessed since it's private
    std::cout << Person::numOfPeople;

    // But this static member is public, so it can be accessed
    std::cout << Person::publicMember;

    // As well as changed
    Person::publicMember = 5;

    return 0;
}
```

3. More on overloading operators.

- Binary operators (two operands)
  - As a **method** of our class, the left hand side operand will be our this pointer

    `<type> operator@(const ClassName& other);`

    Example: *// a == b; a will be the* `this` *pointer and b will be* `other`

    ```cpp
    bool Complex::operator==(const Complex& other) const {
        return this->real == other.real && this->imag == other.imag;
    }
    ```

  - As a **function** using our class

    `<type> operator@([const] ClassName& lhs, const ClassName& rhs);`

    Example: *// a == b; a will be the* `lhs` *pointer and b will be* `rhs`

    ```cpp
    bool operator==(const Complex& lhs, const Complex& rhs) {
        return lhs.getReal() == rhs.getReal() &&
               lhs.getImag() == rhs.getImag();
    }
    ```

- As a **friend function** using our class
  ```
  friend <type> operator@([const] ClassName& lhs, const ClassName& rhs);
  ```
  Example:
  ```
  bool operator==(const Complex& lhs, const Complex& rhs) {
      return lhs.real == rhs.real && lhs.imag == rhs.imag;
  }
  ```

- Unary operators (only one operand)
  - As a **method** of our class, the operand will be our this pointer
    ```
    <type> operator@();
    ```
    Example: *// -a; a will be the* this *pointer*
    ```
    Complex Complex::operator-() const {
        return Complex(-real, -imag);
    }
    ```

  - As a **function** using our class
    ```
    <type> operator@([const] ClassName& obj);
    ```
    Example: *// -a; a will be* obj
    ```
    Complex operator-(const Complex& obj) {
        return Complex(-obj.getReal(), -obj.getImag());
    }
    ```

  - As a **friend function** using our class
    ```
    friend <type> operator@([const] ClassName& obj);
    ```
    Example: *// -a; a will be* obj
    ```
    Complex operator-(const Complex& obj) {
        return Complex(-obj.real, -obj.imag);
    }
    ```

- Conversion operators
  ```
  [explicit] operator <type>() const;
  ```
  We can define how our class can be converted to another type. For example if we have a class Rational that has a numerator and a denominator, it can easily be converted to a double, just by dividing the numerator and the denominator.
  Example:
  ```
  Rational::operator double() const {
      return (double)numerator / denominator;
  }

  // This way we can do something like this
  Rational rat(1, 3);  // ⅓
  double num = 0.25;

  cout << num + rat;   // Here rat will be converted to double and added
                       // to num, 0.583333 wll be printed
  ```
  Note: If `explicit` is declared before the conversion operator, objects from our class will **only** be converted if we explicitly say (<type>)obj. *Example: (double)rat + num*