# Оператори:

Да се върнем на класът `Rational`

Там имаме метода `add()`

```
Rational Rational::add(const Rational& b)
{
    Rational c;

    c.denominator = this->denominator * b.denominator;
    c.numerator = denominator*b.numerator + b.denominator*numerator;

    return c;
}
```

т.е. когато искаме да съберем две рационални числа трябва да използваме `c = a.add(b)`. Нямаше ли да е по-добре ако можем да напишем `c = a+b`. По подразбиране, това ще събере числителите поотделно и знаменателите поотделно. Това събиране е грешно. За да го поправим, трябва да променим начина по който работи оператора + за класа `Rational`.

Това действие се нарича **предефиниране на оператори**

# Предефиниране на оператори

Нека предефинираме оператора `+` на класа `Rational`

```
#pragma once
class Rational
{
private:
    int numerator;
    int denominator;

public:
    Rational(int a= 0, int b = 1); //Конструктор с два параметъра, с аргументи по
подразбиране
    Rational(const Rational&); //Конструктор за копиране
    void setNum(int a);
    void setDenom(int a);
    void print() const;

    int getNum() const;
    int getDenom() const;

    Rational add(const Rational& b) const;

    Rational operator+(Rational& b) const;
```

```
};
```

```
...
Rational Rational::operator+(Rational& rhs) const
{
    return add(rhs);
}
...
```

Всички оператори в C++ са под формата на методи. Типът на функцията ще е от тип `Rational` (Все пак, искаме да съберем две числа и да върне резултата им).

Някои полезни оператори, които могат да се предефинират:

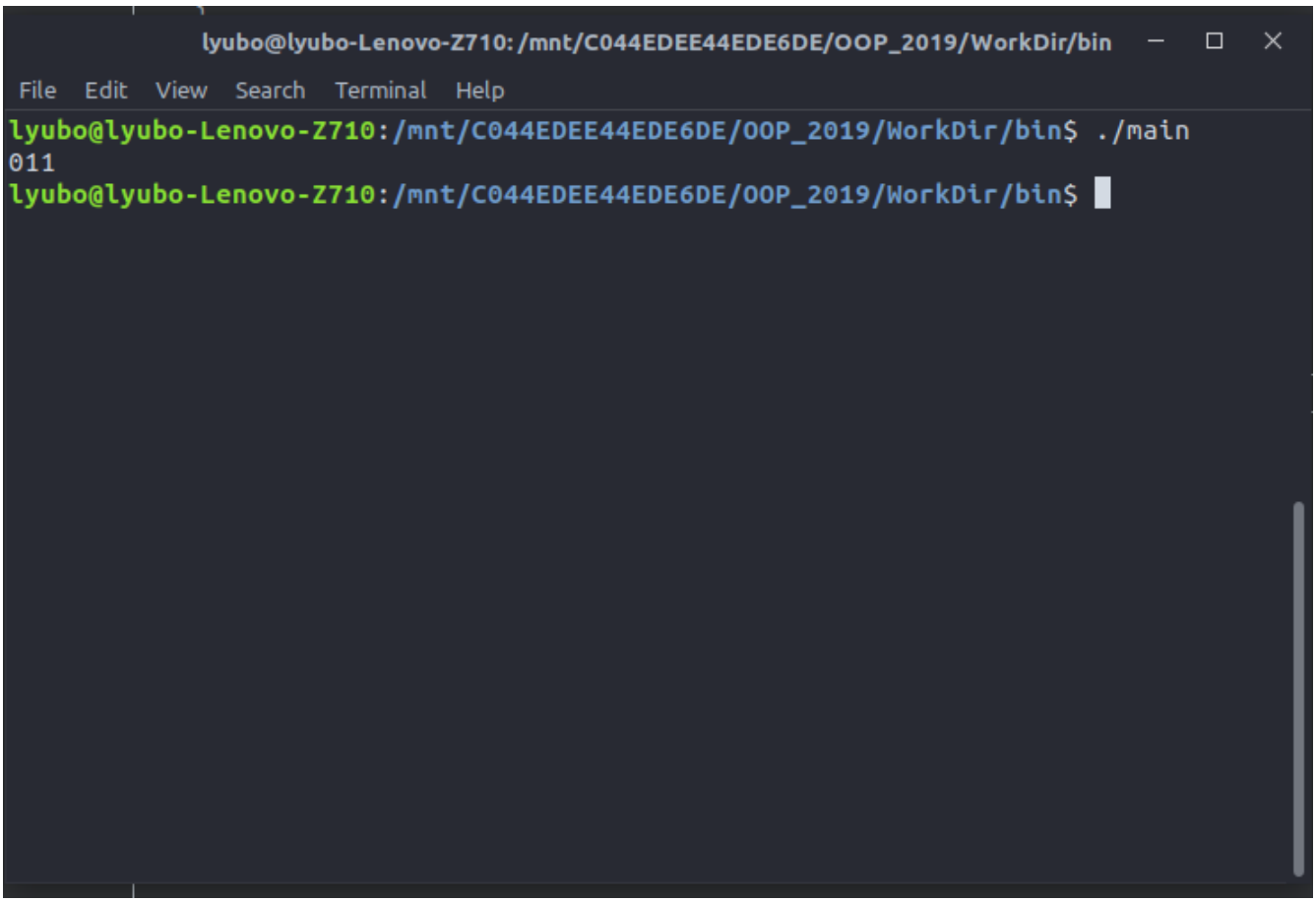
`+`, `-`, `*`, `/`, `+=`, `<<`, `>>`, `==`, `=`, `++`, `( )`, `[ ]`, `!`, `>`, `<`

Ще разгледаме примери за операторите `==`, `!=`, `<`

```
bool operator==(Rational& b) const;
bool operator!=(Rational& b) const;
bool operator<(Rational& b) const;
```

```
bool Rational::operator==(Rational& rhs) const
{
    if(numerator / denominator == rhs.numerator / rhs.denominator && numerator %
denominator == rhs.numerator % rhs.denominator)
    {
        return true;
    }
    else return false;
}
bool Rational::operator!=(Rational& rhs) const
{
    return !(*this == rhs);
}
bool Rational::operator<(Rational& rhs) const
{
    if((double)numerator / (double)denominator < (double)rhs.numerator /
(double)rhs.denominator) return true;
}
```

```cpp
#include <iostream>
#include "Rational.h"
int main() {

    Rational a(2,3);
    Rational b(3,4);

    std::cout << (a == b);
    std::cout << (a != b);
    std::cout << (a < b);
}
```




Пълен списък с операторите, които могат да се предефинират: https://en.cppreference.com/w/cpp/language/operators

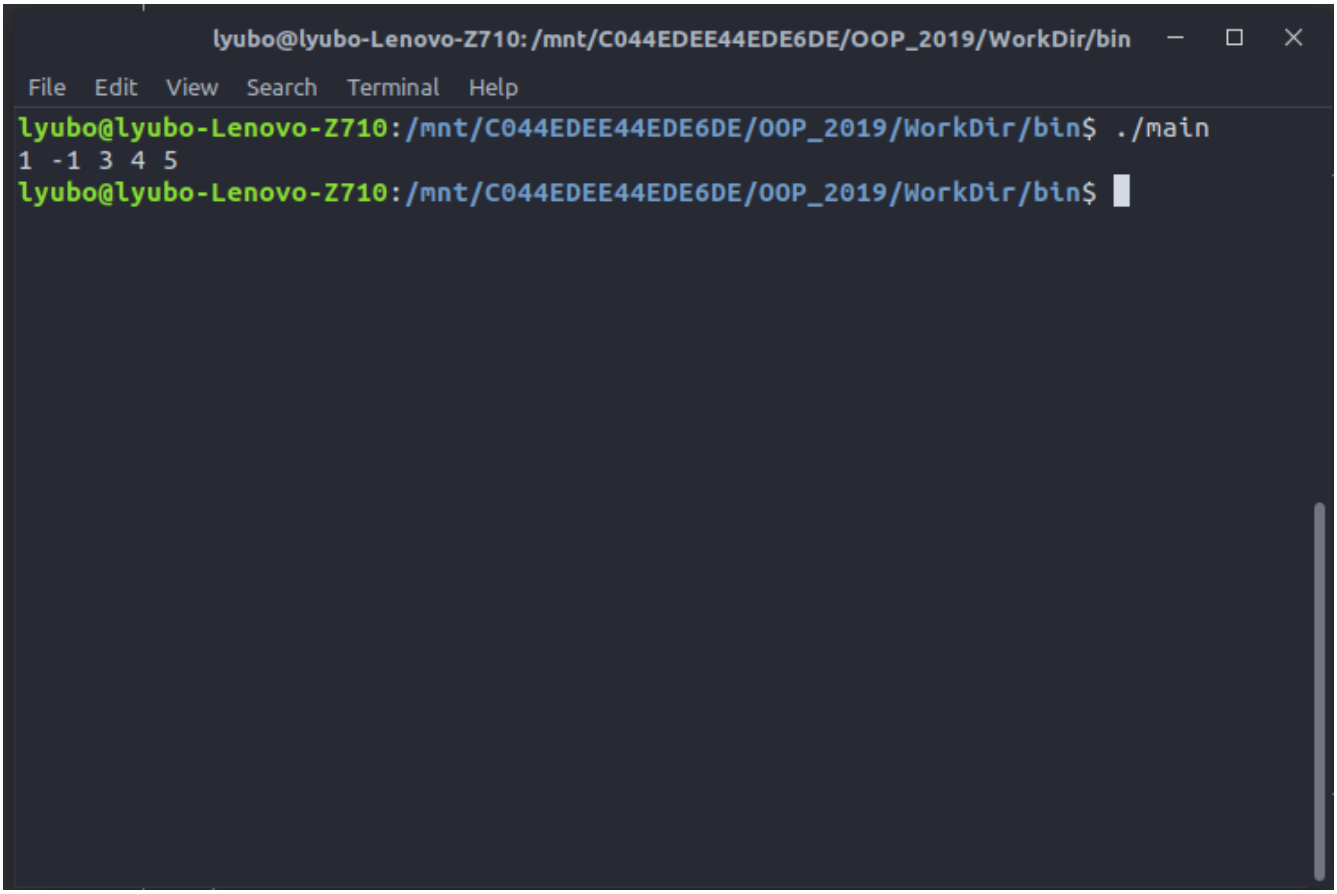
## Предефиниране на оператор =

По подразбиране, оператора `=` копира стойностите едно към едно. При използване на динамична памет това е проблем. Затова се налага да се предефинира оператора равно. Нека разгледаме как става това при нашия клас `MyVector`

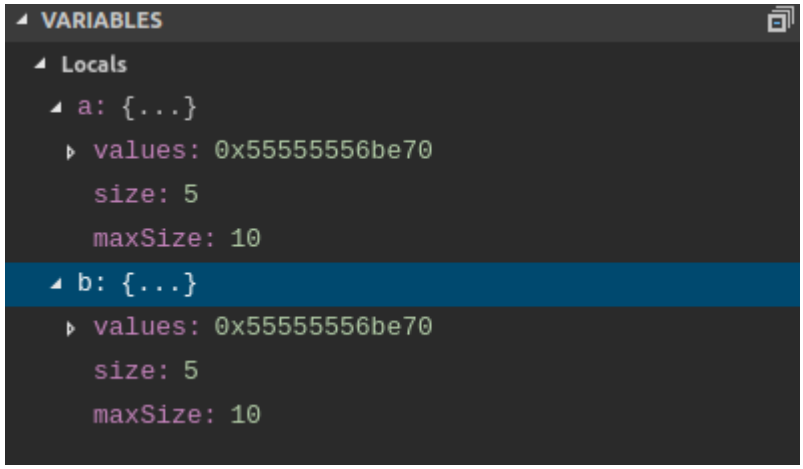
Нека имаме следният код:

```cpp
#include <iostream>
```

```cpp
#include "MyVector.h"

int main() {

    MyVector a,b;

    for(int i = 1; i < 6; i++)
        a.push_back(i);

    b = a;

    a.set_element(1,-1);
    b.print();
}
```




Виждаме пример за това как се копира паметта без да се заделя нова за втория обект и когато променил първия обект, това оказва влияние на втория.




Ако разгледаме към кой адрес сочат `values` на `a` и `b` забелязваме, че те сочат към един и същ адрес. Това може да се оправи, ако предефинираме оператора `=`, така че да **изтрива** старите стойности на обекта и заделя памет за нови.

```
MyVector& MyVector::operator=(const MyVector& rhs)
{
    if(this != &rhs)
    {
        delete[] values;

        maxSize = rhs.maxSize;
        size = rhs.size;

        values = new int[maxSize];

        for(int i = 0; i < size; i++)
        {
            values[i] = rhs.values[i];
        }
    }
    return *this;
}
```

Някои особености на оператора `=`:

- Не е конструктор, т.е. трябва да укажем типа на резултата, а именно `MyVector&` (ако разпишем пълната форма на оператора `=`, ще получим `b=b.operator=(a)`, тоест искаме да работим с обект, който е извън функцията)
- Първо проверяваме дали нямаме случай, в който даваме обект да е равен на себе си (`a=a`). Тогава няма смисъл да трием памет, да я заделяме наново, и да копираме
- Понеже обекта към когото ще копираме може да не е празен, или пък да няма достатъчно памет, първо трябва да изтрием старата памет, и след това да заделим нова
- Накрая връщаме обекта, който седи отляво на оператора `=` (в нашият случай, това е `this`)

---