Zadanie 1

Zdefiniuj klasę **Segment** reprezentującą odcinek $[A, B]$ na osi liczbowej

```
class Segment {
    double A,B;
public:
    Segment(double A, double B) : A(A), B(B) { }
    // ...
};
```

Następnie zdefiniuj odpowiednie metody i funkcje, tak, aby dla odcinka seg i liczby d typu **double**

- wartością wyrażenia d*seg lub seg*d był odcinek powstały z seg przez przeskalowanie w stosunku d (tzn. współrzędne końca i początku tego odcinka mają być równe d*A i d*B, gdzie A i B to współrzędne początku i końca odcinka seg);
- wartością wyrażenia seg/d był odcinek powstały z seg przez przeskalowanie w stosunku $\frac{1}{d}$ (odcinek seg „podzielony" przez d);
- wartością wyrażenia seg+d lub d+seg był odcinek seg przesunięty o d w prawo;
- wartością wyrażenia seg-d był odcinek seg przesunięty o d w lewo;
- wartością wyrażenia seg1+seg2 był najmniejszy odcinek zawierający odcinki seg1 i seg2;
- wartością wyrażenia seg(d) było **true** wtedy, gdy d należy do odcinka seg i **false** w przeciwnym przypadku.

Przeciąż też operator **operator<<** tak, aby następująca funkcja **main**

download *Segment.cpp*

```
int main() {
    using std::cout; using std::endl;

    Segment seg{2,3}, s = 1 + 2*((seg-2)/2+seg)/3;

    cout << s << endl << std::boolalpha;
    for (double x = 0.5; x < 4; x += 1)
        cout << "x=" << x << ": " << s(x) << endl;
}
```

wydrukowała coś w rodzaju

```
[1,3]
x=0.5: false
x=1.5: true
x=2.5: true
x=3.5: false
```

Zadanie 2

Zdefiniuj klasę **Resistor**, obiekty której reprezentują oporniki elektryczne. Klasa powinna zawierać

- jedno (prywatne) pole typu **double** — opór opornika;
- konstruktor domyślny (opornik o oporze 0);
- konstruktor ustawiający opór na wartość zadaną jego argumentem;
- metodę **double r()** zwracającą wartość oporu;
- metodę **void r(double)** modyfikującą wartość oporu.

Ponadto napisz zaprzyjaźnione funkcje przeciążające operatory

- **operator+** zwracającą przez wartość obiekt klasy **Resistor** reprezentujący opór zastępczy dwóch podanych (w postaci obiektów) oporników, zakładając, że są one połączone szeregowo;
- **operator*** — podobnie, ale dla połączenia równoległego;
- **operator<<** pozwalającą na wstawianie obiektów klasy do strumienia wyjściowego.

Na przykład fragment

```
Resistor r1, r2{6};
r1.r(3);
std::cout << (r1 + r2) << " " << (r1 * r2) << std::endl;
```

powinien wydrukować

```
9 2
```

Zadanie 3

Stwórz klasę **StackArr**, która definiuje prostą implementację stosu liczb całkowitych; implementacja oparta jest na tablicy o ustalonym rozmiarze. Klasa udostępnia:

- konstruktor domyślny tworzący stos pusty;
- konstruktor `StackArr(initializer_list<int>)` tworzący stos na podstawie liczb przekazanych w inicjatorze listowym;
- metodę `void push(int e)` wstawiającą nowy węzeł na wierzchołek stosu;
- metodę `int pop()` usuwającą i zwracającą dane z wierzchołka stosu;
- metodę `int peek()` zwracającą, ale *bez* usuwania, dane z wierzchołka stosu;
- metodę `bool empty()` zwracającą **true** wtedy i tylko wtedy, gdy stos jest pusty; w przeciwnym przypadku zwracane jest **false**.
- metodę `bool full()` zwracającą **true** wtedy i tylko wtedy, gdy stos jest pełen: żaden nowy element nie może być dodany;
- metodę `size_t avail()` zwracającą liczbę „wolnych miejsc” — liczbę elementów, które można jeszcze położyć na stosie aż stanie się „pełen”;
- przeciążenie operatora konwersji do typu **bool** — **true** oznacza, że stos nie jest pusty;

- metodę przciążającą **operator<<** tak, żeby `stos << 7 << 8;` wstawiło na stos 7 i 8;
- metodę przciążającą **operator>>** tak, aby `stack >> n >> m;` zdjęło dwie wartości ze stosu i wstawiło je do zmiennych całkowitych `n` i `m`;
- zaprzyjaźnioną wolną (globalną) funkcję przeciążającą operator wstawiania do strumienia wyjściowego obiektów definiowanego typu.

Stos jest zaimplementowany jako tablica o ustalonym rozmiarze będąca składową obiektu (a więc *nie* alokowana dynamicznie). Wygodnie jest zdefiniować składową (na przykład top), której wartość jest indeksem pierwszego „wolnego" miejsca w tablicy; jest to jednocześnie liczba elementów już wstawionych.

Nielegalne operacje (wstawianie na stos pełny, zdejmowanie ze stosu pustego itd.) powinny powodować zgłoszenie wyjątku **out_of_range**.
Na przykład, następujący program (gdzie SIZE jest celowo bardzo małe jak na praktyczne zastosowania)

download *StackArr.cpp*

```
#include <cassert>           // assert
#include <stdexcept>         // out_of_range
#include <initializer_list>
#include <cstring>           // memcpy; useful in the ctor
#include <iostream>

class StackArr {
    enum : size_t {SIZE = 8};
    int arr[SIZE];
    size_t top;    // index of the first "free" location
public:
    // ...
};

int main() {
    using std::cout; using std::endl;

    StackArr stack{1,2,3,4,5};
    stack << 6 << 7 << 8;
    cout << stack << endl;

    int a, b, c;
    stack >> c >> b >> a;
    assert(b == 7);
    assert(stack.peek() == 5);
    while (stack.avail() > 0) {
        stack.push(9);
    }
    assert(stack.full());
    assert(!stack.empty());
```

```
    while(stack.peek() > 3) {
        cout << "pop:  " << stack.pop() << endl;
    }
    cout << "Removing the rest..." << endl;
    while (stack) {
        cout << "pop:  " << stack.pop() << endl;
    }
    assert(stack.empty());
}
```

powinien wydrukować

```
[ 1 2 3 4 5 6 7 8 ]
pop:  9
pop:  9
pop:  9
pop:  5
pop:  4
Removing the rest...
pop:  3
pop:  2
pop:  1
```
