Zadanie 0 – Przeciążanie operatorów (1pkt)

Klasa Wektor reprezentuje wektor w 3D i posiada trzy składowe prywatne odpowiadające współrzędnym w układzie kartezjańskim. Zaimplementuj dla tej klasy następujące operatory:

- + – dwuargumentowy operator dodawania wektorów
- - – dwuargumentowy operator odejmowania wektorów
- * – dwuargumentowy operator mnożenia skalarnego wektorów
- == – dwuargumentowy operator sprawdzający, czy wektory są identyczne
- - – jednoargumentowy operator odpowiadający mnożeniu wektora przez -1

Szablon:

```
#include <iostream>
#include <cmath>

class Wektor
{
private:
        double x;
        double y;
        double z;
public:
        void setX(double a){x=a;}
        void setY(double a){y=a;}
        void setZ(double a){z=a;}
        double getX(){return x;}
        double getY(){return y;}
        double getZ(){return z;}
        Wektor(const double a=0., const double b=0., const double c=0.)
        :x(a), y(b), z(c){}
        double getModule(){return sqrt(x*x+y*y+z*z);}
        friend std::ostream& operator<<(std::ostream &out, const Wektor& c);
};

std::ostream& operator<<(std::ostream &os, const Wektor& w)
{
        os << "(" << w.x << "," << w.y << "," << w.z << ")";
        return os;
};

int main()
{
        Wektor w(5., -1., 3.);
        Wektor v;
        v.setX(1.);
        v.setY(2.);
        v.setZ(-3.);
```

```
        std::cout << w << std::endl << v << std::endl;
        std::cout << -w << std::endl;
        std::cout << w+v << std::endl;
        std::cout << w-v << std::endl;
        std::cout << w*v << std::endl;
        std::cout << (w==v) << std::endl;
        std::cout << (Wektor(5., -1., 3.)==w) << std::endl;
        return 0;
}
```

Output:
(5,-1,3)
(1,2,-3)
(-5,1,-3)
(6,1,0)
(4,-3,6)
-6
0
1

Zadanie 1 – Operatory dla klasy-wrappera (1pkt)

Klasa Napis jest tzw. wrapperem dla typu wbudowanego char[]. Jest to klasa posiadająca tylko jedno pole – tablicę znaków, oraz metody mające ułatwić operacje na tej tablicy. Dla klasy Napis zdefiniuj następujące operatory operator strumieniowy << wypisujący zawartość tablicy znaków oraz operatory porównania, tj. $<, <=, ==, != , >, >=$. Operatory mniejszości i większości mają na celu porównanie dwóch Napisów. Jeżeli Napis n1 jest mniejszy niż Napis n2, oznacza to, że n1 jest alfabetycznie wcześniej niż n2. Porównanie możesz zrealizować porównując kolejne znaki w obu tablicach, gdyż dla typu wbudowanego char są zdefiniowane operatory porównania. W przypadku, gdy jeden napis jest taki jak drugi, ale z dodatkowymi literami na końcu, np. ”kruk” i ”kruki”, to przyjmij, że ”kruk” ¡ ”kruki”.

Szablon

```
#include <iostream>
#include <vector>
using namespace std;

class Napis
{
private:
        char* str;
        int length;
public:
        Napis(const char* txt="")
        {
                length =strlen(txt);
                str = new char[length];
                strcpy(str, txt);
        }
        Napis(const Napis& s)
        {
```

```
                length =s.getLength();
                str = new char[length];
                const char* st = s.getStr();
                strcpy(str, st);
        }
        ~Napis()
        {
                if(str)
                        delete str;
        }
        void setStr(const char* txt)
        {
                if(str)
                        delete str;
                length =strlen(txt);
                str = new char[length];
                strcpy(str, txt);
        }
        const char* getStr() const {return str;}
        int getLength() const {return length;}
};

int main()
{
        Napis n1;
        n1.setStr("krowa");
        Napis n2("kruk");
        Napis n3("kruki");

        vector<Napis> v;
        v.push_back(n1);
        v.push_back(n2);
        v.push_back(n3);
        for(auto iter=v.begin(); iter!=v.end(); ++iter)
        {
                for(auto iter2=v.begin(); iter2!=v.end(); ++iter2)
                {
                        cout << *iter << " " << *iter2 << " "
                        << ((*iter) == (*iter2)) << " "
                        << ((*iter) != (*iter2)) << " "
                        << ((*iter) > (*iter2)) << " "
                        << ((*iter) <= (*iter2)) << " "
                        << ((*iter) < (*iter2)) << " "
                        << ((*iter) >= (*iter2)) << endl;
                }
        }
        //Dodawanie dla chetnych
        //Napis n4 = n1+n2;
        //cout<< n4 << endl;
        return 0;
```

```
}
```

Output
krowa krowa 1 0 0 1 0 1
krowa kruk 0 1 0 1 1 0
krowa kruki 0 1 0 1 1 0
kruk krowa 0 1 1 0 0 1
kruk kruk 1 0 0 1 0 1
kruk kruki 0 1 0 1 1 0
kruki krowa 0 1 1 0 0 1
kruki kruk 0 1 1 0 0 1
kruki kruki 1 0 0 1 0 1
Dla chętnych: Napisz operator dodawania dwóch Napisów, który zwróci nowy Napis powstały z dopisania jednego ciągu znaków za drugim. W zakomentowanym przykładzie n4 będzie "krowakruk".

Iteratory

Iteratory są specjalnymi obiektami, które umożliwiają poruszanie się po kolekcjach z biblioteki standardowej C++. Do tej pory poznaliśmy już iterator **std::vector¡T¿::iterator**, którego czasem używaliśmy w pętli for do iterowania po kolekcji **std::vector**.
Do zrobienia zadań z tej serii wystarczy wiedzieć, że będziemy używać iteratorów jednokierunkowych, a więc takich dla których zdefiniowane są operatory inkrementacji(++), tożsamości (==, !=) oraz dereferencji (*). Operator dereferencji użyty na iteratorze pozwala uzyskać dostęp do wskazywanego obiektu w kolekcji.
Niektóre z iteratorów użytych w przykładach posiadają więcej operatorów i dodatkowe metody, ale do zrobienia zadań wystarczy użycie wymienionych 4 operatorów.

Zadanie 2 – sortowanie bąbelkowe (1pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void bubbleSort(ForwardIt begin, ForwardIt end)
```

wykonującej sortowanie bąbelkowe pomiedzy dwoma iteratorami jednokierunkowymi typu ForwardIt.
Do zamiany wartości dwóch iteratorów a i b możesz użyć **std::iter swap(a,b)** z **¡algorithm¿** lub użyć dereferencji **std::swap(*a,*b)**.
Pamietaj, żeby zakończyć szybko sortowanie, jeżeli w poprzedniej iteracji żadne elementy nie zostały zamienione miejscami.
Szablon:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void bubbleSort(ForwardIt begin, ForwardIt end);

int main() {
```

```
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  bubbleSort(v.begin(), v.end());
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  bubbleSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //list
  std::list<int> l = { 7, 5, 16, 8 };
  bubbleSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.

Zadanie 3 – sortowanie przez wybieranie (1pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void selectionSort(ForwardIt begin, ForwardIt end)
```

wykonującej sortowanie przez wybieranie pomiedzy dwoma iteratorami jednokierunkowymi.
Do zamiany wartości dwóch iteratorów a i b możesz użyć **std::iter swap(a,b)** z **¡algorithm¿** lub **std::swap(*a,*b)**.
Do znalezienia iteratora do najmniejszego elementu pomiedzy iteratorami first i last możesz użyć funkcji **std::min_element(first,last)** z **¡algorithm¿**.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void selectionSort(ForwardIt begin, ForwardIt end);

int main() {
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  selectionSort(v.begin(), v.end());
```

```
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  selectionSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  //list
  std::list<int> l = { 7, 5, 16, 8 };
  selectionSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.

Zadanie 4 – Sortowanie przez wstawianie (1pkt)

Napisz szablon funkcji

```
template <class ForwardIt>
void insertionSort(ForwardIt begin, ForwardIt end);
```

realizującej sortowanie przez wstawianie. Nie będziesz potrzebował(a) iteratorów dwukierunkowych, jeżeli wykorzystasz następujące funkcje z biblioteki standardowej, z nagłówka **¡algorithm¿**:

- **std::upper_bound(start, stop, v)** – zwraca iterator z zakresu od iteratora start do iteratora stop, który odpowiada pierwszej napotkanej wartości większej od v.
- **std::next(it)** – zwraca następny iterator po iteratorze it (odpowiednik użycia ++, nie zmienia it).
- std::rotate(start, val, stop) – dokonuje przesunięcia w lewo dla zakresu od iteratora start do iteratora stop. Przesunięcie będzie takie, żeby po jego dokonaniu iterator val stał na początku zakresu (tam gdzie był start).

Szablon

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <forward_list>
#include <list>

template <class ForwardIt>
void insertionSort(ForwardIt begin, ForwardIt end);
```

```cpp
int main() {
  //vector
  std::vector<int> v = {6, 4, 2, 2, 3, 1};
  insertionSort(v.begin(), v.end());
  for (auto i : v) {
    std::cout << i << " ";
  }
  std::cout << std::endl;

  //forward_list
  std::forward_list<int> mylist = { 34, 77, 16, 2 };
  insertionSort(mylist.begin(), mylist.end());
  for (auto i : mylist) {
    std::cout << i << " ";
  }
  std::cout << std::endl;

  //list
  std::list<int> l = { 7, 5, 16, 8, -3 };
  insertionSort(l.begin(), l.end());
  for (auto i : l) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Uwaga: Nie tworzyć nowych tablic ani kolekcji.