Zadanie 0 – metoda bisekcji (1pkt)

Napisz szablon funkcji

```
template <class T>
bool bisection(double &x, T f, double from, double to, double eps);
```

znajdujący metodą bisekcji pierwiastek x równania z jedną niewiadomą f w przedziale od from do to. Procedura znajdowania pierwiastka powinna trwać do osiągniecia dokładności eps (to-from <eps). Sprawdź, czy wartości przekazane do argumentów są sensowne (from < to, 0 < eps < to-from) oraz spełnione są założenia. Jeżeli tak, funkcja powinna zwrócić wartość true i przypisać znaleziony pierwiastek do x. W innym przypadku zwrócona powinna zostać wartość false.
Przykład:

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

template <class T>
bool bisection(double &x, T f, double from, double to, double eps);

double f1(double x) { return x * x; }
double f2(double x) { return x * x - 2.; }
double f3(double x) { return exp(x) + x - 1; }

int main()
{

        double x;
        for (auto fx : {f1, f2, f3})
        {
                for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001})
                {
                        if (bisection(x, fx, -1, 6, eps))
                        {
                                std::cout << std::setprecision(8) << "eps = " << eps
                                << "\t root = " << x << std::endl;
                        }
                        else
                        {
                                std::cout << "Unable to find root" << std::endl;
                                break;
                        }
                }
                std::cout << std::endl;
        }
        return 0;
}
```

Przykładowy output: Unable to find root

eps = 0.1 root = 1.4335938

eps = 0.01 root = 1.4165039
eps = 0.001 root = 1.4143677
eps = 0.0001 root = 1.4142342
eps = 1e-07 root = 1.4142136

eps = 0.1 root = 0.01171875
eps = 0.01 root = 0.0014648438
eps = 0.001 root = 0.00018310547
eps = 0.0001 root = -3.8146973e-06
eps = 1e-07 root = 1.8626451e-08
Uwaga: Nie tworzyć nowych tablic czy kolekcji wewnątrz funkcji bisection. Nie korzystać z gotowych implementacji algorytmu. Wartości w outpucie niekoniecznie muszą być identyczne z przykładem, ale powinny zbiegać do prawdziwych rozwiązań.

Zadanie 1 – liczby zespolone (1pkt)

Napisz klasę Complex, która będzie reprezentować liczbę zespoloną. Klasa powinna mieć 4 pola: część rzeczywistą, część urojoną, moduł, fazę. Powinna mieć dwa konstruktory: domyślny (bez argumentów), który ustawia wszystkie pola na 0, oraz konstruktor przyjmujący dwa parametry będące częścią rzeczywistą i urojoną liczby zespolonej. Dodatkowo klasa powinna zawierać dwie metody:

```
void Complex::printAlg()
```

drukującą na standardowe wyjście algebraiczną postać liczby zespolonej, tj. $x \pm i * y$. Oraz drugą metodę:

```
void Complex::printExp()
```

drukującą liczbę zespoloną w postaci eksponencjalnej, tj. $R * exp(i * \phi)$. Jeżeli faza jest ujemna, to przed jednostką urojoną powinien stać minus, jeżeli dodatnia to nic nie powinno stać.
Dla ułatwienia, wszystkie pola i metody klasy mogą mieć specyfikator dostępu *public*.
Dodatkowo, uzupełnij maina tak, żeby utworzyć $std :: vector$ o nazwie v, który będzie zawierał obiekty typu Complex. Następnie dodaj do niego elementy odpowiadające liczbom:
$(\sqrt{3}/2, 1/2), (-1/2, \sqrt{3}/2), (-1, 0), (0, -1)$

Szablon programu:

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

//tu napisz klase Complex

int main()
{
        //tu stworz wektor i wypelnij

        for(auto iter = v.begin(); iter!=v.end(); iter++)
        {
                iter -> printAlg();
                iter -> printExp();
                std::cout << std::endl;
        }
```

```
        return 0;
}
```

Output:
0.866025 + i*0.5
1*exp(i0.523599)

-0.5 + i*0.866025
1*exp(i2.0944)

-1 + i*0
1*exp(i3.14159)

0 - i*1
1*exp(-i1.5708)

Uwaga: Nie trzeba alokować pamięci dynamicznie, ale jeśli ktoś chce, to musi pamiętać o jej zwolnieniu.
Podpowiedź: Do liczenia fazy może się przydać funkcja *tan2*(). W konstruktorach pomocne może być użycie wskaźnika *this*. Wektor typu Complex tworzy i wypełnia się dokładnie w taki sam sposób jak wektory typów wbudowanych.
Dla chętnych: Napisz funkcje, które będą dodawać, odejmować, mnożyć, dzielić 2 liczby zespolone. Można to zrobić tak, że funkcje będą metodami klasy Complex, przyjmującymi jeden argument typu Complex (albo Complex*). Wtedy dodanie dwóch liczb A i B wyglądać będzie: "A.add(B);" i w wyniku tego wywołania liczba A będzie teraz sumą starego A i B.

Zadanie 2 – metody Newtona i Steffensena (1pkt)

Napisz szablony funkcji

```
template <class T>
bool newton(double &x, T f, double x0, double eps, int n = 1000);

template <class T>
bool steffensen(double &x, T f, double x0, double eps, int n = 1000);
```

które znajdą pierwiastek x funkcji f, zaczynając od punktu x0. Szukanie ma odbywać się do osiągnięcia dokładności mniejszej niż eps ($|x_{n+1} - x_n| < \text{eps}$) lub do przekroczenia maksymalnej liczby iteracji n. W pierwszym przypadku funkcja ma zwrócić *true*, w drugim przypadku lub w razie błędnych danych funkcja ma zwrócić $false$. W obu przypadkach najlepsze przybliżenie pierwiastka powinno zostać przypisane do x.
Przydatne wzory:

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)} \tag{1}$$

$$g_{Newt}(x_n) = f'(x_n) \tag{2}$$

$$g_{Steff}(x_n) = \frac{f(x_n + f(x_n)) - f(x_n)}{f(x_n)} \tag{3}$$

Szablon:

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

```
template <class T>
bool newton(double &x, T f, double x0, double eps, int n = 1000);

template <class T>
bool steffensen(double &x, T f, double x0, double eps, int n = 1000);

double f1(double x) { return x * x; }

double f2(double x) { return x * x - 2.; }

double f3(double x) { return exp(x) + x - 1; }

int main() {
  double x;
  std::cout<<"Newton :"<<std::endl;
   for (auto fx : {f1, f2, f3}) {
      for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
        if (newton(x, fx, 1.4, eps)) {
          std::cout << std::setprecision(8) << "\teps = " << eps
                    << "\troot = " << x << std::endl;
        } else {
          std::cout << "Unable to find root" << std::endl;
          break;
        }
      }
      std::cout << std::endl;
    }
    std::cout<<"Stefensen :"<<std::endl;
   for (auto fx : {f1, f2, f3}) {
      for (auto eps : {0.1, 0.01, 0.001, 0.0001, 0.0000001}) {
        if (steffensen(x, fx, 1.4, eps)) {
          std::cout << std::setprecision(8) << "\teps = " << eps
                    << "\troot = " << x << std::endl;
        } else {
          std::cout << "Unable to find root" << std::endl;
          break;
        }
      }
      std::cout << std::endl;
    }
}
```

Output:

Newton :
eps = 0.1 root = 0.0875
eps = 0.01 root = 0.00546875
eps = 0.001 root = 0.00068359375
eps = 0.0001 root = 8.5449219e-05
eps = 1e-07 root = 8.3446503e-08

eps = 0.1 root = 1.4142857
eps = 0.01 root = 1.4142136
eps = 0.001 root = 1.4142136
eps = 0.0001 root = 1.4142136
eps = 1e-07 root = 1.4142136
eps = 0.1 root = 0.00129105
eps = 0.01 root = 4.1679256e-07
eps = 0.001 root = 4.3299152e-14
eps = 0.0001 root = 4.3299152e-14
eps = 1e-07 root = 4.5362831e-19
Stefensen :
eps = 0.1 root = 0.070547949
eps = 0.01 root = 0.0093676589
eps = 0.001 root = 0.00059063152
eps = 0.0001 root = 7.3867098e-05
eps = 1e-07 root = 7.2141161e-08
eps = 0.1 root = 1.4144928
eps = 0.01 root = 1.4142137
eps = 0.001 root = 1.4142137
eps = 0.0001 root = 1.4142136
eps = 1e-07 root = 1.4142136
eps = 0.1 root = 1.343199
eps = 0.01 root = 1.0932516e-07
eps = 0.001 root = 1.0932516e-07
eps = 0.0001 root = 8.8532142e-15
eps = 1e-07 root = -2.8570037e-17

Zadanie 3 – Baza studentów (1pkt)

Napisz definicję klasy Student. Klasa ma zawierać następujące składowe:

- Pola typu char[100] o nazwach name, surname
- Pole typu unsigned o nazwie albumNo
- Pole typu std::vector<double>* o nazwie marks
- Konstruktor domyślny, ustawiający "" dla łańcuchów znaków, 0 dla nru albumu oraz tworzący na stosie nowy wektor i przypisujący wskaźnik do niego polu marks.
- Konstruktor przyjmujący jako parametry imię, nazwisko i nr albumu. Również ma utworzyć na stosie wektor i przypisać do składowej.
- Konstruktor kopiujący
- Metody typu "set" i "get" dla imienia, nazwiska i nru albumu.
- Metodę typu get dla wektora z ocenami.
- Metodę **void addMark(double m)**, która doda nową ocenę.
- Destruktor zwalniający pamięć po wektorze ocen.

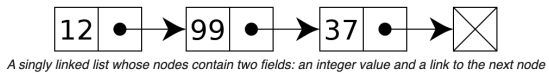
Wszystkie pola mają mieć specyfikator dostępu ”private” a metody ”public”.
W funkcji main stwórz wektor obiektów typu Student i wypełnij go przynajmniej 4 obiektami, przetestuj przy tym zarówno możliwość podawania wartości pól do konstruktora, jak i korzystanie z konstruktora domyślnego i późniejsze ustawienie wartości pól za pomocą metod set. W ten sposób dostaniemy naszą prymitywną bazę danych. Dla każdego ze studentów dodaj przynajmniej po trzy różne oceny od 2 do 5 włącznie (możesz zrobić to ręcznie, użyć generatora liczb pseudolosowych, wykorzystać pętlę for, wymyślić coś innego). Na koniec użyj pętli for (lub for each) żeby wypisać w czytelny sposób na std::cout pierwszą literę imienia studenta, nazwisko i oceny.
Przykładowy output: J. Kowalski 2.94 2.81 2.72 A. Nowak 4.5 2 3 R. Grzyb 2.34 2.98 3.76
Trzeba pamiętać, że łańcuchy znaków są tablicami. Aby przepisać łańcuch znaków do pola trzeba użyć wbudowanej funkcji strcpy. Popatrz na fragment rozwiązania, jeden z konstruktorów:

```
Student(const char* name, const char* sur, unsigned no)
{
        strcpy(this->name, name);
        /...
}
```

Zauważ, że argumenty konstruktora mają typ ”const char*”.
Uwaga: W mainie nie musisz tworzyć wektora ani obiektów na stosie, tylko dla pola ”marks”. Pamiętaj, żeby zaimplementować zwolnienie pamięci w destruktorze.
Podpowiedź: Wektory od typów zdefiniowanych przez programistę tworzymy tak samo jak od typów wbudowanych. Aby można było to zrobić, musi być zdefiniowany konstruktor kopiujący! Jeżeli nie wiesz jak to zrobić, albo jak iterować po wektorze pętlą for, zapoznaj się z ćwiczeniami do tej serii. W teoretycznym minimum są informacje o specyfikatorach dostępu oraz metodach typu ”set” i ”get”.
Dla chętnych: Niech oceny studentów będą losowane z rozkładu normalnego o ustalonej średniej i odchyleniu. Dodaj metodę liczącą średnią ocen studenta. Stwórz populację studentów o większej liczebności i policz średnią ocen w populacji.

Zadanie 4 – lista jednokierunkowa (1pkt)

Lista jednokierunkowa to uporządkowana struktura danych, w której każdy element zawiera informację o elemencie następnym. Taka konstrukcja umożliwia iterację po elementach, wystarczy mieć dostęp do pierwszego elementu listy, żeby jeden po drugim dojść do dowolnego elementu położonego dalej. Ideę listy jednokierunkowej ilustruje poniższy obrazek:



Rysunek 1: Przykład listy jednokierunkowej. Każdy element zawiera liczbę całkowitą typu integer oraz łączę (link) do następnego elementu w liście. Ostatni element listy nie wskazuje na kolejny element, tylko na umowny ”koniec”. Źródło: `https://en.wikipedia.org/`

Zauważmy, że w liście jednokierunkowej możemy iterować jedynie w jednym kierunku (stąd nazwa). Ostatni element listy powinien wskazywać na umowny ”koniec”, aby zapewnić poprawne działania. Innym rozwiązaniem jest stworzenie listy cyklicznej, w której ostatni element wskazuje na pierwszy.
W C++ można bardzo łatwo stworzyć listę obiektów. Wystarczy wziąć jakąś klasę, powiedzmy T, i dodać do niej nowe pole o typie wskaźnikowym wskazującym na obiekt tej klasy, czyli pole o typie T*. Tworząc nowe obiekty klasy T, zapisujemy do wskaźników informacje o następnym elemencie aż do ostatniego. Ostatniemu elementowi listy przypisujemy nullptr, jako wskaźnik na kolejny element. Po utworzeniu listy nie potrzebujemy już informacji o położeniu obiektów w pamięci, czyli nie potrzebujemy zmiennych czy

wskaźników na wszystkie elementy. Wystarczy nam dostęp do pierwszego elementu listy, z którego możemy dostać się do drugiego, z niego do trzeciego itd. Kiedy natrafimy na element, którego pole-wskaźnik będzie miało wartość nullptr, będziemy wiedzieć, że jesteśmy w ostatnim elemencie listy i będziemy mogli zakończyć pętlę. Łatwo można się domyślić jak zrealizować listę dwukierunkową, wystarczy dodać drugie pole-wskaźnik, tym razem wskazujące na element poprzedzający.
Po co używać list? Dla wydajności i elastyczności. Używając dynamicznej alokacji pamięci możemy tworzyć listy obiektów i nie musimy zachowywać do nich wskaźników, tylko do pierwszego elementu. Następnie, gdy chcemy usunąć element z listy, powiedzmy element k-ty, to zwalniamy po nim pamięć i zmieniamy pole-wskaźnik elementu (k-1)-tego, żeby wskazywało na element (k+1)-szy. Co ważne, nie musimy przesuwać elementów w pamięci! Listy można łączyć i zapętlać w prosty sposób. Inny przykład to sortowanie, w przypadku list nie przenosi ani nie kopiuje się żadnych elementów, a jedynie zmienia połączenia między nimi.
W zadaniu uzupełnij poniższe funkcje, aby zaimplementować funkcjonalność listy jednokierunkowej dla klasy Kontener.

```
void dodajEl(Kontener* biezacy, Kontener* nowy)
```

Ta funkcja ma dodać do listy nowy element o nazwie ”nowy”. Nowy element ma być dodany za elementem ”biezacy”. Sprawdź, czy któryś ze wskaźników jest NULL. Funkcja ma w taki sam sposób realizować zarówno dodawanie elementów na końcu listy, jak i w środku. Dodawania na początku nie trzeba uwzględniać.

```
void wypiszListe(Kontener* pocz)
```

Ta funkcja wypisuje na standardowe wyjście wartości pola ”no” dla kolejnych elementów listy, zaczynając od ”pocz”. Kolejne wartości powinny być oddzielone spacjami a na koniec powinno nastąpić przejście do nowej linii (patrz przykładowy output). Tu również zabezpiecz się przed nullptr.

```
Kontener* usunElPoID(Kontener* pocz, unsigned no)
```

To najtrudniejsza z funkcji do napisania. Funkcja ta dostaje (umowny) początek listy dla którego porównuje wartość elementu listy z podanym do niej argumentem ”no”. Jeżeli obydwie wartości są równe, element jest usuwany za pomocą operatora *delete*, a połączenia w pozostałych są ustawiane tak, żeby lista nadal działała poprawnie. Procedura jest powtarzana dla wszystkich elementów listy, aż do napotkania nullptr. Skutkiem działania jest usunięcie **wszystkich** elementów listy o podanej wartości pola ”no”, a nie tylko pierwszego. Jeżeli żaden z elementów nie ma pożądanej wartości pola ”no” to funkcja zostawi listę niezmienioną. Na koniec funkcja ma zwrócić wskaźnik na początek listy. Jest to nam potrzebne, bo szczególnym przypadkiem jest, gdy pierwszy (lub kilka pierwszych) elementów listy są do usunięcia. Będziesz potrzebował(a) w tej funkcji dużo if/else oraz pętli while/do...while.

Przykład:

```
#include <iostream> //nie ma spacji przed >
using namespace std;

class Kontener
{
        public:
                unsigned no;
                Kontener* nast;
                Kontener()
                {
                        this->no = 0;
```

```
                        this->nast = nullptr;
                };
                Kontener(unsigned no)
                {
                        this->no = no; //ta sama nazwa rozne rzeczy!
                        this->nast = nullptr;
                };
};

void dodajEl(Kontener* biezacy, Kontener* nowy)
{
        // dodaj element "nowy" do listy za "biezacy"
}

void wypiszListe(Kontener* pocz)
{
        //wypisuje pola no dla elementow listy
        //oddziela je spacja, na koniec przechodzi do nowej linii
}

Kontener* usunElPoID(Kontener* pocz, unsigned no)
{
 //usuwa wszystkie elementy o podanym no
 //i zwraca wskaznik na poczatek listy
}

int main()
{
        //pierwszy element listy
        Kontener* pocz = new Kontener();
        Kontener* biezacy = pocz;
        Kontener* dzies = nullptr;

        // dodaj elementy do listy
        for(unsigned ii=1; ii<33; ii++)
        {
                Kontener* nowy = new Kontener(ii%10);
                dodajEl(biezacy, nowy);
                biezacy = nowy;
                if(ii==10)
                        dzies = biezacy;
        }
        //wypisz cala liste
        wypiszListe(pocz);

        //dodaj po elemencie ii=10
        dodajEl(dzies, new Kontener(100));
        wypiszListe(pocz);
        //wypisz od elementu ii=10
        wypiszListe(dzies);
```

```
        //usun wszystkie elementy z id=2 po elemencie ii=10
        usunElPoID(dzies, 2);
        wypiszListe(pocz);

        //dodaje element z id=0 po pierwszym
        dodajEl(pocz, new Kontener(0));
        wypiszListe(pocz);
        for(unsigned ii=0; ii<=11; ii++)
        {
                //usun wszystkie elementy z id=ii
                pocz = usunElPoID(pocz, ii);
                wypiszListe(pocz);
        }
        //usuwamy ostatni element jaki zostal
        pocz = usunElPoID(pocz, 100);
        //probujemy wypisac pusta liste
        wypiszListe(pocz);
        return 0;
}
```

Output:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 100 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 100 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 100 1 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1
0 0 1 2 3 4 5 6 7 8 9 0 100 1 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1
1 2 3 4 5 6 7 8 9 100 1 3 4 5 6 7 8 9 1 3 4 5 6 7 8 9 1
2 3 4 5 6 7 8 9 100 3 4 5 6 7 8 9 3 4 5 6 7 8 9
3 4 5 6 7 8 9 100 3 4 5 6 7 8 9 3 4 5 6 7 8 9
4 5 6 7 8 9 100 4 5 6 7 8 9 4 5 6 7 8 9
5 6 7 8 9 100 5 6 7 8 9 5 6 7 8 9
6 7 8 9 100 6 7 8 9 6 7 8 9
7 8 9 100 7 8 9 7 8 9
8 9 100 8 9 8 9
9 100 9 9
100
100
100
[ERROR] Nullptr!

Uwaga: Nie zmieniać definicji klasy ani maina. Nie tworzyć żadnych tablic ani kolekcji. Zmieniając listę zawsze pamiętaj o przełączeniu pól-wskaźników w odpowiednich elementach. Pole-wskaźnik ostatniego elementu ma być nullptr.

Podpowiedź: Uważaj na wskaźnik nullptr! W tym zadaniu należy go używać, ale trzeba być przy tym precyzyjnym. Zawsze się upewnij, że dany element istnieje zanim spróbujesz odczytać jego pole.

Dla chętnych: Pobaw się listami. Zaimplementuj listę dwukierunkową i/lub cykliczną i dostosuj napisane funkcje. Napisz funkcję, która usuwa n-ty element listy licząc od początku (lista jednokierunkowa/dwukierunkowa) albo od końca (konieczna lista dwukierunkowa albo 2 iteracje).