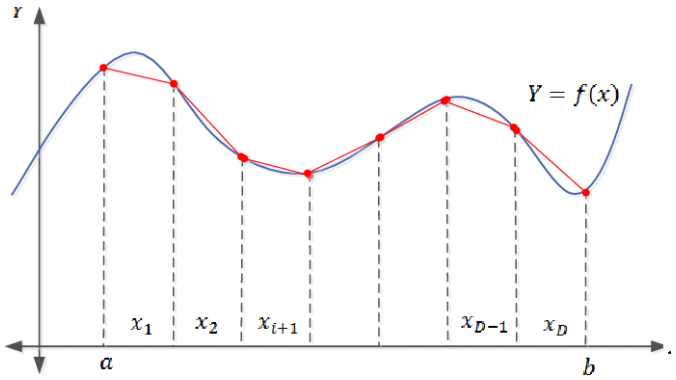
Zadanie 0 – całkowanie metodą trapezów

Całkowanie metodą trapezów jest udoskonaleniem metody prostokątów, pozwalającym osiągnąć trochę większą dokładność. Algorytm jest taki sam jak dla metody prostokątów (patrz zadanie 1 z poprzedniej serii), ale zamiast brać wartość funkcji na środku przedziału, mnożymy szerokość przedziału przez średnią arytmetyczną wartości funkcji na krańcach przedziału, co odpowiada liczeniu pola trapezu pod wykresem krzywej.



Rysunek 1: Wizualizacja całkowania metodą trapezów. Zródło: Google Image.

Napisz funkcję **calka**, która przyjmuje 5 argumentów:

- tablicę wskaźników na funkcje typu **double funkcja(double x)**.
- rozmiar tablicy
- dolną granicę całkowania jako liczbę typu double
- górną granicę całkowania jako liczbę typu double
- całkowitą liczbę przedziałów

Funkcja calka nie powinna zwracać żadnej wartości (typ void) tylko wyświetlić wynik na std::cout. Funkcja calka() ma radzić sobie z sytuacją, gdy w tablicy będzie wskaźnik pusty i ma w takiej sytuacji wyświetlić stosowny komunikat i kontynuować obliczenia dla następnej funkcji w tablicy. Oznacza to, że tym razem nie używamy *exit*() do zatrzymania programu. Funkcja calka() ma radzić sobie w sensowny sposób również z przypadkiem, gdy liczba przedziałów będzie niepoprawna. Przypadek, gdy dolna granica przedziału całkowania jest większa od górnej również musi być uwzględniony. Funkcja ma wyświetlać wynik w następującej postaci:
Calka z funkcji nr 2 od 0 do 3.14159 wynosi 0.0

Przykładowy kod:

```
#include <iostream> //Tu nie moze byc spacji przed >
#include <cmath>     //Tu nie moze byc spacji przed >

using namespace std;

double sin2(double x)
{
        return sin(x)*sin(x);
}
```

```
\\ Tu napisz definicje funkcji calka

int main()
{
        double (*ftab[])(double) = {sin, cos, nullptr, nullptr, sin2};
        calka(0.0, M_PI, ftab, 5, 10);
        cout << endl;
        calka(0.0, -M_PI, ftab, 5, 1000);
        cout << endl;
        calka(5.0, 5.0, ftab, 5, 1000);
        cout << endl;
        calka(0.0, M_PI, ftab, 5, -6);
        cout << endl;

        return 0;
}
```

Output:

Calka z funkcji nr 0 od 0 do 3.14159 wynosi 1.98352
Calka z funkcji nr 1 od 0 do 3.14159 wynosi 1.04636e-16
Na pozycji nr 2 jest nullptr!
Na pozycji nr 3 jest nullptr!
Calka z funkcji nr 4 od 0 do 3.14159 wynosi 1.5708

Calka z funkcji nr 0 od 0 do -3.14159 wynosi 2
Calka z funkcji nr 1 od 0 do -3.14159 wynosi 3.7669e-17
Na pozycji nr 2 jest nullptr!
Na pozycji nr 3 jest nullptr!
Calka z funkcji nr 4 od 0 do -3.14159 wynosi -1.5708

Calka z funkcji nr 0 od 5 do 5 wynosi -0
Calka z funkcji nr 1 od 5 do 5 wynosi 0
Na pozycji nr 2 jest nullptr!
Na pozycji nr 3 jest nullptr!
Calka z funkcji nr 4 od 5 do 5 wynosi 0

Liczba przedzialow musi byc wieksza od 0!

Uwaga: Jeżeli komuś nie kompiluje się przez M_PI to można wpisać 3,14159.
Podpowiedź: To zadanie jest bardzo podobne do zadania 1 z serii 6. Zmiana polega na tym, że mamy tablicę wskaźników na funkcję oraz wypisywanie wyniku zostało przeniesione do jej ciała. Wskaźniki funkcyjne można znaleźć w materiałach do poprzedniej serii.

Zadanie 1 – Interpolacja liniowa

Napisz funkcję

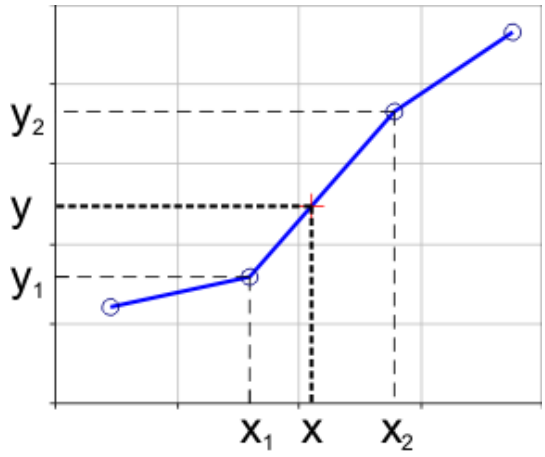
```
double linear(const std::vector<double> * const px,
```

```
                const std::vector<double> * const py, const double x)
```

która wykonywać będzie interpolację liniową w punkcie x dla funkcji zadanej przez wartości py w punktach px. Najelepiej zrobić to w dwóch krokach:

1. Znajdź wartości w px, które są najbliższe x, tzn. dwa argumenty funkcji, z których jeden jest największym elementem px mniejszym od x, a drugi jest najmniejszym elementem px większym od x. Szczególnym przypadkiem do rozważenia jest, gdy x należy do px.

2. Dla znalezionych sąsiadów x napisz równanie prostej i wykorzystaj je, żeby znaleźć wartość tej prostej w punkcie x (dokonaj interpolacji funkcji prostą).



Rysunek 2: Przykład interpolacji liniowej. Źródło: Google Image.

Przykładowy kod:

```
#include <iostream> //nie wolno spacji przed >
#include <vector>
#include <cmath>

/*
const std::vector<double> * const  oznacza staly wskaznik
 (nie mozna zmienic na co wskazuje) na stala
 (nie mozna zmienic wartosci elementu wskazywanego)
*/
double linear(const std::vector<double> * const px,
                const std::vector<double> * const py, const double x)
{
        // uzupelnij
}

int main()
{
        std::vector<double> px{2, -1, 0, 3};
        std::vector<double> py{7, 1, 1, 25};
        std::vector<double> x{0, -0.5, 1, 2.2};
        for (auto i : x)
        {
```

```
                std::cout << "p(" << i << ") = "
                          << linear(&px, &py, i) << std::endl;
        }
}
```

Output:
p(0) = 1
p(-0.5) = 1
p(1) = 4
p(2.2) = 10.6

Uwaga: Nie tworzyć nowych tablic czy kolekcji wewnątrz funkcji linear. Można założyć, że wskaźniki przekazywane do funkcji linear nie są puste. Można założyć, że przekazywane wektory zawsze zawierają co najmniej dwa punkty i wszystkie punkty są zawsze różne. Należy rozważyć przypadek, gdy x należy do px. Nie wolno zakładać posortowania wektorów. Nie wolno zakładać, że wartości należą do jakiegoś przedziału (np. od INT_MIN do INT_MAX).
Podpowiedź: Pierwsza część zadania jest podobna do znanego już problemu szukania minimum i maximum wraz z ilością wystąpień. Tym razem nie interesuje nas ilość wystąpień tylko indeksy w wektorze.

Zadanie 2 – kwadratury Gaussa

Napisz funkcję (lub szablon funkcji)

```
double gaussQuad(double from, double to, std::vector<double> w,
std::vector<double> x, ...);
```

która w ostatnim argumencie powinna pobierać wskaźnik funkcyjny (odpowiadający funkcji: double f(double)).
Funkcja wykonywać powinna całkowanie metodą kwadratur Gaussa w zakresie od from do to. Na zajęciach nie omawialiśmy metod znajdowania pierwiastków wielomianów, dlatego zarówno pierwiastki jak i wagi przekazywane powinny być do funkcji w wektorach x i w. Gotowe wartości pierwiastków i wag dla wielomianów Legendre'a podano w przykładowym mainie. Kwadratura Gaussa:

$$\int_a^b F(x)dx = \int_a^b w(x)f(x)dx \approx \sum_{i=0}^{n} \frac{b-a}{2} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right) \tag{1}$$

Wartość kwadratury jest dokładna, gdy f(x) da się dobrze przybliżyć przez wielomian stopnia równego lub mniejszego 2n + 1. Gdy zmienność funkcji w zadanym zakresie jest duża, zamiast przybliżać funkcję wielomianami co raz wyższego stopnia można podzielić zakres na mnniejsze obszary (jak w metodzie trapezów) i przybliżać na nich wielomianami niskiego stopnia.

Przykład:

```
#include <cmath> //zadnych spacji przed >
#include <iostream>
#include <vector>

double gaussQuad(double from, double to, std::vector<double> w,
std::vector<double> x, ...);

double f1(double x) { return 2 * x * x + x + 2; }
```

```
double f2(double x) { return cos(x) * x; }

double poly(double x) {
        double sum = 0;
        for (int i = 1; i != 9; ++i)
        {
                sum += (i + 1) * pow(x, i);
        }
return sum;
};

std::vector<double> w{};
int main() {
        auto xexp = [](double i) { return i * exp(i); };
        // n=3
        std::vector<double> w{0.555555555555556, 0.888888888888889,
        0.555555555555556};
        std::vector<double> x{-0.774596669241483, 0, 0.774596669241483};
        std::cout << "n=" << x.size() << std::endl;
        std::cout << "\tf1 (should be 22.9973): "
                << gaussQuad(-2.2, 2.2, w, x, f1)
                << std::endl;
        std::cout << "\tf2 (should be 0): "
                << gaussQuad(0, 2 * M_PI, w, x, f2)
                << std::endl;
        std::cout << "\tpoly[8] (should be 8): "
                << gaussQuad(0, 1, w, x, poly)
                << std::endl;
        std::cout << "\tx*exp(x) (should be 1): "
                << gaussQuad(0, 1, w, x, xexp)
                << std::endl;
        // n=5;
        w = {0.236926885056189, 0.478628670499366, 0.568888888888889,
        0.478628670499366, 0.236926885056189};
        x = {-0.906179845938664, -0.538469310105683, 0, 0.538469310105683,
        0.906179845938664};
        std::cout << "n=" << x.size() << std::endl;
        std::cout << "\tf2 (should be 0): "
                << gaussQuad(0, 2 * M_PI, w, x, f2)
                << std::endl;
        std::cout << "\tpoly[8] (should be 8): "
                << gaussQuad(0, 1, w, x, poly)
                << std::endl;
        std::cout << "\tx*exp(x) (should be 1): "
                << gaussQuad(0, 1, w, x, xexp)
                << std::endl;
        return 0;
}
```

Output:

n=3
f1 (should be 22.9973): 22.9973
f2 (should be 0): -0.443228
poly[8] (should be 8): 7.96362
x*exp(x) (should be 1): 0.999995

n=5
f2 (should be 0): -0.000608033
poly[8] (should be 8): 8
x*exp(x) (should be 1): 1

Zadanie 3 – tworzenie macierzy na stosie

Uzupełnij definicję funkcji

```
std::vector< double* >* tworzMacierz(unsigned n)
```

która tworzy na stosie (z użyciem operatora new) obiekt typu

```
std::vector< double* >
```

a następnie tworzy na stosie n tablic typu double i wypełnia wektor wskaźnikami do tych tablic. Wszystkie tablice mają mieć n elementów i mają być wypełnione tak, żeby wektor reprezentował macierz diagonalną o liczbach od n do 1 na diagonali (w takiej kolejności, tzn. $M_{11} = n, M_{22} = n-1...$). Elementy niedagonalne mają być równe 0. Funkcja na koniec ma zwrócić wskaźnik do wektora. W szczególnym przypadku, gdy $n == 0$, funkcja tworzMacierz ma zwrócić wskaźnik pusty (nullptr).

Następnie uzupełnij definicję funkcji:

```
void usunMacierz(std::vector<double*>* M)
```

której zadaniem jest poprawne zwolnienie pamięci po wszystkich tablicach typu double, a na koniec po obiekcie $std :: vector < double* >$. Aby to zrobić należy najpierw zwolnić pamięć dla danej tablicy, a nastepnie usunąć wskaźnik do niej za pomocą std::vector::erase. Dopiero po usunięciu wszystkich tablic, trzeba usunąć pamięć po wektorze.

Wykorzystaj poniższy szablon:

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

using namespace std;

void wypiszMacierz(vector<double*>* M)
{
        if(M)
        {
                for(auto it = M->begin(); it != M->end(); it++)
                {
                        for(int ii=0; ii<M->size(); ii++)
                        {
                                cout << (*it)[ii] << " ";
                        }
                        cout << endl;
```

```
                }
        }
        else
                cout << "Pusta!" << endl;

}

vector<double*>* tworzMacierz(unsigned n)
{
        //...
}

void usunMacierz(vector<double*>* M)
{
        //...
}

int main()
{
        unsigned n = 5;
        cout << "Tworze macierz..." << endl;
        auto M = tworzMacierz(n);
        cout << "Oto ona:" << endl;
        wypiszMacierz(M);
        cout << "Usuwam macierz..." << endl;
        usunMacierz(M);
        cout << "Usunieto!" << endl;

        cout << "Tworze macierz..." << endl;
        auto N = tworzMacierz(0);
        cout << "Oto ona:" << endl;
        wypiszMacierz(N);
        cout << "Usuwam macierz..." << endl;
        usunMacierz(N);
}
```

Output: **Tworze macierz...**
Oto ona:
5 0 0 0 0
0 4 0 0 0
0 0 3 0 0
0 0 0 2 0
0 0 0 0 1
Usuwam macierz...
Usunięto!
Tworze macierz...
Oto ona:
Pusta!
Usuwam macierz...
Nie ma co usuwać!

Zadanie 4 – zabawa wskaźnikami

Napisz funkcję

```
void przestaw(char* pa, char*pb, const unsigned n)
```

która jako argumenty przyjmuje dwa wskaźniki na dowolne elementy tablicy znaków char oraz liczbę nieujemną n. Zadaniem funkcji jest zmiana tablicy tab w taki sposób, żeby wszystkie elementy pomiędzy wskaźnikami pa i pb (łącznie z nimi) zostały przesunięte cyklicznie o n. Jeżeli pa wskazuje na element stojący w tablicy przed elementem wskazywanym przez pb to przesunięcie powinno odbyć się w prawo. Jeżeli jest odwrotnie, to przesunięcie powinno być w lewo. Zatem zawsze przesuwamy od pa do pb.
Przykład:
Mamy tablicę ['a', 'b', 'c', 'd', 'e', 'f', 'g'], wskaźnik pa wskazuje na literę 'b' a wskaźnik pb na literę 'f', n wynosi 2. Skutkiem działania programu powinno być zmienienie tablicy tab tak, żeby otrzymać tablicę ['a', 'e', 'f', 'b', 'c', 'd', 'g'].
Inny przykład, ta sama tablica, pa wskazuje na 'f' a pb na 'b', przesunięcie znów wynosi 2, dostajemy ['a', 'd', 'e', 'f', 'b', 'c', 'g'].

Przykładowy kod:

```
#include <iostream> \\nie wolno spacji przed >
#include <cmath>

using namespace std;

void wypisz(char* wsk, size_t rozmiar)
{
        cout << "[";
        for(int ii=0; ii<rozmiar; ii++)
        {
                //arytmetyka wskaznikow
                cout << *(wsk+ii);
                if(ii < rozmiar-1)
                        cout << ", ";
        }
        cout << "]" << endl;
}

void przestaw(char* pa, char* pb, const unsigned n)
{
        // Uzupelnij
}

int main()
{
        char tab[12] = {'a', 'b', 'c', 'd', 'e', 'f', 'g',
                'h', 'i', 'j', 'k', 'l'};
        cout << "Tablica:" << endl;
        wypisz(tab, 12);
        cout << "Przesuwamy w prawo o 2 od tab[3] do tab[10] " << endl;
        przestaw(tab+3, &tab[11]-1, 2);
        wypisz(tab, 12);
```

```
        cout << "Przesuwamy w lewo o 2 od tab[10] do tab[3] " << endl;
        przestaw( &tab[11]-1, tab+3, 2);
        wypisz(tab, 12);
        cout << "Przesuwamy w lewo o 13 cala tablice " << endl;
        przestaw( &tab[11], tab, 13);
        wypisz(tab, 12);
        cout << "Przesuwamy w lewo o 0 cala tablice  " << endl;
        przestaw( &tab[11], tab, 0);
        wypisz(tab, 12);
        cout << "Przesuwamy w prawo o 1 od  tab[5] do tab[11]" << endl;
        przestaw( &tab[5], &tab[11], 1);
        wypisz(tab, 12);
        cout << "Proba uzycia nullptr" << endl;
        przestaw( &tab[5], nullptr, 1);
        wypisz(tab, 12);

        return 0;
}
```

Output:
Tablica:
[a, b, c, d, e, f, g, h, i, j, k, l]
Przesuwamy w prawo o 2 od tab[3]do tab[10]
[a, b, c, j, k, d, e, f, g, h, i, l]
Przesuwamy w lewo o 2 od tab[10]do tab [3]
[a, b, c, d, e, f, g, h, i, j, k, l]
Przesuwamy w lewo o 13 cala tablice
[b, c, d, e, f, g, h, i, j, k, l, a]
Przesuwamy w lewo o 0 cala tablice
[b, c, d, e, f, g, h, i, j, k, l, a]
Przesuwamy w prawo o 1 od tab[5]do tab[11]
[b, c, d, e, f, a, g, h, i, j, k, l]
Proba uzycia nullptr
Wskaznik jest pusty!
[b, c, d, e, f, a, g, h, i, j, k, l]

Uwaga: W przykładzie tablica jest posortowana, ale w żadnym razie nie wolno tego zakładać w programie. Nie wolno zakładać również wielkości tablicy. Wskaźniki pa i pb albo są puste albo wskazują na dwa elementy tablicy zaalokowanej jako jeden blok, zatem można i należy używać arytmetyki wskaźników. Jeżeli któryś ze wskaźników jest pusty, należy wyświetlić komunikat i zostawić tablicę niezmodyfikowaną. Nie wolno modyfikować sygnatury funkcji przestaw. Nie wolno w żaden sposób przekazywać wielkości tablicy do funkcji przestaw.
Podpowiedź: Odejmując dwa wskaźniki od siebie można uzyskać odległość elementów w tablicy. Może się przydać tworzenie tablicy na stosie (z użyciem operatora new), należy jednak bezwzględnie pamiętać o zwolnieniu pamięci!