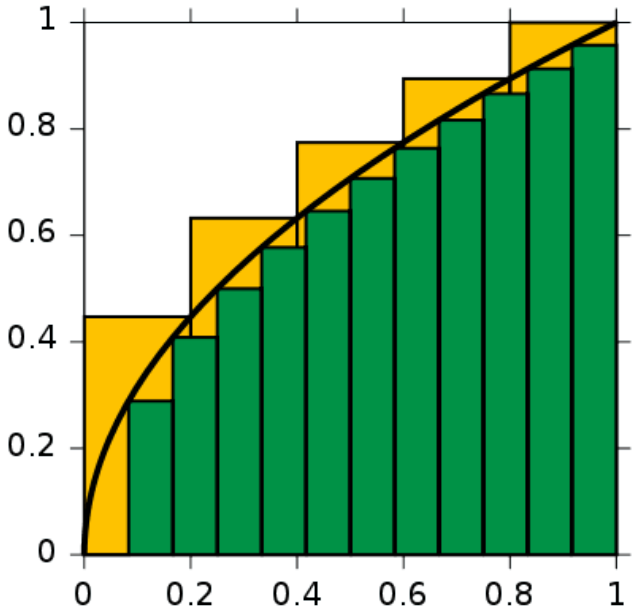
Zadanie 0 – całkowanie metodą prostokątów (1,2 pkt)

Najprostszym algorytmem całkującym jest metoda prostokątów, nawiązująca do formalnej definicji całki oznaczonej. Rozważmy funkcję jednego argumentu. W metodzie tej dzielimy przedział całkowania na n podprzedziałów. Następnie dla każdego podprzedziału wybieramy jeden punkt i liczymy dla niego wartość naszej funkcji. Mnożąc tą wartość przez szerokość przedziału uzyskujemy pole prostokąta pomiędzy osią OX a wykresem funkcji (patrz obrazek). Sumując pola prostokątów uzyskujemy przybliżoną wartość całki oznaczonej.



Rysunek 1: Źródło: Google Image

Ważne jest, żeby przedziały na które dzielimy przedział całkowania były jak najmniejsze. Jeżeli chcemy podzielić odcinek [0,1] na 4 przedziały, to złym podziełem będzie {(0,9), (0.9, 0.95), (0.95,0.99), (0.99, 1)}. Zazwyczaj po prostu dzieli się przedział całkowania na n przedziałów identycznej szerokości. Jeśli chodzi o wybór punktu dla którego liczymy wartość funkcji całkowanej to przy dostatecznie dużej ilości przedziałów nie ma dużego wpływu na dokładność (o ile funkcja nie jest bardzo szybko zmienna). W zadaniu proszę liczyć dla punktów na środkach przedziałów.
Dla waszej wygody przygotowałem szablon, w którym wystarczy uzupełnić funkcję całkującą. Chcemy, żeby całkowanie można przeprowadzić dla dowolnej funkcji przyjmującej jeden parametr typu double i zwracającej double. Program musi sobie radzić z sytuacją gdy do funkcji poda się pusty wskaźnik. Powinnien wtedy pojawić się stosowny komunikat a program powinien zakończyć pracę, np. wywołując **exit(1);**. Wynik powinien być poprawny dla przedziałów rosnących i malejących oraz dla pustego przedziału.

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

using namespace std;

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

```
// tutaj napisz funkcje do calkowania (u mnie nazywa sie calka)

int main()
{
        cout << "Calka z sin(x) od 0 do pi przy 10 podzialach: "
         << calka(0, M_PI, sin, 10) << endl;
        cout << "Calka z sin(x) od 0 do pi przy 100 podzialach: "
         << calka(0, M_PI, sin, 100) << endl;
        cout << "Calka z sin(x) od 0 do pi przy 1000 podzialow: "
         << calka(0, M_PI, sin, 1000) << endl;
        cout << "Calka z cos(x) od 0 do pi przy 1000000 podzialow: "
         << calka(0, M_PI, cos, 1000000) << endl;
        cout << "Calka z sin^2(x) od 0 do 2pi przy 1000 podzialow: "
        << calka(0, 2*M_PI, sin2, 1000) << endl;
        cout << "Calka z sin^2(x) od 2pi do 0 przy 1000 podzialow: "
        << calka(2*M_PI, 0, sin2, 1000) << endl;
        cout << "Calka z sin^2(x) od 0 do 0 przy 1000 podzialow: "
         << calka(0, 0, sin2, 1000) << endl;
        // cout << "Calka z sin^2(x) od 0 do pi przy 0 podzialow: "
         << calka(0, M_PI, sin2, 0) << endl;
        cout << "Proba uzycia nullptr: " <<
         calka(0, M_PI, nullptr, 10) << endl;

        return 0;
}
```

Output:
Calka z $\sin(x)$ od 0 do pi przy 10 podziałach: 2.00825
Calka z $\sin(x)$ od 0 do pi przy 100 podziałach: 2.00008
Calka z $\sin(x)$ od 0 do pi przy 1000 podziałów: 2
Calka z $\cos(x)$ od 0 do pi przy 1000000 podziałów: -2.92468e-16
Calka z $\sin^2(x)$ od 0 do 2pi przy 1000 podziałów: 3.14159
Calka z $\sin^2(x)$ od 2pi do 0 przy 1000 podziałów: -3.14159
Calka z $\sin^2(x)$ od 0 do 0 przy 1000 podziałów: 0
Próba użycia nullptr: Nie podano funkcji!
Uwaga: Nie tworzyć tablic ani kolekcji. Nie używać gotowych implementacji algorytmów całkujących. Nie przeciążać funkcji całkującej. W razie problemów z kompilacją spróbować użyć flagi -std=c++11. Uzyskanie outputu identycznego z podanym nie oznacza automatycznie, że program jest poprawny. Otrzymanie podobnego, ale nieidentycznego outputu nie musi oznaczać, że program jest błędny.
Podpowiedź: Trzeba użyć wskaźników na funkcje. Na podstawie przykładów domyśl się, jak wygląda sygnatura funkcji całka (jakie argumenty przyjmuje i co zwraca).

Zadanie 1 – Interpolacja liniowa (1,2 pkt)

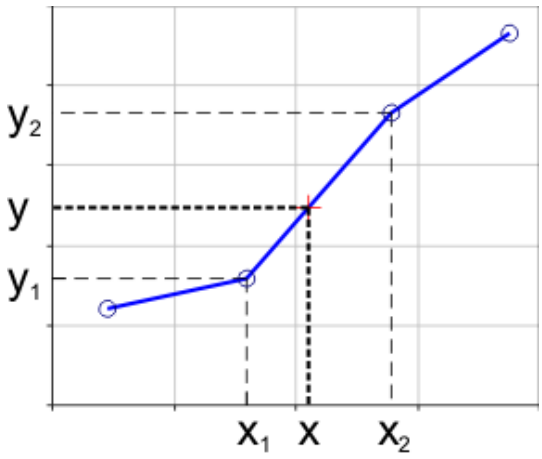
Napisz funkcję

```
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 (1,27 pkt)

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