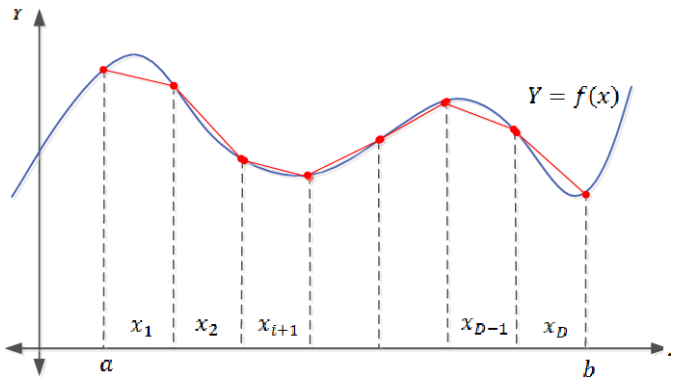
Zadanie 0 – całkowanie metodą trapezów (0,67 pkt)

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:

```
#define _USE_MATH_DEFINES
#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 0 z serii 7. 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 – metoda Simpsona (1pkt)

Napisz funkcję:

```
double simpsonIntegration(double(*f)(double), double from, double to, int n);
```

która liczy całkę oznaczoną z funkcji f na przedziale od *from* do *to*. Pamiętaj o sprawdzeniu poprawności danych. Jeżeli dane są złe, możesz np. wyświetlić komunikat o błędzie i zwrócić 0. Pamiętaj, że metoda Simpsona ma sens jedynie dla parzystych n. Nie zakładaj $from < to$.

Wykorzystaj poniższy szablon:

```
#include <cmath> //nie ma spacji przed >
#include <iostream>

double simpsonIntegration(double (*f)(double), double from, double to, int n);

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

int main()
{
        std::cout << simpsonIntegration(square, -1, 1, 6) << std::endl;
        std::cout << simpsonIntegration(nullptr, -1, 1, 6) << std::endl;
        std::cout << simpsonIntegration(xexp, 0, 1, 6) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, 6) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, 33) << std::endl;
        std::cout << simpsonIntegration(poly, 0, 1, -33) << std::endl;
        std::cout << simpsonIntegration(poly, 1, 0, 100) << std::endl;
        return 0;
}
```

Output:
0.666667
[ERROR] Nullptr!
0
1.00003
8.02462
[ERROR] Wrong value of n! 0
[ERROR] Wrong value of n! 0
-8

Uwaga: Nie tworzyć żadnych tablic ani kolekcji. Nie wykorzystywać żadnych gotowych algorytmów.
Podpowiedź: Zadanie jest proste, wystarczy zaimplementować odpowiedni wzór i sprawdzić, czy dane są poprawne.

Zadanie 2 – inne zastosowanie metody Simpsona (1pkt)

Skopiuj funkcję całkującą metodą Simpsona, którą napisałeś w poprzednim zadaniu.
Napisz funkcję:

```
double logarithm(double x, int n)
```

która wykorzysta napisaną w Zadaniu 1 funkcję simpsonIntegration(), żeby policzyć wartość logarytmu z liczby x przy n podziałach w metodzie Simpsona. Dla x mniejszego niż 0 zwróć *NAN*. *NAN* jest stałą używaną do określania liczb, które są niezdefiniowane albo nie można ich reprezentować. Zazwyczaj, jeżeli funkcja wykonująca działania numeryczne napotyka błędne argumenty, np. funkcja $1/x$ dla $x = 0$, to w zależności od intencji programisty robi jedną z kilku rzeczy:

1. Wyświetla komunikat o błędzie i przerywa działanie, najczęściej z wykorzystaniem wyjątków, które pozwalają wychwycić błąd i kontynuować pracę programu pomimo błędu.
2. Wyświetla komunikat o błędzie, ale nie przerywa działania, zwracając jakąś wartość domyślną (tak jak to zrobiliśmy w Zadaniu 1).
3. Nie wyświetla błędu ale zwraca NAN. Jeżeli wynik ma być wykorzystywany gdzieś dalej, to należy sprawdzić czy nie jest NAN za pomocą funkcji isnan().

Wykorzystaj poniższy przykład:

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

double simpsonIntegration(double (*f)(double), double from, double to, int n);

double logarithm(double x, int n);

int main()
{
        std::cout << "Ln(7)=" << logarithm(7, 16) << std::endl;
        std::cout << "Ln(e)=" << logarithm(2.718281828459, 10) << std::endl;
        std::cout << "Ln(0.5)=" << logarithm(0.5, 6) << std::endl;
        std::cout << "Ln(-1)=" << logarithm(-1, 6) << std::endl;
        std::cout << "Ln(0)=" << logarithm(0, 6) <<std::endl;
        auto val = std::isnan(logarithm(-2, 2)) ? "Yes!" : "No!";
        std::cout << "Is ln(-2) nan? " << val << std::endl;
        return 0;
}
```

Output:
Ln(7)=1.94642
Ln(e)=1.00003
Ln(0.5)=-0.69317
Ln(-1)=nan
Ln(0)=-inf
Is ln(-2) nan? Yes!

Uwaga: Nie tworzyć tablic ani kolekcji. Nie modyfikować funkcji całkującej.
Podpowiedź: Cała trudność polega na wymyśleniu jak policzyć logarytm wykorzystując całkowanie. Przyda się napisanie jeszcze jednej prostej funkcji (albo użycie funkcji lambda jeśli ktoś zna).

Zadanie 3 – zabawa wskaźnikami (1 pkt)

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!