Programowanie II

Lista 4

Podział aplikacji, wektory

Aplikacje warto dzielić na więcej plików

Plik nagłówkowy – w tym wypadku o nazwie naglowek.h. Zawiera określenia wszystkich zmiennych, definicji klas i funkcji, które mają być dostępne w zakresie całego programu (przykład przytoczony wprost z książki p. Gręboszа).

```
/*
Zmienne, które będą funkcjonować w przestrzeni wspólnej programu
będą dostępne w zakresach wszystkich plików, które będę zawierały odniesienie
do tego pliku nagłówkowego - w tym wypadku naglowek.h
*/
extern int ile_pigmejow;
extern int ile_europejczykow;

/*
Analogicznie jak wyżej - nagłówki metod które mają funkcjonować w przetrzeni
wspólnej programu
*/
void funkcja_etiopska();
void funkcja_francuska();
void funkcja_niemiecka();
```

plik europa.cpp – taka programowa biblioteka funkcji

```
#include "stdafx.h"
#include <iostream>;
#include "naglowek.h"
//odwołanie do pliku nagłówkowego
#include <string>
using namespace std;
int ile_europejczykow = 6;
/*
właściwa deklaracja zmiennej udsostępnianej później za pośrednictwem
naglowek.h
*/

void funkcja_niemiecka()
{
    cout << "Jestem w Europie (Berlin)\n";
    //korzystamy ze "swojej" zmiennej (ile_europejczykow), ale również z zadeklrowanej w innym pliku (ile_pigmejow)
    cout << "Mam przyjaciol  - " << ile_pigmejow << " Pigmejow, oraz " << ile_europejczykow << " Europejczykow\n";

}
void funkcja_francuska()
{
    cout << "Jestem w Europie (Paryz)\n";
    //analogicznie jak w przypadku poprzednims
    cout << "Mam przyjaciol  - " << ile_pigmejow << " Pigmejow, oraz " << ile_europejczykow << " Europejczykow\n";

```

główny plik aplikacji:

```cpp
#include "stdafx.h"
#include <iostream>
#include "naglowek.h"
//odwołanie do pliku nagłówkowego
using namespace std;

int ile_pigmejow = 10;
/*
właściwa deklaracja zmiennej udsostępnianej później za pośrednictwem
naglowek.h
*/
void funkcja_etiopska()
{
    cout << "Jestem w Afryce (Adis Abeba)\n";
    //korzystamy ze "swojej" zmiennej (ile_europejczykow), ale również z zadeklrowanej w innym pliku (ile_pigmejow)
    cout << "Mam przyjaciol  - " << ile_pigmejow << " Pigmejow, oraz " << ile_europejczykow << " Europejczykow\n";
}


int main()
{
    funkcja_etiopska();
    funkcja_francuska();
    funkcja_niemiecka();
    system("pause");
    return 0;
}
```

```
Jestem w Afryce (Adis Abeba)
Mam przyjaciol  - 10 Pigmejow, oraz 6 Europejczykow
Jestem w Europie (Paryz)
Mam przyjaciol  - 10 Pigmejow, oraz 6 Europejczykow
Jestem w Europie (Berlin)
Mam przyjaciol  - 10 Pigmejow, oraz 6 Europejczykow
Press any key to continue . . .
```

Przykład w którym funkcje zdefiniowane w rożnych plikach bibliotecznych (plik_1.cpp oraz plik_2.cpp) korzystają z siebie wzajemnie, a także się wywoływane z poziomu programu „głównego”.

naglowek.h

```cpp
#pragma once
#include "stdafx.h"
#include <iostream>
#include <string>

extern int wartosc_1;
extern int wartosc_2;

void funkcja_z_pliku_1(int param1=0);
void funkcja_z_pliku_2(int param1=0);
```

plik_1.cpp

```cpp
#include "stdafx.h"
#include <iostream>
#include "naglowek.h"
using namespace std;

int wartosc_1 = 10;

void funkcja_z_pliku_1(int param1)
{
    cout << "****************************************\n";
    wartosc_1++;
    if (param1 == 0)
    {
        cout << "Wywolanie funkcji z pliku 1 z poziomu main()\n";
    }
    else
    {
        cout << "Wywolanie funkcji z pliku 1 z poziomu innej funkcji \n" << param1;

    }
    cout << "wartosc 1: " << wartosc_1 << endl;
    cout << "wartosc 2: " << wartosc_2 << endl;
    cout << "****************************************\n";
    funkcja_z_pliku_2(1);
}
```

plik_2.cpp

```cpp
#include "stdafx.h"
#include <iostream>
#include "naglowek.h"
#include <string>

using namespace std;

int wartosc_2 = 10;
void funkcja_z_pliku_2(int param1)
{
    cout << "****************************************\n";
    wartosc_2++;
    if (param1 == 0)
    {
        cout << "Wywolanie funkcji z pliku 2 z poziomu main()\n";
    }
    else
    {
        cout << "Wywolanie funkcji z pliku 2 z poziomu innej funkcji\n";

    }
    cout << "wartosc 1: " << wartosc_1 << endl;
    cout << "wartosc 2: " << wartosc_2 << endl;
    cout << "****************************************\n";
}
```

plik „główny”

```cpp
/*
Plik głowny aplikacji
*/

#include "stdafx.h"
#include <iostream>
#include "naglowek.h"

int main()
{
    funkcja_z_pliku_1();
    funkcja_z_pliku_2();
    system("pause");
    return 0;
}
```

Klasa vector

```cpp
/*
Jeżeli potrzebujemy przechowywać większą ilosć zmiennych
to oczywiście naturalnym skojarzneiem jest wykorzystanie tablic.
Tablice są wydajne ale nie zawsze proste i inticyjne w obsłudze.
Dlatego wprowadzono pojęcie <vector> - czyli taka tablica tylko nieco "sprytniejsza"
Aby móc wykorzystać std::vector należy dodać do programu dyrektywę #include <vector>
*/
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<double> wartosc_pomiarow;
    vector<string> studenci;
    vector<int> elementy{ 4,7,3,9,10 };
    vector<string> mieszkancy{"Nowak","Kowalski","Tomaszewski" };
    //różne metody deklaracji wektorów, inicjaja wektorów
    for (int i = 0; i < elementy.size(); i++)
        cout << elementy[i] << ", ";
    cout << endl;
    for (int i = 0; i < mieszkancy.size(); i++)
        cout << mieszkancy[i] << ", ";
    cout << endl;
    //klasyczna pętla przechodząc po wszystkich elementach wektora (jak po tablicy)
    for (auto element : elementy)
        cout << element << ", ";
    cout << endl;
    for (string element : mieszkancy)
        cout << element << ", ";
    cout << endl;
    //tzw. pętla zakresowa
    system("pause");
    return 0;
}
```

```
4, 7, 3, 9, 10,
Nowak, Kowalski, Tomaszewski,
4, 7, 3, 9, 10,
Nowak, Kowalski, Tomaszewski,
Press any key to continue . . .
```

Metody klasy vector

```cpp
#include "pch.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void wyswietlWektor(string nazwa, vector<int> &wyswietlany)
{
    cout << "rozmiar wektora "+nazwa+ " to: "<< int(wyswietlany.size())<<", zawartosc: ";
    for (auto element : wyswietlany)
        cout << element << ", ";
    cout << endl;
};

void wyswietlWektorOdTylu(string nazwa, vector<int> &wyswietlany)
{
    cout << nazwa + " od konca to: ";
    vector<int>::reverse_iterator iterator;
    // tzw iterator odwrotny - pozwala poruszać się po wektorze w kolejności od konca do początku
    for (iterator= wyswietlany.rbegin(); iterator != wyswietlany.rend(); ++iterator)
    {
        cout << *iterator << ", ";
    }
    cout << endl;
};

int main()
{

    vector<int> wektor1;
    vector<int> wektor2;
    vector<int> wektor3;
    vector<int> wektor4;
```

```
36
37      int tablicaIntegerow[] = { 5,6,7,8 };
38      /*
39      metoda assign przypisuje nową zawartość do obiektu klasy vector zamienieniając jego aktualną zawartość
40      i modyfikuje jego rozmiar
41      */
42      wektor1.assign(7, 100);  //do wektor1 trafi 7 razy wartość 100
43      wyswietlWektor("wektor1", wektor1);
44      vector<int>::iterator iterator; //zmienna pomocnicza - wskaźnik na jednen element wektora
45      iterator = wektor1.begin() + 1;
46      /*
47      metoda begin() zwraca wskaźnik na pierwszy element wektora - powiększamy go o 1 ustawiając tym
48      samym położenie wskaźnika na 2 element wektora wektor1
49      */
50      wektor2.assign(iterator, wektor1.end() - 1);
51      /*
52      metoda end() zwraca wskaźnik do ostaniego elementu wektora. W efekcie do wektora2 zostaną
53      przekopiowane wartości z wektora pierwszego począwszy od pozycji 2 do przedostatniej
54      */
55      wyswietlWektor("wektor2", wektor2);
56
57      wektor3.assign(tablicaIntegerow, tablicaIntegerow+2);
58      /*
59      przepisanie 2 elementów z tablicy podanej jako drugi argument
60      */
61      wyswietlWektor("wektor3", wektor3);
62      int rozmiarTablicy = sizeof(tablicaIntegerow) / sizeof(tablicaIntegerow[0]);
63      wektor4.assign(tablicaIntegerow, tablicaIntegerow + rozmiarTablicy);
64      /*
65      przepisanie wszystkich elementów z tablicy podanej jako drugi argument
66      */
67      wyswietlWektor("wektor4", wektor4);
37      int tablicaIntegerow[] = { 5,6,7,8 };
38      /*
39      metoda assign przypisuje nową zawartość do obiektu klasy vector zamienieniając jego aktualną zawartość
40      i modyfikuje jego rozmiar
41      */
42      wektor1.assign(7, 100);  //do wektor1 trafi 7 razy wartość 100
43      wyswietlWektor("wektor1", wektor1);
44      vector<int>::iterator iterator; //zmienna pomocnicza - wskaźnik na jednen element wektora
45      iterator = wektor1.begin() + 1;
46      /*
47      metoda begin() zwraca wskaźnik na pierwszy element wektora - powiększamy go o 1 ustawiając tym
48      samym położenie wskaźnika na 2 element wektora wektor1
49      */
50      wektor2.assign(iterator, wektor1.end() - 1);
51      /*
52      metoda end() zwraca wskaźnik do ostaniego elementu wektora. W efekcie do wektora2 zostaną
53      przekopiowane wartości z wektora pierwszego począwszy od pozycji 2 do przedostatniej
54      */
55      wyswietlWektor("wektor2", wektor2);
56
57      wektor3.assign(tablicaIntegerow, tablicaIntegerow+2);
58      /*
59      przepisanie 2 elementów z tablicy podanej jako drugi argument
60      */
61      wyswietlWektor("wektor3", wektor3);
62      int rozmiarTablicy = sizeof(tablicaIntegerow) / sizeof(tablicaIntegerow[0]);
63      wektor4.assign(tablicaIntegerow, tablicaIntegerow + rozmiarTablicy);
64      /*
65      przepisanie wszystkich elementów z tablicy podanej jako drugi argument
66      */
67      wyswietlWektor("wektor4", wektor4);
```

```
68      cout << "Wartosc na pozycji 2 w wektor4 to: " << wektor4.at(1) << endl;
69      cout << "capasity() dla wektor4 to: " << wektor4.capacity() << endl;
70      //ilość pamięci zajmowana przez wektor
71      cout << "max_size () dla wektor4 to: " << wektor4.max_size() << endl;
72      //teoretyczny maksymalny rozmiar wektora
73      wektor1.clear(); //czyści wektor i uztswia jego rozmiar na 0
74      cout << "wektor1 po wykonaniu clear(), ";
75      wyswietlWektor("wektor1", wektor1);
76      wektor1.assign(wektor4.begin(), wektor4.end());
77      wyswietlWektor("wektor1 po przepisaniu z wektor4", wektor1);
78      wektor1.emplace(wektor1.begin() + 1, 100);
79      //emplace() pozwala dopisać nowy element we wskazanym miejscu
80      wyswietlWektor("wektor1 po dopisaniu wartosci 100 na drugiej pozycji", wektor1);
81      wektor1.emplace(wektor1.end(), 200);
82      wyswietlWektor("wektor1 po dopisaniu wartosci 200 na koncu", wektor1);
83      //  erase() usuwa wskazny element lub grupę elementów na pozycjach od ... do
84      wektor1.erase(wektor1.begin() + 4);
85      wyswietlWektor("wektor1 po usunieciu 5 piatej wartosci ", wektor1);
86      wektor1.erase(wektor1.begin(), wektor1.begin() + 2);
87      wyswietlWektor("wektor1 po usunieciu 2 pierwszych wartosci ", wektor1);
88      wektor1.push_back(250);
89      wyswietlWektor("wektor1 po dodaniu 250 na koncu ", wektor1);
90      wektor1.pop_back();
91      wyswietlWektor("wektor1 po usunieciu 250 na koncu ", wektor1);
92      wyswietlWektorOdTylu("wektor1 od tylu ", wektor1);
93
94      system("pause");
95  }
```

```
rozmiar wektora wektor1 to: 7, zawartosc: 100, 100, 100, 100, 100, 100, 100,
rozmiar wektora wektor2 to: 5, zawartosc: 100, 100, 100, 100, 100,
rozmiar wektora wektor3 to: 2, zawartosc: 5, 6,
rozmiar wektora wektor4 to: 4, zawartosc: 5, 6, 7, 8,
Wartosc na pozycji 2 w wektor4 to: 6
capasity() dla wektor4 to: 4
max_size () dla wektor4 to: 1073741823
wektor1 po wykonaniu clear(), rozmiar wektora wektor1 to: 0, zawartosc:
rozmiar wektora wektor1 po przepisaniu z wektor4 to: 4, zawartosc: 5, 6, 7, 8,
rozmiar wektora wektor1 po dopisaniu wartosci 100 na drugiej pozycji to: 5, zawartosc: 5, 100, 6, 7, 8,
rozmiar wektora wektor1 po dopisaniu wartosci 200 na koncu to: 6, zawartosc: 5, 100, 6, 7, 8, 200,
rozmiar wektora wektor1 po usunieciu 5 piatej wartosci  to: 5, zawartosc: 5, 100, 6, 7, 200,
rozmiar wektora wektor1 po usunieciu 2 pierwszych wartosci  to: 3, zawartosc: 6, 7, 200,
rozmiar wektora wektor1 po dodaniu 250 na koncu  to: 4, zawartosc: 6, 7, 200, 250,
rozmiar wektora wektor1 po usunieciu 250 na koncu  to: 3, zawartosc: 6, 7, 200,
wektor1 od tylu  od konca to: 200, 7, 6,
Press any key to continue . . .
```

Wektory wielowymiarowe

```cpp
/*
Tablice w C++ (jak i w C) mają zasadniczą wadę - ich rozmiar musi być
znay w momencie kompilacji - a jeśli tego nie wiadomo - można wykorzystać
wektory, które w przypadku więcej niż jednego wymiaru również funkcjonują
na analogicznej zasadzie jak tablice wielowymiarowe i są po prostu wektorami wektorów
*/

#include "pch.h"
#include <iostream>
#include <vector>
using namespace std;

//
vector<vector<char >> wektor2d_wypel{
                                        {'a','b','c'},
                                        {'d','e','f'}
};
vector<vector<char >> wektor2d_pusty{4,vector<char>(5)};

//wektor wektorów
void WyswietlWektor2D(vector<vector<char >> wyswietlany)
{
    int IloscWierszy = wyswietlany.size();
    //badamy "zewnętrzny" wymiar aby ustalić liczbę wierszy
    for (int i = 0; i < IloscWierszy; i++)
    {
        int IloscKolumn = wyswietlany[i].size();
        //badamy wymiar każdego z wierszy (nie zawsze kazdy musi mieć taki sam
        for (int j = 0; j < IloscKolumn; j++)
            cout <<"|"<<wyswietlany[i][j];
        cout << "|"<<endl;
    }

}
```

```
35  void WyswietlWektor2D_ver2(vector<vector<char >> wyswietlany)
36  //nie trzeba badać rozmiarów wektora - można wykorzystać pętlę zakresową
37  {
38
39      for (auto wiersz:wyswietlany)
40      {
41          for (auto pozycja:wiersz)
42              cout << "|" << pozycja;
43          cout << "|" << endl;
44      }
45
46  }
47
48  void WypelnijWektor2D(vector<vector<char >> & wypelniany)
49  {
50      //przekazujemy wektor do funkcji przez referencję
51      char znak = '!';
52      int IloscWierszy = wypelniany.size();
53      //badamy "zewnętrzny" wymiar aby ustalić liczbę wierszy
54      for (int i = 0; i < IloscWierszy; i++)
55      {
56          int IloscKolumn = wypelniany[i].size();
57          for (int j = 0; j < IloscKolumn; j++)
58              wypelniany[i][j] = znak++;
59      }
60
61  }
```

```
62
63  int main()
64  {
65      WyswietlWektor2D(wektor2d_wypel);
66      WyswietlWektor2D_ver2(wektor2d_wypel);
67      WypelnijWektor2D(wektor2d_pusty);
68      WyswietlWektor2D_ver2(wektor2d_pusty);
69      system("pause");
70  }
71
```

```
|a|b|c|
|d|e|f|
|a|b|c|
|d|e|f|
|!|"|#|$|%|
|&|'|(|)|*|
|+|,|-|.|/|
|0|1|2|3|4|
```

Uwaga !!!
Podczas realizacji zadań korzystamy z rozwiązań wykorzystywanych w przykładach

Zadanie 1 (4 pkt.)

Napisz program, który wczyta do jednowymiarowego wektora tyle różnych liczb całkowitych od użytkownika ile będzie on chciał, a następnie:

- znajduje największą i najmniejszą z nich, a także ich pozycje w zbiorze
- wyznaczy średnią wartość w tablicy
- zwróci pozycję wartości podanej przez użytkownika

Oczywiście powyższe funkcjonalności realizowane powinny być przez odpowiedni funkcje zgromadzone w stosownej bibliotece.

Zadanie 2 (10 pkt.)

Napisz program umożliwiający realizację następujących operacji na macierzach o dowolnych wymiarach (macierz w postaci wektora dwuwymiarowego):

- dodawanie macierzy
- odejmowanie macierzy
- mnożenie macierzy

Zawartość oraz wymiary macierzy wczytywane mają być od użytkownika. Oczywiście operacje wykonujemy tylko na macierzach o właściwych relacjach wymiarów . Program kończy swoje działanie dopiero po jednoznacznym wskazaniu tego przez użytkownika – pozwala wielokrotnie wykonywać operacje. Oczywiście powyższe funkcjonalności realizowane powinny być przez odpowiedni funkcje zgromadzone w stosownych bibliotekach.