Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

6. Napisy i macierze

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

6.1. Własne biblioteki funkcji

Jako że program czytany jest (przez nas jak i przez kompilator) od góry do dołu, *definicja każdej funkcji musi się pojawić przed jej pierwszym użyciem*. W niektórych przypadkach może się jednak zdarzyć, że lepiej będzie (np. dla czytelności) umieścić definicję funkcji w jakimś innym miejscu. Można to zrobić pod warunkiem, że obiektom z niej korzystającym zostanie udostępniona jej *deklaracja*.

Intuicyjnie, na etapie pisania programów innych obiektom wystarcza informacja, że dana funkcja istnieje, nazywa się tak i tak, przyjmuje dane wartości i zwraca coś określonego typu. Dokładna specyfikacja jej działania nie jest im de facto potrzebna.

Informacja

Deklaracji funkcji dokonujemy w miejscu, w którym nie została ona jeszcze użyta (ale poza inną funkcją). Podajemy wtedy jej *deklarator zakończony średnikiem*. Co ważne, w deklaracji musimy użyć takiej samej nazwy funkcji oraz typów argumentów wejściowych i wyjściowych jak w definicji. Nazwy (identyfikatory) tych argumentów mogą być jednak inne, a nawet mogą zostać pominięte.

Definicja funkcji może wówczas pojawić się w dowolnym innym miejscu programu (także w innym pliku źródłowym, zob. dalej).

Przykład:

```
1  #include <iostream>
2  using namespace std;
3
4  double kwadrat(double); // deklaracja - tu jest średnik!
5
6  int main()
7  {
8     double x = kwadrat(2.0);
9     cout << x                          << endl;
10    cout << kwadrat(x)+kwadrat(8.0)  << endl;
11    cout << kwadrat(kwadrat(0.5))    << endl;
12    return 0;
13 }
14
15 double kwadrat(double x) // definicja - tu nie ma średnika!
16 {
17    return x*x;
18 }
```

■

Zapamiętaj

Kolejny raz mamy okazję obserwować, jak ważne jest, by nadawać obiektom (zmiennym, funkcjom) intuicyjne, samoopisujące się identyfikatory!

Zbiór różnych funkcji, które warto mieć „pod ręką”, możemy umieścić w osobnych plikach, tworząc tzw. *bibliotekę* funkcji. Dzięki temu program stanie się bardziej czytelny,

łatwiej będziemy mogli zapanować nad jego złożonością, a ponadto otrzymamy sposobność wykorzystania pewnego kodu ponownie w przyszłości. Raz napisany i przetestowany zestaw funkcji może oszczędzić sporo pracy.

Aby stworzyć bibliotekę funkcji, tworzymy najczęściej dwa dodatkowe pliki.

– plik nagłówkowy (ang. *header file*), `nazwabiblioteki.h` — zawierający tylko deklaracje funkcji,
– plik źródłowy (ang. *source file*), `nazwabiblioteki.cpp` — zawierający definicje zadeklarowanych funkcji.

Plik nagłówkowy należy dołączyć do wszystkich plików, które korzystają z funkcji z danej biblioteki (również do pliku źródłowego biblioteki). Dokonujemy tego za pomocą dyrektywy:

```
#include "nazwabiblioteki.h"
```

Zwróćmy uwagę, że nazwa naszej biblioteki, znajdującej się wraz z innymi plikami tworzonego projektu, ujęta jest w cudzysłowy, a nie w nawiasy trójkątne (które służą do ładowania bibliotek systemowych).

Informacja

Najlepiej, gdy wszystkie pliki źródłowe i nagłówkowe będą znajdować się w tym samym katalogu.

Przykład 1. Dla ilustracji przyjrzymy się, jak wyglądałby program korzystający z biblioteki zawierającej dwie następujące funkcje. Niech

$$\mathsf{min} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z},$$

$$\mathsf{max} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z},$$

będą funkcjami takimi, że

$$\mathsf{min}(n, m) := \begin{cases} n & \text{dla } n \leqslant m, \\ m & \text{dla } n > m, \end{cases}$$

oraz

$$\mathsf{max}(n, m) := \begin{cases} n & \text{dla } n \geqslant m, \\ m & \text{dla } n < m. \end{cases}$$

Nasz program będzie składał się z trzech plików: nagłówkowego `podreczna.h` oraz dwóch źródłowych `podreczna.cpp` i `program.cpp`.

1. Oto zawartość pliku `podreczna.h` — zawierającego tylko deklaracje funkcji pomocniczych.

```
1 #pragma once  // na początku każdego pliku nagłówkowego!
2
3 int min(int i, int j);
4 int max(int i, int j);
```

Zapamiętaj

Dyrektywa

```
#pragma once
```

powinna znaleźć się na początku każdego pliku nagłówkowego. Dzięki niej zapobiegniemy wielokrotnemu dołączaniu tego pliku w przypadku korzystania z niego przez inne biblioteki.

Ciekawostka

Dyrektywa `#pragma once` dostępna jest tylko w kompilatorze *Visual C++*. W innych kompilatorach uzyskanie tego samego efektu jest nieco bardziej złożone:

```
#ifndef __PODRECZNA_H
#define __PODRECZNA_H

// deklaracje....

#endif
```

2. Definicje funkcji z naszej biblioteki znajdą się w pliku `podreczna.cpp`.

```
1  #include "podreczna.h" /* dołącz plik z deklaracjami */
2
3  int min(int x, int y)  /* nazwy parametrów nie muszą być takie same
       jak w deklaracji */
4  {
5     if (x <= y) return x;
6     else        return y;
7  }
8
9  int max(int w, int z)
10 {
11    if (w >= z)
12       return w;
13    return z;
14 }
```

3. Gdy biblioteka jest gotowa, można przystąpić do napisania funkcji głównej. Oto zawartość pliku `program.cpp`.

```
1  #include <iostream>     /* dołącz bibliotekę systemową */
2  using namespace std;
3
4  #include "podreczna.h" /* dołącz własną bibliotekę */
5
6  int main(void)
7  {
8     int x, y;
```

```
9
10      cout << "Podaj dwie liczby. ";
11      cin  >> x >> y; // wczytaj z klawiatury
12
13      cout << "Max=" << max(x,y) << endl;
14      cout << "Min=" << min(x,y) << endl;
15
16      return 0;
17  }
```

■

6.2. Napisy (ciągi znaków)

Do tej pory nie zastanawialiśmy się, w jaki sposób można reprezentować w naszych programach napisy. Często są one nam potrzebne, np. gdy chcemy zakomunikować coś ważnego użytkownikowi, czy też przechować bądź przetworzyć informacje o nieliczbowym charakterze.

6.2.1. Kod ASCII

Pojedyncze znaki drukowane przechowywane są najczęściej w postaci obiektów typu `char` (ang. *character*). Oczywiście pamiętamy, że jest to po prostu typ, służący do przechowywania 8 bitowych *liczb* (sic!) całkowitych.

Istnieje ogólnie przyjęta umowa (standard), że liczbom z zakresu 0–127 *odpowiadają* ściśle określone znaki, tzw. kod ASCII (ang. *American Standard Code for Information Interchange*). Zestawiają je tablice 6.1–6.3.

Tab. 6.1. Kody ASCII cz. I — znaki kontrolne

DEC	HEX	Znaczenie	DEC	HEX	Znak
0	00	Null (\0)	16	10	Data Link Escape
1	01	Start Of Heading	17	11	Device Control 1
2	02	Start of Text	18	12	Device Control 2
3	03	End of Text	19	13	Device Control 3
4	04	End of Transmission	20	14	Device Control 4
5	05	Enquiry	21	15	Negative Acknowledge
6	06	Acknowledge	22	16	Synchronous Idle
7	07	Bell (\a)	23	17	End of Transmission Block
8	08	Backspace (\b)	24	18	Cancel
9	09	Horizontal Tab (\t)	25	19	End of Medium
10	0A	Line Feed (\n)	26	1A	Substitute
11	0B	Vertical Tab	27	1B	Escape
12	0C	Form Feed	28	1C	File Separator
13	0D	Carriage Return (\r)	29	1D	Group Separator
14	0E	Shift Out	30	1E	Record Separator
15	0F	Shift In	31	1F	Unit Separator

Tab. 6.2. Kody ASCII cz. II

DEC	HEX	Znaczenie	DEC	HEX	Znak	DEC	HEX	Znaczenie
32	20	Spacja	48	30	0	64	40	@
33	21	!	49	31	1	65	41	A
34	22	”	50	32	2	66	42	B
35	23	#	51	33	3	67	43	C
36	24	$	52	34	4	68	44	D
37	25	%	53	35	5	69	45	E
38	26	&	54	36	6	70	46	F
39	27	’	55	37	7	71	47	G
40	28	(	56	38	8	72	48	H
41	29	)	57	39	9	73	49	I
42	2A	*	58	3A	:	74	4A	J
43	2B	+	59	3B	;	75	4B	K
44	2C	,	60	3C	<	76	4C	L
45	2D	-	61	3D	=	77	4D	M
46	2E	.	62	3E	>	78	4E	N
47	2F	/	63	3F	?	79	4F	O

Tab. 6.3. Kody ASCII cz. III

DEC	HEX	Znak	DEC	HEX	Znaczenie	DEC	HEX	Znak
80	50	P	96	60	`	112	70	p
81	51	Q	97	61	a	113	71	q
82	52	R	98	62	b	114	72	r
83	53	S	99	63	c	115	73	s
84	54	T	100	64	d	116	74	t
85	55	U	101	65	e	117	75	u
86	56	V	102	66	f	118	76	v
87	57	W	103	67	g	119	77	w
88	58	X	104	68	h	120	78	x
89	59	Y	105	69	i	121	79	y
90	5A	Z	106	6A	j	122	7A	z
91	5B	[	107	6B	k	123	7B	{
92	5C	\	108	6C	l	124	7C	|
93	5D	]	109	6D	m	125	7D	}
94	5E	^	110	6E	n	126	7E	∼
95	5F	_	111	6F	o	127	7F	Delete

Na szczęście w języku C++ nie musimy pamiętać, która liczba odpowiada jakiemu symbolowi. Aby uzyskać wartość liczbową symbolu drukowanego, należy ująć go w *apostrofy*.

```
char c1 = 'A';
char c2 = '\n'; // znak nowej linii

cout << c1 << c2; //  "A" i przejście do nowej linii
cout << (int)c1 << (char)59 << (int)c2; //  "65;13"
```

Jak widzimy, domyślnie wypisanie na ekran zmiennej typu `char` jest równoważne z wydrukowaniem symbolu. Można to zachowanie zmienić, rzutując ją na typ `int`.

Ponadto przyglądając się uważniej tablicy kodów ASCII, warto zanotować następujące prawidłowości.

- Mamy następujący porządek leksykograficzny:
  `'A'` $<$ `'B'` $<$ ... $<$ `'Z'` $<$ `'a'` $<$ `'b'` $<$ ... $<$ `'z'`.
- Symbol cyfry `c` $\in \{0, 1, \ldots, 9\}$ można uzyskać za pomocą wyrażenia `'0'+c`.
- Kod ASCII `n`-tej wielkiej litery alfabetu łacińskiego to `'A'+n-1`.
- Kod ASCII `n`-tej małej litery alfabetu łacińskiego to `'a'+n-1`.
- Zamiana litery `l` na małą literę następuje za pomocą operacji `l+32`.
- Zamiana litery `l` na wielką literę następuje za pomocą operacji `l-32`.

Pozostałe symbole odpowiadające wartościom liczbowym (0x80–0xFF) nie są określone przez standard ASCII. Zdefiniowane są one przez inne kodowania, np. CP-1250 (Windows) bądź ISO-8859-2 (Internet, Linux) zawierają polskie znaki diakrytyczne. Jak widzimy, sprawa polskich „ogonków" jest nieco skomplikowana. Zatem `na poczatkowym etapie programowania uzywajmy tylko liter alfabetu lacinskiego w programach, ktore przetwarzaja napisy`.

Ciekawostka

Istnieją jeszcze inne standardy kodowania, zwane UNICODE (UTF-8, UTF-16, ...), w których jednemu znakowi niekoniecznie musi odpowiadać jeden bajt. Obsługa ich jednak jest nieco skomplikowana, zatem nie będziemy się nimi zajmować podczas tego kursu.

Informacja

Biblioteka `<cctype>` zawiera kilka funkcji przydatnych do sprawdzania, czy np. dany znak jest literą alfabetu łacińskiego, cyfrą „białym znakiem" (spacją, tabulacją itp.) itd. Zob. więcej: `http://www.cplusplus.com/reference/clibrary/cctype/`.

6.2.2. Reprezentacja napisów (ciągów znaków)

Wiemy już, w jaki sposób reprezentować pojedyncze znaki.

Zapamiętaj

Najprostszym sposobem reprezentowania ciągów symboli drukowanych, czyli *napisów*, są tablice elementów typu `char` *zakończone* umownie bajtem (liczbą całkowitą) o wartości *zero* (znak `'\0'`).

Napisy można utworzyć używając cudzysłowu (`"..."`). Takie napisy jednak są tablicami tylko do odczytu. Nie wiadomo bowiem, w jakim miejscu w pamięci zostaną one umieszczone i czy przypadkiem nie są one wykorzystywane w wielu miejscach tego samego programu.

```
char* napis1 = "Pewien napis."; // 13 znaków + bajt 0
// *prawie* równoważnie:
char napis2[14] = // tablica o ustalonym rozmiarze
   { 'P', 'e', 'w', 'i', 'e', 'n', ' ',
     'n', 'a', 'p', 'i', 's', '.', '\0' };

// Znaki w zmiennej napis1 są tylko do odczytu!
napis1[1] = 'k'; // nie wiadomo, co się stanie
napis2[1] = 'k'; // ok
```

6.2.3. Operacje na napisach

Jako że napisy są zwykłymi tablicami liczb całkowitych, implementacja podstawowych operacji na nich jest dość prosta. W niniejszym paragrafie rozważymy kilka z nich, resztę pozostawiając jako ćwiczenie.

Najpierw przyjrzymy się wypisywaniu.

```
char* napis = "Jakiś napis."; // tablica znaków zakończona zerem

// Zatem:
cout << napis;

// Jest równoważne:
int i=0;
while (napis[i] != '\0') // dopóki nie koniec napisu
{
   cout << napis[i];
   ++i;
}
```

Zapamiętaj

Jeśli mamy zaalokowaną tablicę typu char o identyfikatorze `napis` mogącą przechowywać maksymalnie $n+1$ znaków (dodatkowe miejsce na bajt zerowy), to wywołanie

```
cin >> napis;
```

wczytuje tylko jeden wyraz (uwaga, by nie wprowadzić więcej niż n znaków!). Jeśli chcemy wczytać cały wiersz, musimy wywołać:

```
cin.getline(napis, n);
```

Długość napisu można sprawdzić w następujący sposób.

```
int dlug(char* napis)
{
   // Zwraca długość napisu (bez bajtu zerowego). Zgodnie
   // z umową, mimo że jest to tablica, potrafimy jednak
   // sprawdzić, gdzie się ona kończy

   int i=0;
   while (napis[i] != 0)
      ++i;

   return i;
}
```

Ostatnim przykładem będzie tworzenie dynamicznie alokowanej kopii napisu.

```
char* kopia(char* napis)
{
   int n = dlug(napis);
   char* nowy = new char[n+1]; // o jeden bajt więcej!

   // nie zapominamy o skopiowaniu bajtu zerowego!
   for (int i=0; i<n+1; ++i)
      nowy[i] = napis[i];

   return nowy; // dalej nie zapomnijmy o dealokacji pamięci
}
```

6.2.4. Biblioteka <cstring>

Biblioteka **<cstring>** definiuje wiele funkcji służących do przetwarzania ciągów znaków[1]. Wybrane narzędzia zestawia tab. 6.4.

Zadanie

Spróbuj napisać własną implementację ww. funkcji samodzielnie.

[1] Zobacz angielskojęzyczną dokumentację dostępną pod adresem **http://www.cplusplus.com/reference/clibrary/cstring/**.

Tab. 6.4. Wybrane funkcje biblioteki `<cstring>`

Deklaracja	Opis
`int strlen(char* nap);`	Zwraca długość napisu.
`char* strcpy(char* cel, char* zrodlo);`	Kopiuje napisy. Uwaga: długość tablicy `cel` nie może być mniejsza niż długość napisu `zrodlo`+1.
`char* strcat(char* cel, char* zrodlo);`	Łączy napisy `cel` i `zrodlo`.
`int strcmp(char* nap1, char* nap2);`	Porównuje napisy. Zwraca 0, jeśli są identyczne. Zwraca wartość dodatnią, jeśli `nap1` jest większy (w porządku leksykograficznym) niż `nap2`.
`char* strchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia danego znaku.
`char* strrchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od ostatniego wystąpienia danego znaku.
`char* strstr(char* nap1, char* nap2);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia podnapisu `nap2` w napisie `nap1`.

6.3. Reprezentacja macierzy

Dana jest macierz określona nad pewnym zbiorem

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix}$$

gdzie n — liczba wierszy, m — liczba kolumn.

Macierze mogą być wygodnie reprezentowane w języku C++ na dwa sposoby:

– jako tablice tablic,

– jako tablice jednowymiarowe.

My na zajęciach wybieramy sposób pierwszy – zapewnia on bardzo wygodny dostęp do poszczególnych elementów, jednak cechuje się nieco zawiłym sposobem tworzenia i usuwania interesujących nas obiektów:

```
int n = ...; // liczba wierszy
int m = ...; // liczba kolumn
typ** A; // tablica o elementach typu "typ*", czyli tablica tablic

A = new typ*[n];
for (int i=0; i<n; ++i)
    A[i] = new typ[m];

// A to n-elementowa tablica tablic m-elementowych

// teraz np. A[0][3] to element w I wierszu i IV kolumnie....

```

```
13 for (int i=0; i<n; ++i)
14    delete [] A[i];
15 delete [] A;
```

Z drugiej strony, macierze możemy reprezentować za pomocą jednowymiarowych tablic. Łatwo się je tworzy, jednak odwoływanie się do elementów jest nieco zawiłe.

```
1 int n = ...; // liczba wierszy
2 int m = ...; // liczba kolumn
3 typ* A; // jednowymiarowa tablica
4
5 A = new typ[n*m];
6
7 // teraz np. A[1+3*n] to element w II wierszu i IV kolumnie....
8
9 delete [] A;
```

Często (i tak jest w powyższym przypadku) zakłada się, że elementy macierzy przechowywane są kolumnowo (najpierw cała I kolumna, potem II itd.).

6.4. Przykładowe algorytmy z wykorzystaniem macierzy

Omówmy kilka przykładowych algorytmów:

1. mnożenie dowolnego wiersza przez stałą,
2. odejmowanie pewnego wiersza przez inny pomnożony przez stałą,
3. zamiana dwóch kolumn,
4. wyszukanie wiersza, który ma największy co do modułu element w danej kolumnie,
5. dodawanie macierzy,
6. mnożenie macierzy,
7. rozwiązywanie układów równań liniowych metodą eliminacji Gaussa.

Będziemy rozpatrywać macierze o wartościach rzeczywistych (reprezentowanych jako tablice tablic o elementach typu `double`). Zauważ, że żadna z funkcji (o ile nie zaznaczono inaczej) nie „psuje” macierzy wejściowej, w większości przypadków zwracana jest nowa, dynamicznie alokowana macierz (oczywiście wtedy, gdy tego się spodziewamy). Takie postępowanie zależy od naszych potrzeb i specyfiki implementowanego programu, czasem być może prościej i wydajniej będzie z niego po prostu zrezygnować.

Zapamiętaj

Powyższe algorytmy będziemy implementować w postaci funkcji. Za każdym razem jako argumenty przekazujemy:

1. macierz (wskaźnik),
2. rozmiar macierzy,
3. parametry wykonywanej operacji.

Przydadzą nam się funkcje, które wyręczą nas w tworzeniu, kasowaniu i (w celach testowych) wypisywaniu na ekran macierzy.

```
double** tworzmacierz(int n, int m)
{
   assert(n > 0 && m > 0);
   double** W = new double*[n]; // tablica tablic
   for (int i=0; i<n; ++i)
      W[i] = new double[m];
   return W;
}
```

```
void kasujmacierz(double** W)
{
   assert(n > 0);
   for (int i=0; i<n; ++i)
      delete [] W[i];
   delete [] W;
}
```

```
void wypiszmacierz(double** W, int n, int m)
{
   assert(n > 0 && m > 0);

   // nie będzie zbyt pięknie, ale...
   for (int i=0; i<n; ++i) // dla każdego wiersza
   {
      for (int j=0; j<m; ++j) // dla każdej kolumny
         cout << '\t' << W[i][j];
      cout << endl;
   }
}
```

6.4.1. Mnożenie wiersza macierzy przez stałą

Załóżmy, że dana jest macierz rzeczywista A typu $n \times m$. Chcemy pomnożyć k-ty wiersz przez stałą rzeczywistą c. Oto przykładowa funkcja, która realizuje tę operację.

```
double** mnozwiersz(double** A, int n, int m, int k, double c)
{
   assert(k >= 0 && k < n && m > 0);
   double** B = tworzmacierz(n, m);

   for (int j=0; j<m; ++j)
   {
      for (int i=0; i<n; ++i)
         if (i == k)
            B[i][j] = A[i][j]*c;
         else
            B[i][j] = A[i][j]; // tylko przepisanie
   }

   return B;
}
```

6.4.2. Odejmowanie wiersza macierzy od innego pomnożonego przez stałą

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy odjąć od k-tego wiersza wiersz l-ty pomnożony przez stałą rzeczywistą c. Oto stosowny kod.

```
double** odskalwiersz(double** A, int n, int m, int k, int l,
    double c)
{
    assert(k >= 0 && k < n && l >= 0 && l < n && m > 0);

    double** B = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
    {
       for (int i=0; i<n; ++i)
          if (i == k)
             B[i][j] = A[i][j] - c*A[l][j];
          else
             B[i][j] = A[i][j];
    }

    return B;
}
```

6.4.3. Zamiana dwóch kolumn

Dana jest macierz A typu $n \times m$. Załóżmy, że chcemy zamienić k-tą kolumnę z l-tą.

```
double** zamienkol(double** A, int n, int m, int k, int l)
{
    assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);

    double** B = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
    {
       for (int i=0; i<n; ++i)
       {
          if (j == k)
             B[i][j] = A[i][l];
          else if (j == l)
             B[i][j] = A[i][k];
          else
             B[i][j] = A[i][j];
       }
    }

    return B;
}
```

Ciekawostka — wersja alternatywna (modyfikująca macierz wejściową):

```
void zamienkol(double** A, int n, int m, int k, int l)
{
    assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);

    for (int i=0; i<n; ++i)
    {
```

```
        double t = A[i][k];
        A[i][k] = A[i][l];
        A[i][l] = t;
    }
}
```

6.4.4. Wiersz z największym co do modułu elementem w kolumnie

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy znaleźć indeks wiersza, który ma największy co do modułu element w kolumnie k.

```
int wyszukajnajwkol(double** A, int n, int m, int k)
{
    assert(k >= 0 && k < m);

    int max = 0;
    for (int i=1; i<n; ++i)
    {
        if (fabs(A[i][k]) >= fabs(A[max][k]))
            max = i;
    }

    return max;
}
```

Przypomnijmy, że funkcja `double fabs(double x)` z biblioteki `<cmath>` zwraca wartość bezwzględną swego argumentu.

6.4.5. Dodawanie macierzy

Dane są dwie macierze A, B typu $n \times m$. Wynikiem operacji dodawania $A+B$ jest macierz:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,m-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,m-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}+b_{0,0} & a_{0,1}+b_{0,1} & \cdots & a_{0,m-1}+b_{0,m-1} \\ a_{1,0}+b_{1,0} & a_{1,1}+b_{1,1} & \cdots & a_{1,m-1}+b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0}+b_{n-1,0} & a_{n-1,1}+b_{n-1,1} & \cdots & a_{n-1,m-1}+b_{n-1,m-1} \end{bmatrix}.$$

Oto kod funkcji służącej do dodawania macierzy.

```
double** dodaj(double** A, double** B, int n, int m)
{
    assert(n > 0 && m > 0);
    double** C = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
        for (int i=0; i<n; ++i)
            C[i][j] = A[i][j] + B[i][j];

    return C; /* nie zapomnij o zwolnieniu pamięci dalej! */
}
```

Ciekawostka

Zamiana kolejności pętli wewnętrznej i zewnętrznej

```
for (int i=0; i<n; ++i)
   for (int j=0; j<m; ++j)
      C[i][j] = A[i][j] + B[i][j];
```

w przypadku macierzy dużych rozmiarów powoduje, że program może wykonywać się wolniej. Na komputerze skromnego autora niniejszego skryptu dla macierzy 10000×10000 czas wykonywania rośnie z 2.7 aż do 24.1 sekundy. Drugie rozwiązanie nie wykorzystuje bowiem w pełni szybkiej *pamięci podręcznej* (ang. *cache*) komputera.

6.4.6. Mnożenie macierzy

Niech A — macierz typu $n \times m$ oraz B — macierz typu $m \times r$. Wynikiem mnożenia macierzy $A \cdot B$ jest macierz C typu $n \times r$, dla której

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj},$$

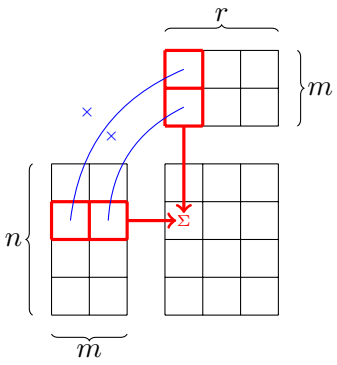
gdzie $0 \leqslant i < n$, $0 \leqslant j < r$ (por. rys. 6.1).

A oto kod funkcji służącej do wyznaczenia iloczynu dwóch macierzy.

```
1  double** mnoz(double** A, double** B, int n, int m, int r)
2  {
3     assert(n > 0 && m > 0 && r > 0);
4
5     double** C = tworzmacierz(n,r);
6
7     for (int j=0; j<r; ++j)
8     {
9        for (int i=0; i<n; ++i)
10       {
11          C[i][j] = 0.0;
12          for (int k=0; k<m; ++k)
13             C[i][j] += A[i][k] * B[k][j];
14       }
15    }
16
17   return C;  /* nie zapomnij o zwolnieniu pamięci dalej! */
18 }
```

Ciekawostka

Podany algorytm wykonuje nmr operacji mnożenia, co dla macierzy kwadratowych o rozmiarach $n \times n$ daje n^3 operacji. Lepszy (i o wiele bardziej skomplikowany) algorytm, autorstwa Coppersmitha i Winograda, ma złożoność rzędu $n^{2,376}$.



Rys. 6.1. Ilustracja algorytmu mnożenia macierzy

6.4.7. Rozwiązywanie układów równań liniowych

Dany jest oznaczony układ równań postaci

$$A\mathbf{x} = \mathbf{b},$$

gdzie A jest macierzą nieosobliwą typu $n \times n$, $\mathbf{b}$ jest n-elementowym wektorem wyrazów wolnych oraz $\mathbf{x}$ jest n-elementowym wektorem niewiadomych.

Rozpatrywany układ równań można zapisać w następującej postaci.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

Uproszczona metoda *eliminacji Gaussa* polega na sprowadzeniu macierzy rozszerzonej $[A|\mathbf{b}]$ do postaci schodkowej $[A'|\mathbf{b}']$:

$$\left[\begin{array}{cccc|c} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} & b_0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} & b_{n-1} \end{array}\right]$$

Dokonuje się tego za pomocą następujących tzw. *operacji elementarnych* (wierszowych):

1. pomnożenie dowolnego wiersza macierzy rozszerzonej przez niezerową stałą,
2. dodanie do dowolnego wiersza kombinacji liniowej pozostałych wierszy.

Następnie uzyskuje się wartości wektora wynikowego $\mathbf{x}$ korzystając z tzw. *eliminacji wstecznej*:

$$x_i = \frac{1}{a'_{i,i}} \left(b'_i - \sum_{k=i+1}^{n-1} a'_{i,k} x_j \right).$$

dla kolejnych $i = n-1, n-2, \ldots, 0$.

Rozpatrzmy przykładową implementację tej metody w języku C++.

```
void schodkowa(double** A, double* b, int n);
double* eliminwst(double** A, double* b, int n);

double* gauss(double** A, double* b, int n)
{
    assert(n > 0);
    schodkowa(A, b, n);
    return eliminwst(A, b, n);
}
```

Funkcja sprowadzająca macierz do postaci schodkowej (która modyfikuje wejściową macierz):

```
void schodkowa(double** A, double* b, int n)
{
    for (int i=0; i<n; ++i)
    { /* dla każdego wiersza */
       assert(fabs(A[i][i]) > 1e-12);//jeśli nie=>macierz osobliwa (
           dlaczego tak?)

       for (int j=i+1; j<n; ++j)
       { // wiersz j:=(wiersz j)-A[j,i]/A[i,i]*(wiersz i)

          for (int k=i; k<n; ++k)
             A[j][k] -= A[j][i]/A[i][i]*[i][k];

          b[j] -= A[j][i]/A[i][i]*b[i];
       }
    }
}
```

Eliminacja wsteczna:

```
double* eliminwst(double** A, double* b, int n)
{
    double* x = new double[n];
    for (int i=n-1; i>=0; --i)
    {
       x[i] = b[i];
       for (int k=i+1; k<n; ++k)
       {
          x[i] -= A[i][k] * x[k];
       }
       x[i] /= A[i][i];
    }
    return x;
}
```

6.5. Ćwiczenia

Zadanie 6.1. Zaimplementuj samodzielnie funkcje z biblioteki `<cstring>`: `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, `strstr()`, `strchr()`, `strrchr()`.

Zadanie 6.2. Napisz funkcję, która tworzy i zwraca nową, dynamicznie stworzoną kopię danego napisu z zamienionymi wszystkimi małymi literami alfabetu łacińskiego (bez polskich znaków diakrytycznych) na wielkie.

Zadanie 6.3. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z końca danego napisu. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.4. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z początku danego łańcucha znaków. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.5. Napisz funkcję, która usuwa z danego napisu wszystkie znaki niebędące cyframi bądź kropką. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.6. Napisz funkcję, która odwróci kolejność znaków w danym napisie. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.7. Napisz funkcję obliczającą, ile razy w danym napisie występuje dany znak.

Zadanie 6.8. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany inny napis, np. w *"ababbababa"* łańcuch *"aba"* występuje 3 razy.

Zadanie 6.9. Napisz funkcję, która dla danych dwóch napisów `n1` i `n2` oraz znaku `c` utworzy nowy, dynamicznie alokowany napis, będący złączeniem `n1` i `n2` z pominięciem wszystkich wystąpień znaku `c`. Dla przykładu, dla napisów `"ala"` i `"ma kota"` oraz znaku `'a'` powinien być zwrócony napis `"lm kot"`.

Zadanie 6.10. Napisz funkcję, która złączy dwa dane napisy `n1` i `n2` i podwoi wszystkie wystąpienia danego znaku `c`. Dla przykładu, dla napisów `"ulubione"`, `"turuburu"` i znaku `'u'` powinniśmy otrzymać nowy napis `"uuluubionetuuruubuuruu"`.

Zadanie 6.11. Palindrom to ciąg liter, które są takie same niezależnie od tego, czy czytamy je od przodu czy od tyłu, np. *kobyłamamałybok*, *możejutrotadamasamadatortujeżom*, *ikarłapałraki*. Napisz funkcję, która sprawdza czy dany napis jest palindromem.

Zadanie 6.12. Napisz funkcję `char* napscal(char* napis1, char* napis2)`, która przyjmuje na wejściu dwa napisy, których litery są posortowane (w porządku leksykograficznym). Funkcja zwraca nowy, dynamicznie alokowany napis będący scaleniem (z zachowaniem porządku) tych napisów, np. *"alz"* i *"lopxz"* wynikiem powinno być *"allopxzz"*.

Zadanie 6.13. Napisz funkcję `int napporownaj(char* napis1, char* napis2)`, która zwraca wartość równą 1, jeśli pierwszy napis występuje w porządku leksykograficznym przed napisem drugim, wartość -1, jeśli po napisie drugim, bądź 0 w przypadku, gdy napisy są identyczne. Dla przykładu, `napporownaj("ola", "ala")==-1` oraz `napporownaj("guziec", "guziec2")==1`.

Zadanie 6.14. Napisz funkcję `char* napsortlitery(char* napis)`, która zwraca nowy, dynamicznie alokowany napis składający się z posortowanych (dowolnym znanym Ci algorytmem) liter napisu wejściowego. Dla przykładu, `napsortlitery("abacdcd")` powinno dać w wyniku napis *"aabccdd"*.

Zadanie 6.15. Napisz funkcję `void sortuj(char* napisy[], int n)`, która wykorzysta dowolny algorytm sortowania tablic oraz funkcję `napporownaj(...)` do leksykograficznego uporządkowania danej tablicy napisów.

Zadanie 6.16. Napisz funkcję `char losuj()`, która wykorzystuje funkcję `rand()` z biblioteki `<cstdlib>`: do „wylosowania” wielkiej litery z alfabetu łacińskiego. Zakładamy, że funkcja `srand()` została wywołana np. w funkcji `main()`.

Zadanie 6.17. Napisz funkcję `char* permutuj(char* napis)`, która zwraca nowy, dynamicznie alokowany napis będący losową permutacją znaków występujących w napisie przekazanym jako argument wejściowy.

Zadanie 6.18. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ oraz liczba $k \in \mathbb{R}$. Napisz funkcję, która wyznaczy wartość kA, czyli implementującą mnożenie macierzy przez skalar.

Zadanie 6.19. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ oraz wektor $\mathbf{b} \in \mathbb{R}^n$. Napisz funkcję, która zwróci macierz $[A|\mathbf{b}]$, czyli A rozszerzoną o nową kolumnę, której wartości pobrane są z $\mathbf{b}$.

Zadanie 6.20. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ oraz wektor $\mathbf{b} \in \mathbb{R}^m$. Napisz funkcję, która zwróci macierz A rozszerzoną o nowy wiersz, którego wartości pobrane są z $\mathbf{b}$.

Zadanie 6.21. Dana jest macierz $A \in \mathbb{R}^{3\times 3}$. Napisz funkcję wyznaczającą $\det A$.

Zadanie 6.22. Dana jest macierz kwadratowa A o 4 wierszach i 4 kolumnach zawierająca wartości rzeczywiste. Napisz rekurencyjną funkcję, która zwróci wyznacznik danej macierzy. Skorzystaj wprost z definicji wyznacznika. Uwaga: taka metoda jest zbyt wolna, by korzystać z niej w praktyce.

Zadanie 6.23. Napisz funkcję, która rozwiązuje układ 3 równań liniowych korzystając z metody Cramera (dana jest macierz 3×3 i wektor). Poprawnie identyfikuj przypadek, w których dany układ nie jest oznaczony.

Zadanie 6.24. Dana jest macierz $A \in \mathbb{Z}^{n\times m}$. Napisz funkcję zwracającą jej transpozycję.

Zadanie 6.25. Dana jest macierz $A \in \mathbb{Z}^{n\times m}$. Napisz funkcję, która dla danego $0 \leqslant i < n$ i $0 \leqslant j < m$ zwróci podmacierz powstałą przez usunięcie z A i-tego wiersza i j-tej kolumny.

Zadanie 6.26. Dana jest kwadratowa macierz A o wartościach całkowitych. Napisz funkcję, która sprawdzi, czy macierz jest symetryczna. Zwróć wynik typu `bool`.

Zadanie 6.27. Dana jest macierz kwadratowa A o wartościach rzeczywistych typu $n \times n$. Napisz funkcję, która zwróci jej ślad, określony jako $\mathrm{tr}(A) = \sum_{i=0}^{n-1} a_{i,i}$.

Zadanie 6.28. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej diagonalę w postaci tablicy jednowymiarowej.

Zadanie 6.29. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej macierz diagonalną, czyli macierz z wyzerowanymi wszystkimi elementami poza przekątną.

Zadanie 6.30. Kwadratem łacińskim stopnia n nazywamy macierz kwadratową typu $n\times n$ o elementach ze zbioru $\{1, 2, \ldots, n\}$ taką, że żaden wiersz ani żadna kolumna nie zawierają dwóch takich samych wartości. Napisz funkcję, która sprawdza, czy dana macierz jest kwadratem łacińskim. Zwróć wynik typu `bool`.

Zadanie 6.31. Kwadratem magicznym stopnia n nazywamy macierz kwadratową typu $n\times n$ o elementach ze zbioru liczb naturalnych taką, że sumy elementów w każdym wierszu, w każdej kolumnie i na każdej z dwóch przekątnych są takie same. Napisz funkcję, która sprawdza, czy dana macierz jest kwadratem magicznym. Zwróć wynik typu `bool`.

Zadanie 6.32. Napisz funkcję, która przyjmuje jako parametr macierz `A` o elementach typu `bool` i zwróci nową, dynamicznie alokowaną macierz, która zawiera te i tylko te wiersze z `A`, które zawierają nie mniej wartości `true` niż `false`.

Zadanie 6.33. Napisz funkcję, która przyjmuje jako parametr macierz `A` o elementach typu `char` i zwróci nową, dynamicznie alokowaną macierz, która zawiera te i tylko te kolumny z `A`, które zawierają parzystą lecz niezerową liczbę wielkich liter.

Zadanie 6.34. Stwórz funkcje `double* wstawW(double* M, int n, int m, int i, double c)` oraz `wstawK()`, która wstawi do danej macierzy M nowy wiersz (nową kolumnę) pomiędzy wierszami (kolumnami) i oraz $i + 1$, oraz wypełni go wartością c.

⋆ **Zadanie 6.35.** Napisz funkcję, która sprawdzi czy dana macierz może być otrzymana z macierzy 1×1 przez ciąg operacji `wstawW()` i `wstawK()`, zdefiniowanych w zadaniu 6.34.

⋆ **Zadanie 6.36.** Napisz nierekurencyjną funkcję wyznaczającą wartość wyrażenia:

- $f(n, m) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (f(i, j) - i - j)^2$,
- $f(n, m) = \begin{cases} 1 & \text{dla } n = 0 \text{ lub } m = 0, \\ \|\{(i, j) \in \mathbb{N}^2 : i < n, j < m, f(i, j) < (i - j)^2\}\| & \text{dla } n, m > 0, \end{cases}$

gdzie $n, m \in \mathbb{N}$.

6.6. Wskazówki i odpowiedzi do ćwiczeń

Odpowiedź do zadania 6.1.

```
int strlen(char *s)
{
  int i = 0;
  while (s[i]!=0) i++;
  return i;
}
```

```
char* strcpy(char *cel, char *zrodlo)
{
  // zalozenie - pamiec dla tablicy przydzielona przed wywolaniem
      funkcji (cel = new char[strlen(zrodlo)+1];)

  int i = 0;
  while (zrodlo[i]!=0)
  {
     cel[i] = zrodlo[i];
     i++;
  }
  cel[i] = '\0'; // wstaw "wartownika"

  return cel;
}
```

```
int strcmp(char *s1, char *s2)
{
  int i = 0;
  int j = 0;
  while (s1[i]!=0 && s1[i] == s2[j])
  {
     i++;
     j++;
  }

  return s1[i] - s2[j];
}
```

```
char* strcat(char* cel, char* zrodlo)
{
  // zalozenie - napis w tablicy cel ma n znakow,
  // napis w tablicy zrodlo ma m znakow,
  // pamiec przydzielona na tablice cel wynosi n+m+1 znakow
  // (bajt zerowy w cel nie musi wystepowac przeciez na koncu
      tablicy!)
  int n = strlen(cel); // cel[n] == '\0' (!)
  int j = 0;
  while (zrodlo[j]!=0)
  {
     cel[n] = zrodlo[j];
     n++;
     j++;
  }
  cel[n] = '\0';

```

```
17   return cel;
18 }
```

```
1  char* strchr(char* nap, char znak)
2  {
3    int i=0;
4    while (nap[i] != 0)
5    {
6       if (nap[i] == znak)
7          return &(nap[i]); // znaleziony - zwracamy podnapis
8
9       i++;
10   }
11
12   // tutaj doszedłszy, stwierdzamy, że znak wcale nie występuje w
         napisie
13   return NULL; // albo return 0; - adres zerowy to "nigdzie"
14 }
```

```
1  char* strrchr(char* nap, char znak)
2  {
3    int i=strlen(nap)-1; // zaczynamy od ostatniego znaku drukowanego
4    while (i>=0)
5    {
6       if (nap[i] == znak)
7          return &(nap[i]); // znaleziony - zwracamy podnapis
8
9       i--;
10   }
11
12   return NULL;
13 }
```

```
1  char* strstr(char* nap1, char* nap2)
2  {
3    int n = strlen(nap1);
4    int m = strlen(nap2);
5
6    assert(n>=m);
7
8    int i, j;
9    for (i=0; i<n-m+1; ++i)
10   {
11      for (j=0; j<m; ++j)
12      {
13         if (nap1[i+j] != nap2[j]) break; // przerywamy petle
14      }
15
16      if (j == m) // doszlismy do konca petli => znaleziony!
17         return &(nap1[i]);
18   }
19
20   return NULL; // on ne le trouve pas :-)
21 }
```

□

Wskazówka do zadania 6.22. Wyznacznik macierzy 4×4 można policzyć ze wzoru

$$\det A = \sum_{i=0}^{3} (-1)^{i+j} a_{ij} \det A_{i,j},$$

gdzie j jest dowolną liczbą ze zbioru $\{0, 1, 2, 3\}$, a $A_{i,j}$ jest podmacierzą 3×3 powstałą przez opuszczenie i-tego wiersza i j-tej kolumny. Do wyznaczenia $\det A_{i,j}$ można skorzystać z nieco zmodyfikowanej funkcji z poprzedniego zadania, której należy przekazać A, i oraz j. □