10 Algoritmy pro práci s vektory a s maticemi

Kreslíková, 27. 10. 2023

10.1 Úvod

Jednoduchá a vícerozměrná pole jsou velmi často používanými datovými typy. Umožňují totiž ukládat a zpracovávat velké objemy dat. Existuje ale také mnoho matematických úloh, které využívají pole nikoli jen pro svou schopnost uchovávat velké objemy dat, ale také pro svou schopnost pracovat snadno jak s jednotlivými prvky pole, tak s polem jako celkem. Jde o úlohy využívající vektory a matice.

V této kapitole si blíže vyzkoušíme, jak vektory a matice realizovat pomocí polí a jak realizovat operace s nimi pomocí jazyka C.

Dále si zde ukážeme, že pole a matice se nevyužívají pouze jako abstrakce vektorů a matic, ale že najdou využití i při jiných typech úloh, jako je například zpracování textových řetězců nebo realizace různých hlavolamů či her.

10.2 Operace s vektory

Uspořádanou *n*-tici reálných čísel $a = (a_1,a_2,a_3, ...,a_n)$ nazýváme *n*-rozměrným reálným (aritmetickým) vektorem.

Pořadí	1.	2.	3.	4.	…	n.
Hodnota	a_1	a_2	a_3	a_4	…	a_n
Index	0	1	2	3	…	n-1

10.2.1 Sémantický význam v programování

Datový typ pole je *N* položek stejného typu opakovaně uložených za sebou v paměti. Na jednotlivé prvky se lze odkazovat pořadovým indexem. V jazyce C vždy indexy začínají od nuly. U pole délky *N* tedy bude rozsah použitelných indexů od *0* do *N–1*.

Vztah polí a vektorů

Datový typ pole má v programovacích jazycích obecnější použití nežli jen ve významu vektoru. Skutečný význam použitého pole závisí na povaze úlohy a interpretaci. Pole znaků takto například můžeme považovat za textový řetězec, pole čísel můžeme interpretovat jako obecnou posloupnost nebo v jiné úloze jako vektor popisující vztahy v *N*-rozměrném prostoru.

Operace nad polem

- přiřazení hodnoty určitému prvku, který je zadán indexem,
- přiřazení stejné hodnoty všem prvkům,
- vyhledání prvku s určitou hodnotou,
- seřazení prvků,
- operace pro jednotlivé prvky pole odpovídající typu prvku,
- vložení prvku(ů) mezi jiné prvky, vyjmutí prvku + odpovídající posun prvků.

Operace nad vektory

- násobení vektoru konstantou,
- skalární součin vektorů,
- vektorový součin vektorů,
- součet vektorů,
- rozdíl vektorů atd.

Implementační poznámka: pole se v jazyce C předávají funkcím automaticky odkazem. Vzpomeňte si, že identifikátor pole má stejný význam, jako konstantní ukazatel na první prvek pole. Při volání funkce se před parametr typu pole nepíše referenční operátor (&).

10.2.2 Násobení vektoru konstantou.

Pro násobení vektoru konstantou *K* platí vztah:

$$K.\mathbf{a} = \mathbf{b}$$

Nebo vektorově:

$$K\vec{a} = \vec{b}$$

Hodnoty jednotlivých prvků nového vektoru pak vypočítáme podle vztahu:

$$b_i = K.a_i \qquad i = 1, 2, \ldots, n$$

Příklad: Násobení vektoru konstantou. Varianta s vektorem obecné délky. Výsledkem je modifikovaný vektor.

```
void multConst(int v[], int n, int k)
{
  for(int i = 0; i < n; i++)
  {
    v[i] *= k;
  }
}
```

Příklad: Násobení vektoru konstantou. Varianta s vektorem obecné délky, který je uložen ve struktuře a alokován dynamicky. Výsledkem bude nově alokovaný vektor.

```
typedef struct tvector
{
  int n;
  int *v;
} TVector;

TVector multConst(const TVector *v, int c)
{
  TVector w = allocVect(v->n);
  for(int i = 0; i < w.n; i++)
  {
    w.v[i] = v->v[i] * c;
  }
  return w;
}
```

Poznámka: V dalších příkladech budeme pro jednoduchost používat staticky alokované pole o velikosti *N*, pokud nebude výslovně uvedeno něco jiného.

```
#define N 100
int pole[N];
```

10.2.3 Součet dvou vektorů.

Pro součet dvou vektorů o stejném rozměru platí:

$$\mathbf{a} + \mathbf{b} = \mathbf{c}$$

$$\vec{a} + \vec{b} = \vec{c}$$

Hodnoty prvků tohoto nového vektoru se pak počítají podle vztahu:

$c_i = a_i + b_i$, kde i = 1, 2, …, n

Příklad: Součet dvou vektorů uložených v poli *a* a *b*, výsledek bude uložen do pole *c*.

```
void addVect(int a[], int b[], int c[])
{
  for(int i = 0; i < N; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```

10.2.4 Skalární součin dvou vektorů

Pro skalární součin dvou vektorů o stejném rozměru platí:

$$\mathbf{a} \,.\, \mathbf{b} = \mathbf{c}$$

$$\vec{a} \cdot \vec{b} = c$$

Hodnota tohoto součinu se počítá podle vztahu:

$$c = \sum_{i=1}^{n} a_i b_i$$

Příklad: Skalární součin dvou vektorů o stejném rozměru.

```
double multScalar(int a[], int b[])
{
  double scalar = 0;
  for(int i = 0; i < N; i++)
  {
    scalar += a[i]*b[i];
  }
  return scalar;
}
```

10.2.5 Vektorový součin

Vektorovým součinem *(a×b)* vektorů *a(a1, a2, a3), b(b1, b2, b3),* nazýváme vektor:

$$\vec{w} = \vec{a} \times \vec{b}$$

$$\vec{w} = \left(\begin{vmatrix} a_2, & a_3 \\ b_2, & b_3 \end{vmatrix}, \begin{vmatrix} a_3, & a_1 \\ b_3, & b_1 \end{vmatrix}, \begin{vmatrix} a_1, & a_2 \\ b_1, & b_2 \end{vmatrix} \right)$$

Příklad: Funkce pro výpočet vektorového součinu dvou vektorů. Výsledek bude uložen do pole *w*.

```
#define N 3
...
void multVect(int a[], int b[], int w[])
{
  w[0] = a[1]*b[2] – a[2]*b[1];
  w[1] = a[2]*b[0] – a[0]*b[2];
  w[2] = a[0]*b[1] – a[1]*b[0];
}
```

10.2.6 Hledání prvočísel – Eratostenovo síto

Často se řeší problém zjištění všech prvočísel od *2* do zadaného čísla *N*. Obvykle se používá metoda, při níž se postupně berou všechna čísla *X* počínaje *2* a konče *N* a tato se dělí všemi čísly od *2* do odmocniny z *X* a zjišťují se zbytky po dělení. Je- li některý zbytek roven nule, pak číslo není prvočíslo, v opačném případě je prvočíslo. Při řešení lze použít různé optimalizace (např. nedělíme číslem *2*, případně dělíme jen prvočísly do odmocniny z *X*).

Kromě tohoto algoritmu existuje ještě jeden, který je nesrovnatelně efektivnější. Nazývá se ***Eratostenovo síto***.

Algoritmus: Výpočet prvočísel pomocí Eratostenova síta s využitím pole.

1. vytvoříme bitové pole tak, že pro každé přirozené číslo od *2* do *N* vyhradíme jeden (bit), *N* je horní hranice výpisu prvočísel (*N-1*).
2. index pole bude uvádět číslo, hodnota bitu rovná *1* bude znamenat, že číslo je prvočíslo, *0,* že není prvočíslo.
3. na začátku pole inicializujeme tak, jako by všechna čísla byla prvočísly.

4. procházíme prvky pole a vyhledáváme první nenulový bit (v prvním průchodu to bude u indexu *2*, dále *3, 5, 7* atd.).
5. nalezené číslo (*P*) je prvočíslo, proto na jeho místě necháme v poli jedničku.
6. nyní bereme všechny násobky *P* a všech čísel neoznačených nulou v předchozích iteracích počínaje *P* (*P*×*P*, *P*×*Q*), *P*×*R*, atd., kde *Q, R*, … jsou čísla, která nebyla dříve označena nulou) a na místa těchto čísel ukládáme nuly (zcela jistě to nejsou prvočísla).
7. postupujeme tak dlouho, dokud hodnota násobku nepřesáhne zadané *N*.
8. vrátíme se k bodu 4) a cyklus opakujeme od současného *P* dotud, dokud není *P* větší než odmocnina z *N*.
9. pole je správně vyplněno, jsme hotovi a chceme-li prvočísla zobrazit, vypíšeme indexy těch prvků pole, kde je jednička.

Ukázka: Aplikace Eratostenova síta na pole o dvaceti prvcích.

Pole po inicializaci 1 - značí je prvočíslo, 0 - značí není prvočíslo

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Najdeme číslo 2, označíme násobky dvou (2×2, 3×2, 4×2, ...)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Najdeme číslo 3, označíme násobky tří (3×3, 5×3, 7×3, ...)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1

Dále čísla 5, 7, … dokud *P* není větší než odmocnina z *N*.

Příklad: Implementace algoritmu Eratostenovo síto v jazyce C.

```
#include <stdio.h>
#include <stdbool.h>
#define N 20
int main(void)
{
  bool a[N];
  for(int i=2; i<N; i++)
    a[i] = true;
  for(int i=2; i<N; i++)
  { // hledání prvočísel
    if (a[i])
      for(int j=i*i; j<N; j+=i)
        a[j] = false;
  }
  for(int i=2; i<N; i++)
  { // výpis prvočísel
    if (a[i])
      printf("%d ", i);
  }
  printf("\nkonec\n");
  return 0;
}
```

Složitost algoritmu: N+N/2+N/3+N/5 + … ~ N ln N.

Úkol: uvedený program neodpovídá zcela přesně výše uvedenému algoritmu. Analyzujte program a upravte ho tak, aby mu odpovídal.

Algoritmus: Výpočet prvočísel pomocí Eratostenova síta pomocí množiny

1. zvolíme množinu zvanou síto a naplníme ji celými čísly z intervalu *2* .. *n/2+1*. Prvky množiny reprezentují lichá čísla (prvek *2* reprezentuje číslo *3*, prvek *3* číslo *5*, obecně prvek *i* číslo *2i–1*).
2. vybereme nejmenší číslo ze síta – je to prvočíslo.
3. Vybrané číslo zahrneme do množiny prvočísel.
4. ze síta odstraníme toto číslo a také všechny jeho násobky.
5. pokud síto není prázdné, opakujeme body 2 až 5.

10.2.7 Reverze řetězce

Reverzí řetězce se myslí úloha, kdy v textovém řetězci dojde k výměně pořadí jednotlivých písmen. Při reverzi řetězce dojde k výměně prvního a posledního znaku, druhého a předposledního, a stejně tak pro všechny ostatní znaky.

0	1	2	3	4	5	6	7	8	9
'D'	'o'	'b	'r'	'y'	' '	'd'	'e'	'n'	'\0'

0	1	2	3	4	5	6	7	8	9
'n'	'e'	'd	' '	'y'	'r '	'b'	'o'	'D'	'\0'

Tento problém lze řešit dvěma způsoby – rekurzivně nebo pomocí cyklu.

Příklad: Reverze řetězce rekurzívně – neefektivní.

```
int _revert(char *s, int n)
{
  char c = s[n];
  if (c == '\0')
    return n;
  int length = _revert(s, n+1);
  s[length-(n+1)] = c;
  return length;
}

void revert(char *s)
{
  _revert(s, 0);
}
```

Příklad: Reverze řetězce – záměnou prvního a posledního, druhého a předposledního, atd., znaku v řetězci.

```
inline void swap(char *a, char *b)
{
  *a ^= *b; *b ^= *a; *a ^= *b;
}

void revert(char *s)
{
  int length = strlen(s);
  for(int i = 0; i < length/2; i++)
    swap(&s[i], &s[length-i-1]);
}
```

Obsah proměnných *a* a *b* při provádění funkce *swap()*

a	0	0	1	0	0	0	1	0
b	0	1	0	1	0	1	1	0
a	0	1	1	1	0	1	0	0
b	0	0	1	0	0	0	1	0
a	0	1	0	1	0	1	1	0

10.3 Operace s maticemi

Soubor čísel uspořádaných do řádků a sloupců nazýváme maticí.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = \left(a_{ij}\right)_{\substack{i=1,2,\dots,m \\ j=1,2,\dots,n}}$$

Čísla a_{ij} se nazývají prvky matice. Matice ukládáme do dvourozměrného pole.

Matice je:

- čtvercová řádu n typ (n, n)
- obdélníková typ (m, n), $m \neq n$
- řádková typ (1, n)
- sloupcová typ (m, 1)
- nulová (značíme $\boldsymbol{0}$) všechny prvky jsou rovny nule,
- jednotková (značíme $\boldsymbol{I}$) - čtvercová matice, prvky hlavní diagonály jsou rovny jedné, všechny ostatní prvky matice jsou rovny nule,
- transponovaná k matici A, když zaměníme řádky matice A za sloupce (značíme A^T),
- symetrická - čtvercová matice A, pro kterou platí $A = A^T$,
- trojúhelníkový tvar matice – všechny prvky hlavní diagonály jsou nenulové a všechny prvky pod hlavní diagonálou jsou rovny nule.

10.3.1 Algoritmy pro práci s maticemi

V této kapitole se budeme zabývat základními algoritmy pro práci s maticemi. Hned na začátku je dobré si uvědomit, že ne všechny operace lze provádět na libovolných maticích. Například průchod po diagonále má většinou smysl pouze u čtvercových matic, sčítání a odčítání má zase smysl pouze na rozměrově kompatibilních maticích. Při skutečné implementaci je potřeba rozměry matic kontrolovat (pokud to za nás nedělá typová kontrola jazyka). Měla by to být vždy první operace před vlastní implementací daného algoritmu.

10.3.2 Prvky na hlavní diagonále

Operace: Přístup k prvkům na hlavní diagonále matice ***A***(*n*,*n*).
Prvky hlavní diagonály: $a_{11}, a_{22}, ..., a_{nn}$
Indexy prvků na hlavní diagonále čtvercové matice:

A[5][5]

0,0	**0,1**	**0,2**	**0,3**	**0,4**
1,0	**1,1**	**1,2**	**1,3**	**1,4**
2,0	**2,1**	**2,2**	**2,3**	**2,4**
3,0	**3,1**	**3,2**	**3,3**	**3,4**
4,0	**4,1**	**4,2**	**4,3**	**4,4**

Příklad: Nadále budeme pracovat se statickými dvojrozměrnými poli.

```
#define R 10
#define S 20

int matice1[R][S]; // obdélníková matice
int matice2[R][R]; // čtvercová matice
```

Příklad: Tisk prvků na hlavní diagonále (a_{ii}) .

```
void printDiagonal(int a[R][R])
{
  for(int i = 0; i < R; i++)
    printf("%d ", a[i][i]);

  printf("\n");
}
```

10.3.3 Prvky na vedlejší diagonále

Operace: Přístup k prvkům na vedlejší diagonále matice ***A***(*n*,*n*).
Indexy prvků na vedlejší diagonále čtvercové matice:

A[5][5]

0,0	**0,1**	**0,2**	**0,3**	**0,4**
1,0	**1,1**	**1,2**	**1,3**	**1,4**
2,0	**2,1**	**2,2**	**2,3**	**2,4**
3,0	**3,1**	**3,2**	**3,3**	**3,4**
4,0	**4,1**	**4,2**	**4,3**	**4,4**

Příklad: Tisk prvků na vedlejší diagonále ($a_{i,n-(i+1)}$).

```
void printSDiagonal(int a[R][R])
{
  for(int i = 0; i < R; i++)
    printf("%d ", a[i][R-(i+1)]);

  printf("\n");
}
```

10.3.4 Prvky horní trojúhelníkové matice

Operace: Indexace prvků horní trojúhelníkové matice:

A[5][5]

0,0	**0,1**	**0,2**	**0,3**	**0,4**
1,0	**1,1**	**1,2**	**1,3**	**1,4**
2,0	**2,1**	**2,2**	**2,3**	**2,4**
3,0	**3,1**	**3,2**	**3,3**	**3,4**
4,0	**4,1**	**4,2**	**4,3**	**4,4**

Příklad: Tisk prvků horní trojúhelníkové matice.

```
void printUpTriangle(int a[R][R])
{
  for(int i = 0; i < R; i++)
    for(int j = i; j < R; j++)
      printf("%d ", a[i][j]);

  printf("\n");
}
```

Úkol: Definujte funkce pro tisk prvků:

- na hlavní diagonále a pod hlavní diagonálou (dolní trojúhelníkové matice)
- na vedlejší diagonále a nad vedlejší diagonálou
- na vedlejší diagonále a pod vedlejší diagonálou

10.3.5 Rovnost matic

Matice A, B stejného typu (m, n).

$$\boldsymbol{A} = \boldsymbol{B} \Leftrightarrow \underset{\substack{i=1,2,\ldots,m \\ j=1,2,\ldots,n}}{\forall} \left(a_{ij} = b_{ij}\right)$$

Příklad: Jsou si dvě matice rovné?

```
bool areEqual(int a[R][S], int b[R][S])
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      if (a[r][s] != b[r][s])
        return false;

  return true;
}
```

10.3.6 Součet matic

Matice *A, B* stejného typu (*m, n*).

$$\mathbf{A} + \mathbf{B} = \left(a_{ij} + b_{ij}\right)$$

Platí:

$$\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{B} + \boldsymbol{A}$$
$$\boldsymbol{A} + \left(\boldsymbol{B} + \boldsymbol{C}\right) = \left(\boldsymbol{A} + \boldsymbol{B}\right) + \boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$$

Příklad: Sečte dvě matice a výsledek uloží do pole a.

```
void addMatrix(int a[R][S], int b[R][S])
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      a[r][s] += b[r][s];
}
```

10.3.7 Násobení matice skalárem

Matice *A* typu (*m, n*).

$$k\boldsymbol{A} = k\left(a_{ij}\right) = \left(ka_{ij}\right)$$

Platí:

$$\left(k_1 + k_2\right)\boldsymbol{A} = k_1\boldsymbol{A} + k_2\boldsymbol{A}$$
$$k\left(\boldsymbol{A} + \boldsymbol{B}\right) = k\boldsymbol{A} + k\boldsymbol{B}$$

Příklad: Vynásobení matice skalárem.

```
void multConst(int a[R][S], int k)
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      a[r][s] *= k;
}
```

10.3.8 Násobení matic

Budeme násobit matici *A* typu *(m, p)* a matici *B* typu *(p, n)*. Výsledek uložíme do matice *C* typu *(m,n)*

$$\mathbf{AB} = \mathbf{C} = \left(c_{ij}\right) = \left(\sum_{k=1}^{p} a_{ik}b_{kj}\right)$$

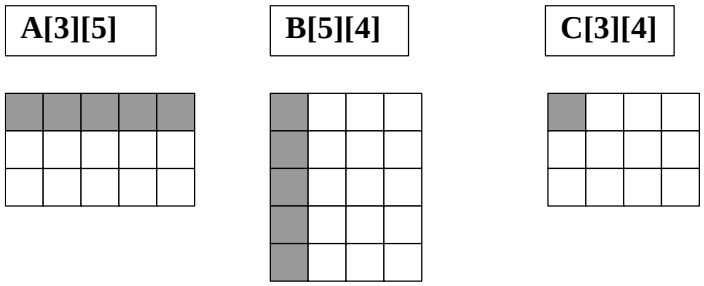
Platí:

$\boldsymbol{AB} \neq \boldsymbol{BA}$ neplatí komutativní zákon

jestliže $\boldsymbol{AB} = \boldsymbol{BA}$, pak matice $\boldsymbol{A}$ a $\boldsymbol{B}$ jsou zaměnitelné

$$\boldsymbol{A}\left(\boldsymbol{BC}\right) = \left(\boldsymbol{AB}\right)\boldsymbol{C} = \boldsymbol{ABC}$$
$$\boldsymbol{A}\left(\boldsymbol{B} + \boldsymbol{C}\right) = \boldsymbol{AB} + \boldsymbol{AC}$$
$$\left(\boldsymbol{A} + \boldsymbol{B}\right)\left(\boldsymbol{C} + \boldsymbol{D}\right) = \boldsymbol{A}\left(\boldsymbol{C} + \boldsymbol{D}\right) + \boldsymbol{B}\left(\boldsymbol{C} + \boldsymbol{D}\right)$$

A - *m* řádků, *p* sloupců
B - *p* řádků, *n* sloupců
C - *m* řádků, *n* sloupců



Příklad: Součin matic.

```
void multMatrix(int a[M][P], int b[P][N], int c[M][N])
{
  for (int i = 0; i < M; i++)
  {
    for (int j = 0; j < N; j++)
    {
      c[i][j] = 0;
      for (int k = 0; k < P; k++)
        c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

10.3.9 Osmisměrka

Další zajímavou úlohou, kterou lze řešit pomocí matic je luštění osmisměrky. Jde v podstatě o obdélníkovou matici písmen, přičemž jsou v ní ve všech osmi směrech ukryta slova. Úkolem je všechna zadaná slova najít a vyškrtat. Zbylá písmena tvoří tajenku.

K	A	L	T	J	S	H	O	D	A
L	L	P	U	K	L	T	O	A	T
A	K	T	A	A	K	A	A	R	R
S	A	A	N	L	A	K	P	E	A
A	R	P	O	V	P	T	O	K	K
R	H	O	M	O	L	I	C	E	A
K	O	L	S	P	E	K	E	S	R
O	R	A	O	C	A	A	L	T	P
S	P	O	K	V	S	T	I	A	A
M	A	T	K	A	F	T	K	A	T
A	I	A	K	O	S	T	K	A	Y

Hledaná slova:
ALKA HORA JUTA KAPLE KARPATY KARTA KASA KAVKA KLAS KOSMONAUT KOST KROK LAPKA MATKA OKRASA OPAT OSMA PAKT PATKA PIETA POCEL POVLAK PROHRA SEKERA SHODA SOPKA TAKT TAKTIKA TLAK VOLHA

A[7][6]

0,0	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**
1,0	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**
2,0	**2,1**	**2,2**	**2,3**	**2,4**	**2,5**
3,0	**3,1**	**3,2**	**3,3**	**3,4**	**3,5**
4,0	**4,1**	**4,2**	**4,3**	**4,4**	**4,5**
5,0	**5,1**	**5,2**	**5,3**	**5,4**	**5,5**
6,0	**6,1**	**6,2**	**6,3**	**6,4**	**6,5**

Při řešení této úlohy pomůže, když si sestrojíme tabulku pro všech osm možných směrů průchodu a zaznamenáme si do ní, jak se mají v daném směru měnit indexy pro průchod maticí a mezní podmínky, které indikují hranici matice v daném směru.

Tabulka, která popisuje pohyb v matici v osmi směrech:

A [R][S]

směr	r_index	S_index	podmínka
vlevo	0	-1	s_index==0
vpravo	0	+1	s_index==S-1
nahoru	-1	0	r_index==0
dolu	+1	0	r_index==R-1
vpravo_n	-1	+1	nalezněte podmínku
vpravo_d	+1	+1	nalezněte podmínku
vlevo_n	-1	-1	nalezněte podmínku
vlevo_d	+1	-1	nalezněte podmínku

10.3.10 Obecný průchod dvojrozměrnou maticí (pohyb na mapě)

Častou úlohou, při které se používají matice je průchod mapou, kdy stojíme na nějakém místě mapy a chceme se dostat na jiné místo pomocí algoritmu. Jako mapa slouží dvojrozměrná matice a prvky této matice určují například nadmořskou výšku, nebo vyznačují překážky. Další podobnou úlohou jsou například šachy, kde matice představuje šachovnici a hodnoty prvků indikují, jaká figurka na dané pozici stojí.

Základní pomůckou pro pohyb v takovém prostředí je tzv. ***směrový vektor***. Jde o dvourozměrný vektor, jehož složky určují, o kolik indexů se máme v dalším kroku v poli pohnout. Pokud si vhodně zvolíme význam jednotlivých složek, tedy první složka bude ovlivňovat pohyb ve vertikálním směru a druhá v horizontálním směru, bude operace posunu na další pozici podle tohoto vektoru velmi triviální. Tuto operaci bude tvořit prosté přičtení jednotlivých složek k indexům prvku na aktuální pozici.

Například hodnota vektoru (2, -1) znamená, že v dalším kroku se pohneme o dvě pozice dolů ve vertikálním směru a o jednu pozici vlevo v horizontálním směru.

A[7][6]

0,0	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**
1,0	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**
2,0	**2,1**	**2,2**	**2,3**	**2,4**	**2,5**
3,0	**3,1**	**3,2**	**3,3**	**3,4**	**3,5**
4,0	**4,1**	**4,2**	**4,3**	**4,4**	**4,5**
5,0	**5,1**	**5,2**	**5,3**	**5,4**	**5,5**
6,0	**6,1**	**6,2**	**6,3**	**6,4**	**6,5**

Pokud je potřeba projít celou maticí po přesně definované křivce, je vhodné si tuto křivku popsat pomocí tabulky.
Úkol: Vytvořte tabulku, která popisuje pohyb v matici po spirále. Složky vektoru *r_index* a *s_index* zde popisují pohyb po řádcích a po sloupcích, ve sloupci *podmínka* uveďte podmínku, při které dojde ke změně směru.

10.4 Shrnutí

Po přečtení této kapitoly a praktickém vyzkoušení řešených úloh je nyní čtenář schopen využívat jednorozměrné pole jako abstrakci matematického vektoru. Dokáže pomocí nich realizovat všechny vektorové operace a využívat je v programech.

Dále čtenář chápe a je schopen implementovat algoritmus pro hledání prvočísel pomocí Eratostenova síta. Také je schopen používat pole pro realizaci textového řetězce a je si vědom specifik, které tato realizace v jazyce C přináší. Čtenář je dále schopen pracovat s vícerozměrnými poli a používat dvourozměrná pole coby abstrakci matic. Dokáže implementovat základní maticové operace a realizovat průchody po diagonálách, horní nebo dolní trojúhelníkovou maticí. Ve všech těchto případech si je vědom, jak pracovat s mezemi polí.

Při řešení obecných úloh s dvourozměrným polem čtenář umí naprogramovat průchod libovolným z osmi přímých směrů a je schopen realizovat také obecné algoritmické průchody pomocí směrových přírůstků.

10.5 Úlohy k procvičení

1. Vytvořte program pro řešení zadané osmisměrky. Co je tajenkou osmisměrky uvedené v kapitole 9.2?
2. Vytvořte program, který vypočítá, zda se na šachovnici lze dostat koněm ze zadané pozice na jinou zadanou pozici.
3. Vytvořte algoritmus pro transpozici matice obecných rozměrů.
4. Vytvořte funkci pro výměnu dvou zadaných řádků matice.
5. Vytvořte funkci pro výměnu dvou zadaných sloupců matice.
6. Vytvořte funkci pro přičtení zadaného řádku matice k jinému zadanému řádku stejné matice.
7. Vytvořte funkci pro přičtení zadaného sloupce matice k jinému zadanému sloupci stejné matice.
8. Vytvořte program pro řešení soustavy rovnic Gaussovou eliminační metodou.

10.6 Kontrolní otázky

1. Jaký je zásadní rozdíl mezi vektorem a polem?
2. Jaký je rozdíl mezi skalárním a vektorovým součinem dvou vektorů? Lze sčítat vektory různých délek?
3. Jaká omezení klademe na vektory a matice při násobení konstantou?
4. K čemu slouží algoritmus nazývaný Eratostenovo síto? Vysvětlete stručně jeho princip.
5. Jaká omezení klademe na matice při jejich sčítání a násobení?
6. Která z verzí algoritmu pro reverzi řetězce (nebo pole) je efektivnější? Proč?