Univerzita sv. Cyrila a Metoda v Trnave

Fakulta prírodných vied
Katedra aplikovanej informatiky

Algoritmizácia a základy štruktúrovaného programovania v jazyku C

2. diel

Jana Jurinová

UNIVERZITA SV. CYRILA A METODA V TRNAVE

FAKULTA PRÍRODNÝCH VIED

Katedra aplikovanej informatiky

ALGORITMIZÁCIA A ZÁKLADY ŠTRUKTÚROVANÉHO PROGRAMOVANIA V JAZYKU C

2. diel

Jana Jurinová

Trnava, 2021

Autor:

Ing. Jana Jurinová, PhD.

Recenzenti:

doc. Ing. Michal Čerňanský, PhD.

Mgr. Ing. Roman Horváth, PhD.

Vysokoškolská učebnica bola schválená Edičnou radou Univerzity sv. Cyrila a Metoda v Trnave a vedením Fakulty prírodných vied UCM.

Za odbornú a jazykovú stránku tejto vysokoškolskej učebnice zodpovedá autorka.

Rukopis neprešiel redakčnou ani jazykovou úpravou.

Vydavateľ: Univerzita sv. Cyrila a Metoda v Trnave, 2021

1. vydanie

ISBN 978-80-572-0215-8

Predhovor

Táto vysokoškolská učebnica je určená predovšetkým študentom prvého ročníka študijného programu aplikovaná informatika, ale aj všetkým tým, ktorí chcú preniknúť do problematiky algoritmizácie a programovania. Učebnica ponúka ucelený pohľad algoritmického riešenia rôznych problémov. Čitateľ tu nájde materiál, ktorý poskytuje efektívny prístup k osvojeniu si ilustrovaných konceptov. Tomu zodpovedá aj štruktúra a radenie jednotlivých kapitol. Tento diel je pokračovaním učebnice s rovnakým názvom. Preto odporúčam čitateľom, ktorí nemajú osvojené základné vedomosti a koncepty súvisiace s algoritmizáciou a programovaním zamerané najmä na kľúčové poznatky, základné riadiace konštrukcie a jednoduché dátové typy, aby siahli najprv po prvom diele a chýbajúce poznatky si doštudovali. Táto učebnica predpokladá ich osvojenie a na viacerých miestach sa odkazuje na prvý diel. Druhý diel je primárne zameraný na ilustráciu algoritmického spracovania problémov využívajúcich štruktúrované dátové typy, návrh funkcií tak, aby sme navrhovali algoritmy, ktoré spĺňajú vlastnosti modifikovateľnosti a štruktúrovanosti a prácu s dynamickými údajovými štruktúrami, resp. externým zdrojom dát. Snahou je všetky objasňované koncepty ilustratívne podporiť príkladmi pri zachovaní jednoduchosti, pri poukázaní na širšie súvislosti a alternatívne možnosti riešenia.

Učebnica vznikla zo súboru prednáškových materiálov, poznámok, skúsenosti a spätnej väzby z praktických cvičení pri výučbe predmetov „algoritmy a dátové štruktúry I“ a „programovanie I“ študentov aplikovanej informatiky na bakalárskom stupni na Fakulte prírodných vied Univerzity sv. Cyrila a Metoda v Trnave. Spôsob výučby týchto predmetov sa vyvíjal v mnohých smeroch, čiastočne s cieľom zamerať sa na vedomostné zázemie prichádzajúcich študentov (nerozvinuté formálne zručnosti v predmetnej oblasti). V dôsledku toho boli témy starostlivo vybrané a zoskupené. Snahou nebolo vytvoriť encyklopedickú publikáciu, čo podporilo zahrnutie tém tradične zdôrazňovaných alebo vynechaných vo väčšine kníh o algoritmoch.

Podobne ako jej prvý diel aj tento vychádza v elektronickej podobe na základe dvoch obmedzujúcich faktorov. Grafická reprezentácia niektorých stredne zložitých algoritmov je natoľko rozsiahla, že ani pri použitom formáte A4 nie je možné v printovej podobe dané diagramy prezentovať v dostatočnej veľkosti. Druhým dôvodom je snaha uľahčiť čitateľom prácu, preto ako súčasť učebnice poskytujem zdrojové kódy všetkých riešených úloh, vďaka ktorým si môže čitateľ okamžite overiť ich funkčnosť a priamo s nimi pracovať. Odporúčam čitateľom, aby s týmito úlohami experimentovali, dotvárali ich, rozširovali, modifikovali,

pretože len aktívnym programovaním sa naučia a osvoja si požadované zručnosti. V každom kóde je vždy čo vylepšiť, prípadne zaujať iný postoj k riešeniu. Kódy programov je možné automaticky otvoriť v požadovanom softvéri po kliknutí na link, ktorý predstavuje jeho označenie. Pri uvádzaní komentárov v programe sú tieto výhradne uvádzané bez diakritiky. Treba zdôrazniť aj zastúpenie anglického jazyka vo výrazových prostriedkoch používaných v programovaní. Pre programátora je znalosť anglického jazyka prakticky nevyhnutná, aspoň na úrovni čítania a porozumenia dokumentácií. Preto väčšina slovenských kľúčových termínov použitých v učebnici je na lepšie pochopenie interpretovaná aj v anglickom jazyku.

Pri čítaní jednotlivých kapitol v chronologickom poradí buďte, prosím, trpezliví. Miestami sú používané pojmy, informácie, ktoré sú detailnejšie vysvetlené až nižšie, resp. boli objasnené v prvom diele. Je tak z viacerých dôvodov, ale hlavným je vyhnúť sa opakovaniu poskytovaných informácií a nenavyšovať umelo rozsah danej učebnice. Ak by ste však aj po preštudovaní celej učebnice, resp. jej prvého dielu a napriek mojej intenzívnej snahe opísať a poskladať informácie logicky, predsa len mali otázky, prípadne ak by ste objavili nejakú chybu, alebo by vám nejaké informácie v tejto učebnici chýbali, neváhajte ma kontaktovať na jana.jurinova@ucm.sk.

Rada by som na tomto mieste poďakovala doc. Ing. Michalovi Čerňanskému, PhD. a Mgr. Ing. Romanovi Horváthovi, PhD. za ich prínosné odborné komentáre, poznámky a poznatky pri recenzovaní tejto učebnice. Ich skúsenosti bezpochyby výrazne prispeli k skvalitneniu informácií v tejto učebnici.

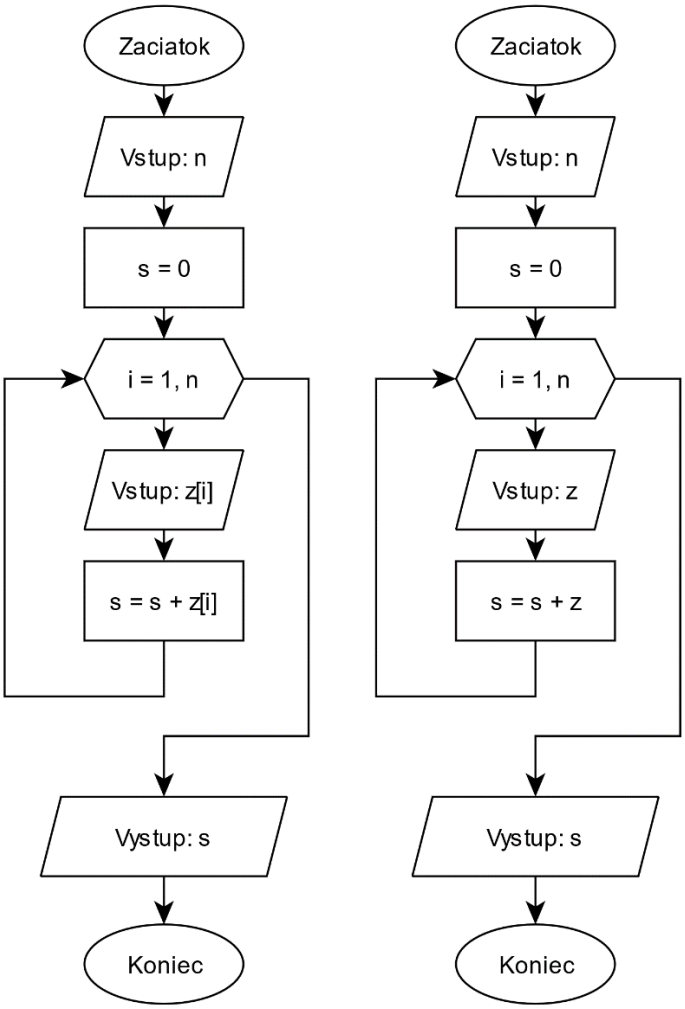
September 2021 Ing. Jana Jurinová, PhD., autorka učebnice

Obsah

1 Štruktúrované údajové typy: polia, štruktúry, uniony

Pre štruktúrované údajové typy, na rozdiel od základných (primitívnych) typov, ktoré boli objasňované v prvom dieli tejto učebnice, je typické, že sú zložené zo základných dátových typov alebo ich kombinácií. Môžu byť homogénne – všetky prvy sú rovnakého typu. Predstaviteľom homogenného dátoveho typu je pole. Alebo môžu byť heterogénne, napr. štruktúra a union, ktoré reprezentujú objekt zložený z dátových objektov rôzneho dátového typu. Sú reprezentované jediným identifikátorom (premennou), pričom je zabezpečený koncept ako pristupovať k jednotlivým členom štruktúrovaného údajového typu – bližšie špecifikované v tejto kapitole. Zoskupenie dát do logických celkov sprehľadnuje výsledný kód aj algoritmus.

Grafické reprezentácie algoritmov na obrázku 1 ilustrujú v prvom prípade čítanie súboru dát so zadaným počtom a súčasným vytváraním súčtu. Údaje zostanú zachované, keďže sa ukladajú do poľa. V druhom prípade ide o čítanie súboru dát so zadaným počtom a súčasným vytváraním súčtu, avšak údaje nezostanú zachované, keďže na ich načítavanie sa používa premenná jednoduchého dátového typu, ktorá vie uchovať len jednu hodnotu. V prvom príklade sa pracuje so štruktúrovaným údajovým typom – poľom, ktoré je v prípade implementácie – ilustrované obrázkom 2, deklarované na riadku 6 s veľkosťou určenou konštantou *MAX*. Na riadkoch 17 – 21 sa realizuje načítanie prvkov od používateľa a ich ukladanie do štruktúrovaného údajového typu. Pocou cyklu *for* na riadkoch 24 – 27 sa realizuje súčet týchto prvkov. Výsledok súčtu je vypísaný na riadku 30. V druhom prípade – ilustrované obrázkom 3, sa pracuje s premennou jednoduchého dátového typu, pri ktorej som použila rovnaký identifikátor ako v prvom príklade, t. j. premenú z, avšak táto premenná je typu float. Výsledok programu bude identický, avšak k zoznamu zadávaných prvkov používateľom sa už po načítaní prvkov nevieme dostať, na rozdiel od implementácie s poľom. Preto aj súčet prvkov je realizovaný ihneď po načítaní každého nového prvku. Ako môžeme vidieť z ukážok kódu, aj spôsob prístupu a manipulácie s prvkami je v prípade použitia poľa iný. K jednotlivým prvkom pristupujeme s pomocou indexu čo môžeme vidieť na obrázku 2, riadok 20, resp. riadok 26.



Obr. 1 Vývojové diagramy na realizáciu súčtu zo zdrojových dát.

```c
#include <stdio.h>
#define MAX 100

int main(void)
{
    float z[MAX], s = 0;
    int n, i;
    // nacitanie poctu prvkov pola
    do
    {
        printf("Zadaj pocet prvkov pola: \n");
        scanf("%d", &n);
    }
    while (n <= 0 || n > MAX);

    // nacitanie prvkov pola
    for (i = 0; i < n; i++)
    {
        printf("\nZadaj %d. cislo: \n", i+1);
        scanf("%f", &z[i]);
    }

```

```
23      // realizacia suctu prvkov pola
24      for (i = 0; i < n; i++)
25      {
26          s = s + z[i];
27      }
28
29      // vypis vysledku
30      printf("\nSucet prvok zo zadaneho pola je %f.\n", s);
31
32      return 0;
33  }
```

Obr. 2 Kód na realizáciu súčtu zo zdrojových dát ukladaných do poľa.

```
1   #include <stdio.h>
2   #define MAX 100
3
4   int main(void)
5   {
6       int n, i;
7       float z, s = 0;
8
9       do
10      {
11          printf("Zadaj pocet prvkov ktore chces spocitavat: \n");
12          scanf("%d", &n);
13      }
14      while (n <= 0);
15
16      // nacitanie prvkov od pouzivatela a realizacia suctu
17      for (i = 0; i < n; i++)
18      {
19          printf("\nZadaj %d. cislo: \n", i+1);
20          scanf("%f", &z);
21          s = s + z;
22      }
23
24      // vypis vysledku
25      printf("\nSucet prvok zo zadaneho pola je %f.\n", s);
26
27      return 0;
28  }
```

Obr. 3 Kód na realizáciu súčtu zo zdrojových dát reprezentovaných s pomocou premennej jednoduchého dátového typu.

Polia sú veľmi často používaný typ premennej v každom programovacom jazyku. Pole je typ, ktorý umožní zozkupiť viac premenných rovnakého typu do jednej premennej, pričom každému prvku je priradené číslo, tzv. index.

1.1 Jednorozmerné pole

Herout (2010) definuje jednorozmerné pole ako štruktúrovaný dátový typ zložený z prvkov rovnakého typu, prístupný pod spoločným identifikátorom, tzv. homogénny štruktúrovaný dátový typ. Prvky poľa sú vždy rovnakého typu, pričom typ môže byť jedným z jednoduchých (*int*, *char*, *float*, *double*), ale aj štruktúrovaných (štruktúra, union, pole a pod.). Pozíciu položky v poli určuje index. Prvky poľa sa v jazyku *C* vždy indexujú od 0. Indexy poľa sa nikdy automaticky v jazyku *C* nekontrolujú. Názov poľa je zároveň smerníkom (ukazovateľom) na prvý prvok poľa. **Syntax:**

```
TYP pole[pocet];        // pole je pocet-prvkové pole typu TYP
```

príklad staticky alokuje blok pamäte pre *pocet* prvkov typu *TYP*, pričom rozsah indexov je od *0* po *pocet − 1*. Ak predpokladáme, že hodnota *pocet* je známa v čase prekladu, tak by to mal byť konštantný výraz – často sa používa symbolická konštanta (ilustratívne vysvetlenie nižšie). Ak je pole definované týmto spôsobom ide o statické pole.

Avšak hodnota *pocet* nemusí byť vždy známa v čase prekladu, t. j. v prípade ak *pocet* nie je konštatný výraz, ale premenná. Vtedy ide o tzv. *variable lenght array*, kde sa pamäť poľa alokuje s veľkosťou podľa hodnoty premennej *pocet*, avšak až počas spustenia programu.

1.1.1 Definícia a inicializácia poľa

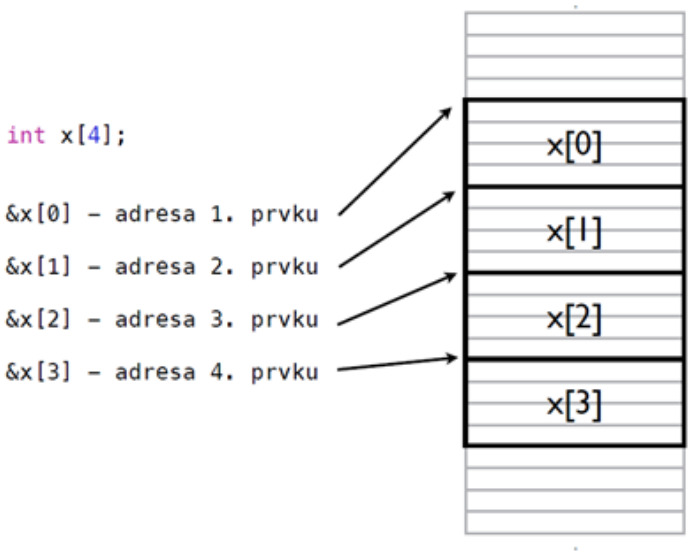
Pole možno inicializovať pri definícii. Nie je to priradenie hodnoty! Definícia inicializovaného poľa nemusí obsahovať veľkosť poľa (prekladač ho dopočíta z inicializácie). Pri rozdielnom počte inicializátorov a veľkosti poľa platí:

- ak veľkosť poľa > počet prvkov : inicializuje sa len začiatok, zvyšok ostáva nedefinovaný,
- ak veľkosť poľa < počet prvkov : chyba (príliš veľa inicializátorov).

Príklad:

```
int pole1[5] = { 2, 4, 6, 8, 10 };     // inicializovane 5 prvkove pole
int pole2[] = { 2, 4, 6, 8, 10 };      // inicializovane 5 prvkove pole
int pole3[5] = { 0 };                  // inicializovane 5 prvkove pole, prvky su 0
int pole4[10] = { 2, 4, 6, 8, 10 };    // poslednych 5 miest sa inicalizuje na 0
int pole5[4] = { 2, 4, 6, 8, 10 };     // chyba, nezmesti sa to tam
float pole6[5] = { 1.2, 2.5, 8.4, 6.0, 4.7 };
double pole7[] = { 1.2, 2.5, 8.4, 6.0, 4.7 };
```

Jednorozmerné pole sa dá inak nazvať vektor (*premenna(a1, a2, a3…)*). Graficky môžeme pole v pamäti reprezentovať:



Obr. 4 Jednorozmerné pole v pamäti.

Skutočnú veľkosť, ktorú zaberá pole v pamäti vieme zistiť v jazyku *C s* pomocou operátora *sizeof()*, ktorý vracia veľkosť v bajtoch. V tomto prípade sa *sizeof(x)* vyhodnotí na **16 B**.

Príklady použitia:

```
int x[10];        // deklaracia pola, pocet prvkov pola = velkost pola
x[0] = 5;         // prvy prvok pola - index 0 - zapisanie hodnoty 5 do prveho prvku
x[9] = 7;         // zapisanie hodnoty 7 do posledneho prvku
x[10] = 1;        // chyba - 10 nie je index posledneho prvku pola - zapis mimo hranic pola
x[i + j * 10]     // index prvku moze byt vyraz typu integer
#define MAX 10   // definicia konstanty, tzv. symbolickeho makra bez parametrov
int x[MAX], y[MAX * 2], z[MAX + 1];    // deklaracia pola s vyuzitim konstanty
```

Jednorozmerné statické pole môžeme inicializovať viacerými spôsobmi:

- Inicializáciou jednotlivých prvkov poľa
- Inicializačným zoznamom – v deklaračnom príkaze môžeme vynechať špecifikáciu rozsahu poľa
- Cyklom
- Inicializáciu poľa môžeme prenechať používateľovi programu

```
#include <stdio.h>
int main(void)
{
  int pole[5];
  pole[0] = 1;
  pole[1] = 2;
  pole[2] = 3;
  pole[3] = 4;
  pole[4] = 5;
  return 0;
}
```

```
#include <stdio.h>
int main(void)
{
  int pole[5] = {1, 2, 3, 4, 5};
  return 0;
}
```

```
#include <stdio.h>
int main(void)
{
  int pole[5];
  int i;
  for (i = 0; i <= 4; i++)
    pole[i] = i+1;
  return 0;
}
```

```
#include <stdio.h>
int main(void)
{
  int i, j, pole[5];
  for (i = 0; i <= 4; ++i)
   {
   printf("Zadajte %d. cislo: ", i+1);
   scanf("%d", &pole[i]);
   }
return 0;
}
```

1.1.2 Príklad – hľadanie minimálneho prvku v poli

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte riešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém nájdenia extrému (vyhľadanie minimálneho prvku) v poli. Veľkosť poľa vyžadujte od používateľa.

Rozbor úlohy: V prvom rade je dôležité poznať veľkosť poľa. Podmienkou úlohy je, aby túto zadal používateľ. Keďže veľkosť poľa musí byť kladné celé číslo väčšie ako 1, je dôležité správne určiť vstupnú podmienku a pri implementácii na túto nezabudnúť. Následne je možné načítať prvky poľa od používateľa s využitím cyklu s presne definovaným počtom opakovaní, t. j. telo cyklu sa zopakuje presne *velkost*-krát. Tak ako pri každom probléme aj tu existuje viacero spôsobov riešenia. Ak by sme zobrali do úvahy, že prvky poľa budú len záporné čísla, tak by sme mohli na začiatku využiť predpoklad, že 0 je naše minimum a následne prejsť celým poľom s kontrolou, či sa tam nenachádza menšie číslo. V takomto prípade by sme však vzhľadom na definíciu problému nesplnili vlastnosť hromadnosti, pretože pre daný problém je nevyhnutné pracovať na celej množine *R*. Vtedy už predpoklad inicializácie na nulu nebude správny (ak by používateľ zadal všetky prvky poľa kladné, nikdy by sme korektné minimum nenašli) a preto je vhodné na začiatku predpokladať, že prvý prvok poľa je naše minimum. V takom prípade však už neprehľadávame celé pole, ale pole bez prvého prvku (inak by pri prvej iterácii cyklu boli prvky porovnávané samé so sebou – čo je zbytočné).

V prípade implementácie v jazyku *C* môžeme využiť ešte ďalší prístup, kde na inicializáciu minima využijeme hranice použitého dátového typu. Pri deklarácii poľa je vždy nevyhnutné určiť dátový typ, ktorý je nemenný počas vykonávania programu a podľa toho aký zvolíme, môžeme využiť v prípade použitia jazyka *C* konštanty *INT_MIN* alebo *INT_MAX*, definované v hlavičkovom súbore *limits.h* v prípade, ak prvky poľa sú typu *int*. Konštanty *FLT_MIN* alebo *FLT_MAX* použijeme v prípade, ak prvky poľa sú typu *float*, resp. *DBL_MIN* alebo *DBL_MAX*, ak prvky poľa sú typu *double*. Tieto konštanty sú definované v hlavičkovom súbore *float.h*.

Pozn. autora: Z pohľadu algoritmizácie riešenie s využitím hraníc dátového typu nie je vhodné. A to z toho dôvodu, že takýto algoritmus nemôžeme považovať za univerzálny.

Nespĺňa vlastnosť hromadnosti, keďže rozsah zdrojových dát je obmedzený. Okrem toho, v prípade reprezentácie algoritmu riešenia v grafickej podobe by sa viazali na vývojové prostredie, čo nie je vhodné, pretože by bola porušená vlastnosť elementárnosti.

Iný pohľad na riešenie problému je, že vôbec nie je nevyhnutné pracovať s hodnotu minima, ale s jeho pozíciou. Riešenie spočíva len v zmene myšlienky inicializácie minima na začiatku na hodnotu prvého prvku (čiže na hodnotu 0 pri implementácii, pretože prvky poľa sa indexujú vždy od 0), teda minimum predstavuje prvý prvok. Následne len prehľadávame zvyšok poľa (bez prvého prvku) a editujeme pozíciu v prípade, ak nájdeme prvok s menšou hodnotou. Tento prístup riešenia poskytuje dokonca vyššiu nosnú informáciu, pretože okrem hodnoty minima poznáme aj jeho pozíciu. V prípade ak by sme problém rozšírili o identifikáciu, resp. počet všetkých miním v poli, tento algoritmus by viedol k istej optimalizácii, pretože by nebolo už nevyhnutné po identifikovaní minimia prehľadávať celé pole, ale len tú časť poľa, ktorá sa nachádza za prvým výskytom minima.

Vstup (vstupné premenné):

n – číslo zadané používateľom, ktoré reprezentuje veľkosť poľa = počet prvkov

pole[n] – pole s veľkosťou n

Výstup (výstupné premenné):

min – minimum = najmenšia hodnota v poli

Vstupné podmienky:

$n \in N$

$pole[i] \in R$

Výstupné podmienky:

$min \in R \wedge min$ je najmenší prvok poľa *pole*

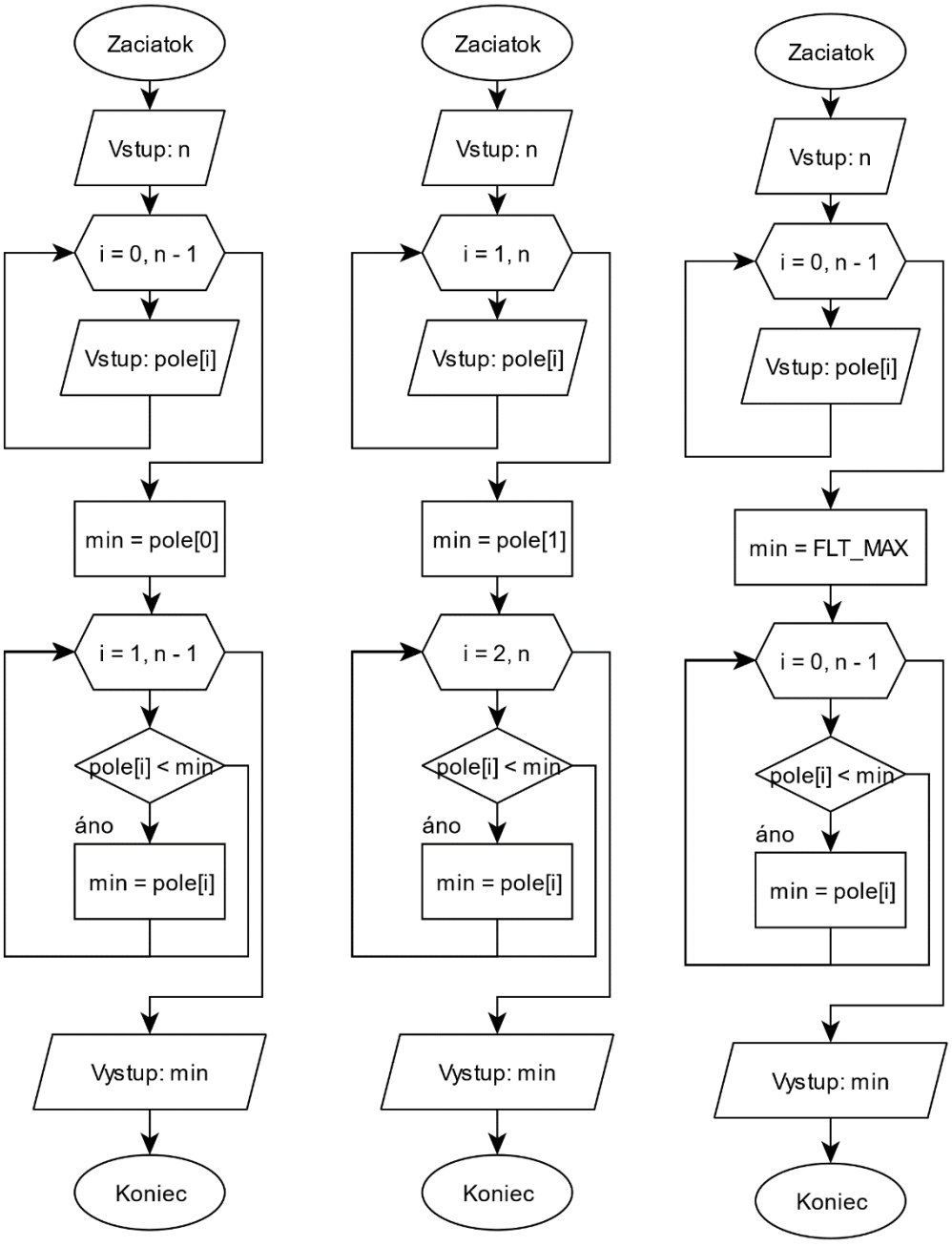
Pomocné premenné:

i – riadiaca premenná cyklu

$i \in \langle 0, n)$

Pozn. autora: Častou chybou študentov je, že pri definícii vstupnej podmienky, kedy sa snažia zapísať, že všetky prvky poľa sú definované na množine R, použijú zápis: $pole[n] \in R$. Tento zápis však nie úplne vystihuje podstatu, pretože ako vieme, pri implementácii každý prvok poľa je prístupný cez index, preto daný zápis možno chápať tak, že prvok poľa s indexom rovným hodnote premennej n je definovaný na množine R. Čo však potom ostatné prvky? Okrem toho,

v prípade jazyka *C*, vieme, že prvok s takýmto indexom ani nie je prvkom nášho poľa (sme mimo hraníc poľa), pretože posledný prvok je určený indexom $n - 1$. Táto dilema má elegantné riešenie, ktoré sme použili aj my. Premenná *i* je riadiaca premenná cyklu. Určením jej podmienky hovoríme, že nadobúda hodnoty daného intervalu, ktoré presne korešpondujú s indexmi prvkov poľa. Takýto zápis je jednoznačný a korektný.

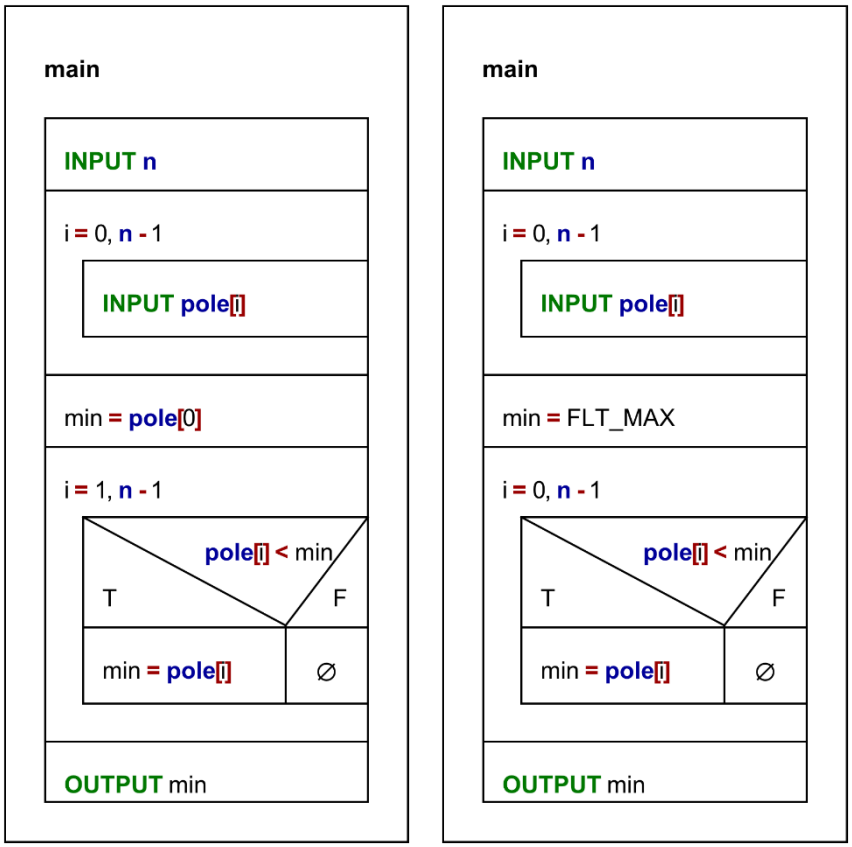


Obr. 5 Vývojové diagramy na riešenie problému hľadania minima v zadanom poli.

Pozn. autora: Prvé dva vývojové diagramy sú zakreslené absolútne korektne. Rozdiel je len v indexácii prvkov poľa, v prvom prípade pristupujeme k indexom prvkov poľa ako

programátori. To znamená, že prvý prvok leží na indexe 0, posledný $n - 1$. V druhom prípade indexujeme prvky od $1 \div n$ (vrátane) ako väčšina bežných používateľov. V prípade ak sa má implementácia algoritmu realizovať priamo v programovacom jazyku, odporúčame používať prvý spôsob, pretože pre študentov je to často podklad k riešeniu a môžu tak narobiť menej chýb a nemusia si držať v hlave, že v jazyku *C* sa prvky poľa indexujú od 0. Posledný vývojový diagram ilustruje, ako by vyzeralo riešenie s použitím symbolickej konštanty *FLT_MAX*. Ako sme uviedli toto riešenie nie je vhodné, porušuje vlastnosť hromadnosti a elementárnosti.

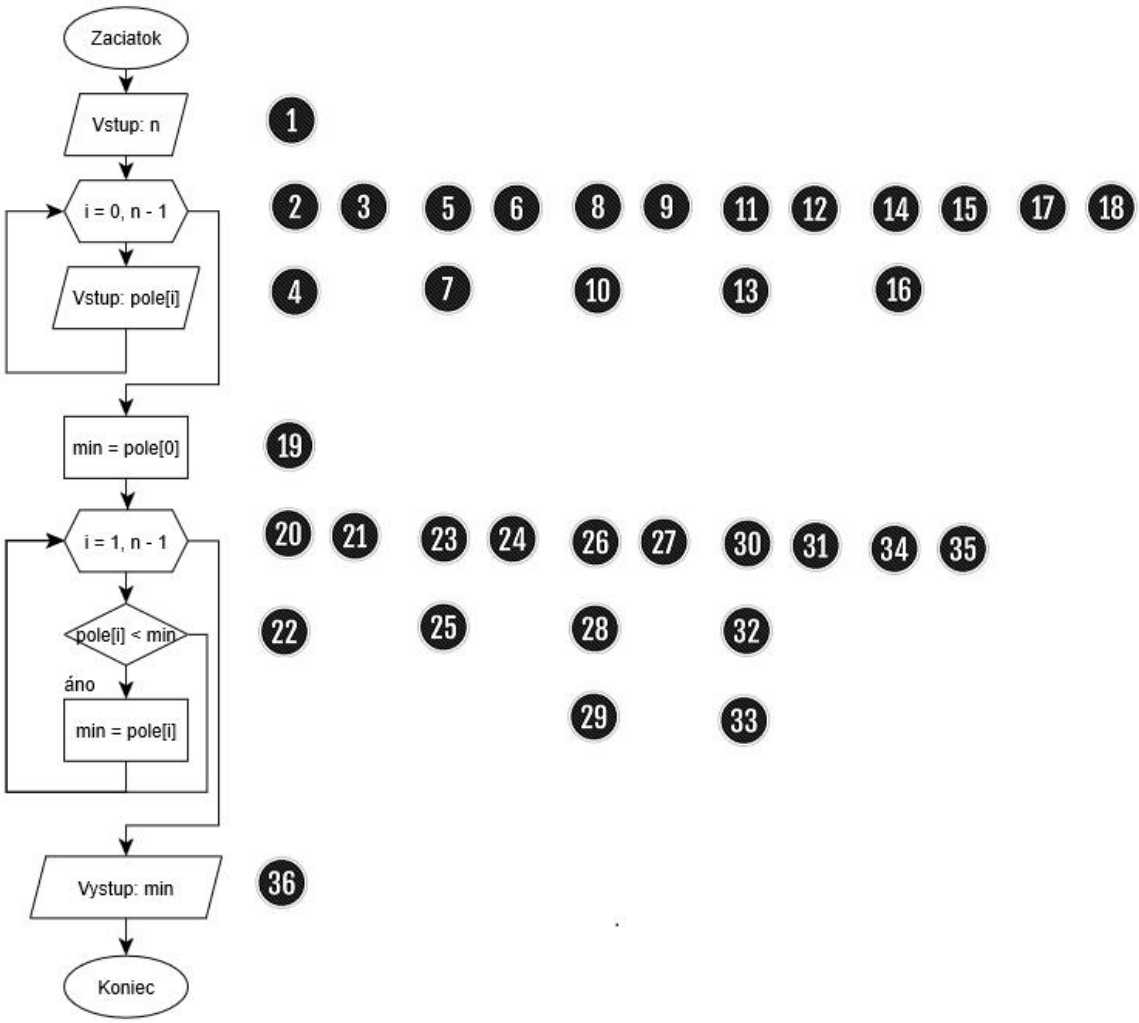
Zakreslenie s pomocou NS diagramu by vyzeralo:



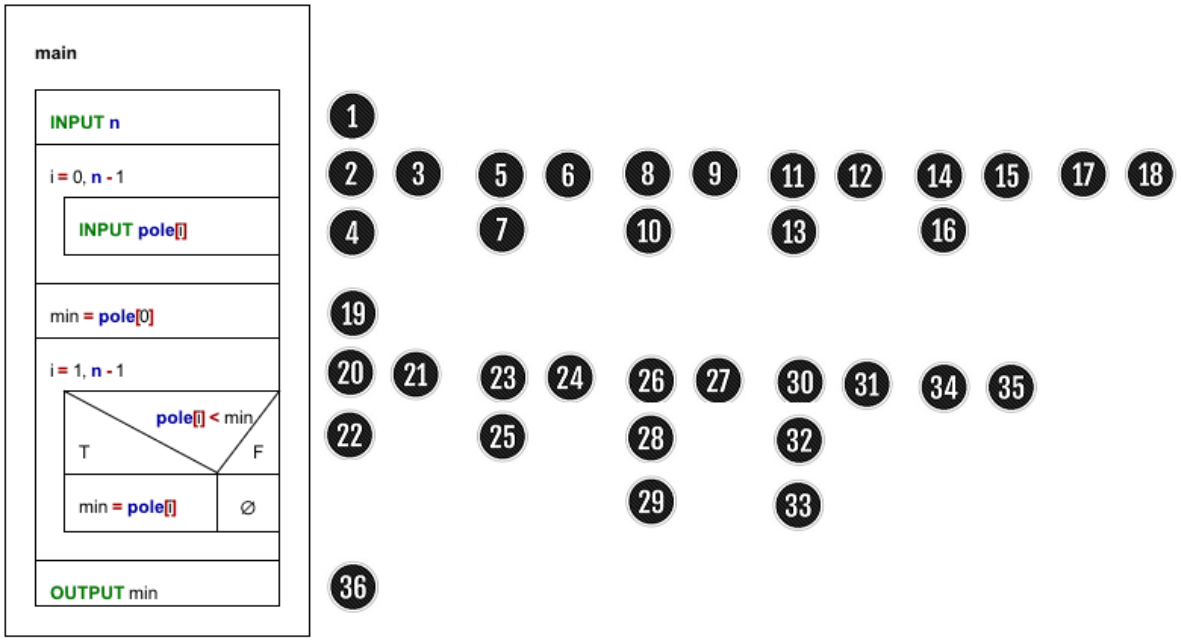
Obr. 6 NS diagramy na riešenie problému hľadania minima v zadanom poli.

Overenie správnosti algoritmu:

Pri jednotlivých akciách algoritmu reprezentovaného diagramom sú uvedené čísla krokov, ktoré korešpondujú s krokmi ilustrovanými v tabuľke.



Obr. 7 Vývojový diagram na riešenie problému hľadania minima v zadanom poli – krokovanie.



Obr. 8 NS diagram na riešenie problému hľadania minima v zadanom poli – krokovanie.

Realizácia algoritmu na príklade $n = 5$ a prvkoch poľa: $-5, 12, 0, -8$ *a* -9:

Krok	**Hodnoty sledovaných premenných**				**Popis jednotlivých krokov**
	n	**i**	**pole[i]**	**min**	
❶	5				Načítanie vstupnej premennej *n*
❷		0			Inicializácia riadiacej premennej cyklu $i = 0$
❸					Testovanie $i \leq n - 1$, 0 je menšie rovné ako 4 → vykonávanie tela cyklu
❹			−5		Načítanie prvého prvku poľa a jeho uloženie na index 0
❺		1			Inkrementácia riadiacej premennej *i*
❻					Testovanie $i \leq n - 1$, 1 je menšie rovné ako 4 → vykonávanie tela cyklu
❼			12		Načítanie druhého prvku poľa a jeho uloženie na index 1
❽		2			Inkrementácia riadiacej premennej *i*
❾					Testovanie $i \leq n - 1$, 2 je menšie rovné ako 4 → vykonávanie tela cyklu
❿			0		Načítanie tretieho prvku poľa a jeho uloženie na index 2
⓫		3			Inkrementácia riadiacej premennej *i*

⓬					Testovanie $i \leq n - 1$, 3 je menšie rovné ako 4 → vykonávanie tela cyklu
⓭			−8		Načítanie štvrtého prvku poľa a jeho uloženie na index 3
⓮		4			Inkrementácia riadiacej premennej *i*
⓯					Testovanie $i \leq n - 1$, 4 je menšie rovné ako 4 → vykonávanie tela cyklu
⓰			−9		Načítanie piateho prvku poľa a jeho uloženie na index 4
⓱		5			Inkrementácia riadiacej premennej *i*
⓲					Testovanie $i \leq n - 1$, 5 nie je menšie rovné ako 4 → koniec cyklu
⓳				−5	Uloženie prvého prvku poľa do premennej *min*
⓴		1			Inicializácia riadiacej premennej cyklu $i = 1$
㉑					Testovanie $i \leq n - 1$, 1 je menšie rovné ako 4 → vykonávanie tela cyklu
㉒					Porovnanie hodnoty poľa na indexe $i = 1$ a hodnoty v premennej *min* *pole[i]* < *min* → 12 nie je menšie ako −5
㉓		2			Inkrementácia riadiacej premennej *i*
㉔					Testovanie $i \leq n - 1$, 2 je menšie rovné ako 4 → vykonávanie tela cyklu
㉕					Porovnanie hodnoty poľa na indexe $i = 2$ a hodnoty v premennej *min.* *pole[i]* < *min* → 0 nie je menšie ako −5
㉖		3			Inkrementácia riadiacej premennej *i*
㉗					Testovanie $i \leq n - 1$, 3 je menšie rovné ako 4 → vykonávanie tela cyklu
㉘					Porovnanie hodnoty poľa na indexe $i = 3$ a hodnoty v premennej *min.* *pole[i]* < *min* → −8 je menšie ako −5
㉙				−8	Uloženie tejto hodnoty *pole[i]* do premennej *min* → *min = −8*
㉚		4			Inkrementácia riadiacej premennej *i*
㉛					Testovanie $i \leq n - 1$, 4 je menšie rovné ako 4 → vykonávanie tela cyklu

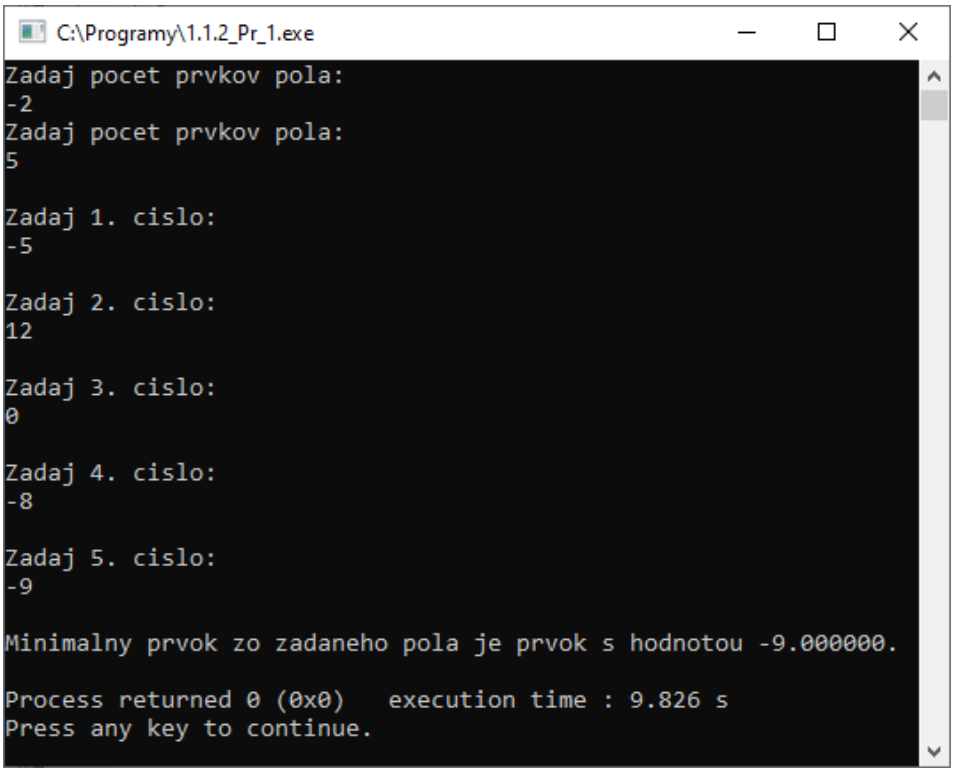
㉜					Porovnanie hodnoty poľa na indexe $i = 4$ a hodnoty v premennej *min*. *pole[i]* < *min* → −9 je menšie ako −8
㉝				−9	Uloženie tejto hodnoty *pole[i]* do premennej *min* → *min* = −9
㉞		5			Inkrementácia riadiacej premennej *i*
㉟					Testovanie $i \leq n - 1$, 5 nie je menšie rovné ako 4 → koniec cyklu
㊱					Výstup obsahu premennej *min* na obrazovku

Jedna z možných implementácií 1.1.2_Pr_1.c:

```c
#include <stdio.h>
#define MAX 100

int main(void)
{
    float pole[MAX], min;
    int n, i;
    // nacitanie poctu prvkov pola
    do
    {
        printf("Zadaj pocet prvkov pola: \n");
        scanf("%d", &n);
    }
    while (n <= 0 || n > MAX);

    // nacitanie prvkov pola
    for (i = 0; i < n; i++)
    {
        printf("\nZadaj %d. cislo: \n", i + 1);
        scanf("%f", &pole[i]);
    }
    // algoritmus hladania minima
    min = pole[0];

    for (i = 1; i < n; i++)
    {
        if (pole[i] < min)
        {
            min = pole[i];
        }
    }
    // vypis vysledku
    printf("\nMinimalny prvok zo zadaneho pola je prvok s hodnotou %f.\n", min);

    return 0;
}
```

Jedna z možných implementácií s využitím konštanty *FLT_MAX* 1.1.2_Pr_2.c:

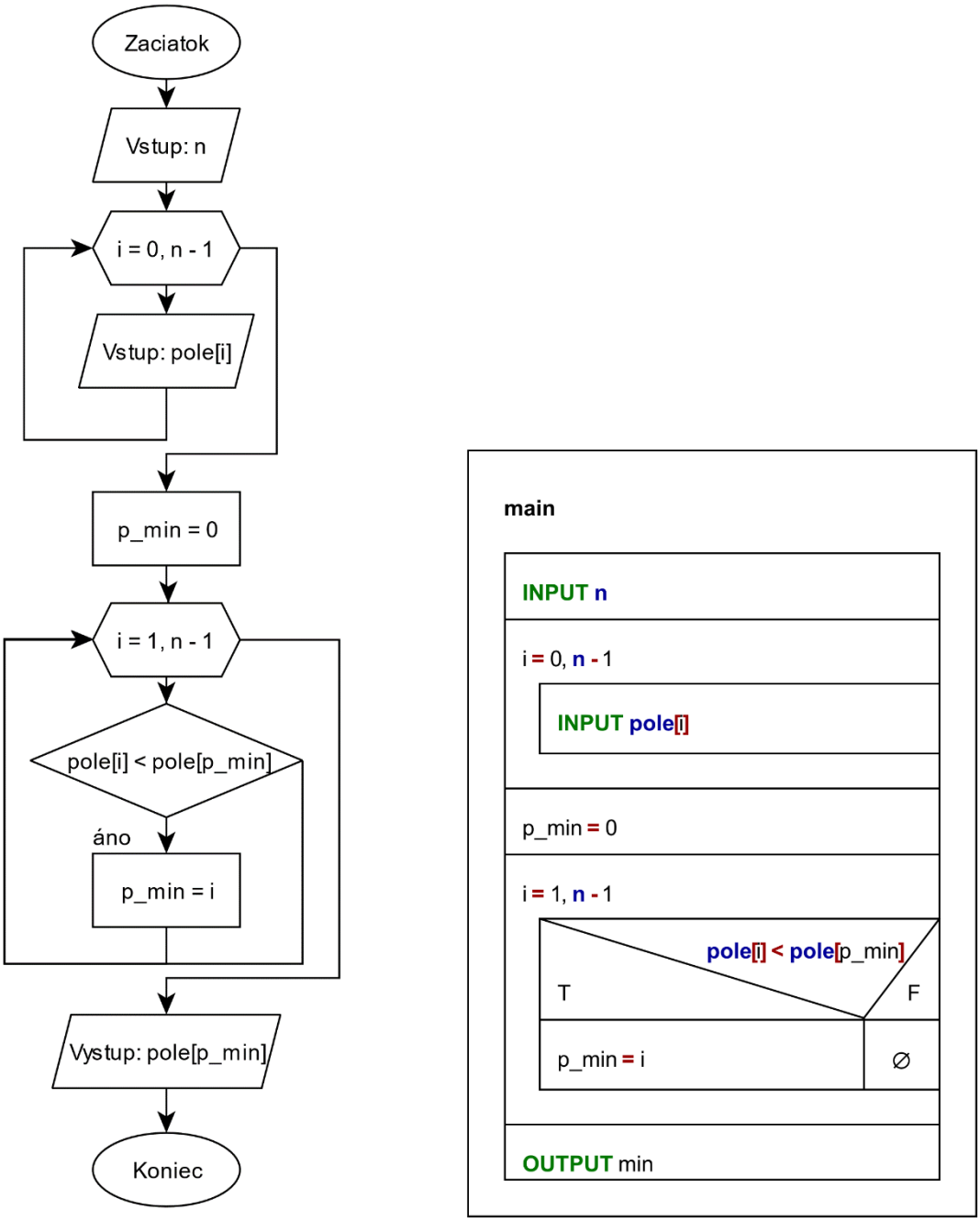
```c
#include <stdio.h>
#include <float.h>
#define MAX 100

int main(void)
{
    float pole[MAX], min = FLT_MAX;
    int n, i;
    // nacitanie poctu prvkov pola
    do
    {
        printf("Zadaj pocet prvkov pola: \n");
        scanf("%d", &n);
    }
    while (n <= 0 || n > MAX);

    // nacitanie prvkov pola
    for (i = 0; i < n; i++)
    {
        printf("Zadaj %d. cislo: \n", i + 1);
        scanf("%f", &pole[i]);
    }
    // algoritmus hladania minima
    for (i = 0; i < n; i++)
    {
        if (pole[i] < min)
        {
            min = pole[i];
        }
    }
    // vypis vysledku
    printf("\nMinimalny prvok zo zadaneho pola je prvok s hodnotou %f.\n", min);

    return 0;
}
```



Obr. 9 Konzolový výstup programu 1.1.2_Pr_1.c a 1.1.2_Pr_2.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 412 B/816 B = *n* (*int* = 4 B) + *pole[100]* (*float* = 4 B/*double* = 8 B . 100 = 400 B/800 B) + *i* (*int* = 4 B) + *min* (*float* = 4 B/*double* = 8 B).

Iný variant riešenia tohto problému by mohla byť na základe myšlienky, že nebudeme pracovať s hodnotou prvku, ale s jeho pozíciou v poli, tak ako sme naznačili vyššie. Využívame teda stále náš predpoklad, že na začiatku je prvý prvok poľa naše minimum, ale nezapamätávame si jeho hodnotu, ale jeho pozíciu:



Obr. 10 Vývojový a NS diagram na riešenie problému hľadania minima v zadanom poli na základe pozície.

Overenie správnosti algoritmu:

Stĺpce tabuľky reprezentujú používané premenné algoritmu. Jednotlivé riadky zachytávajú zmenu ich hodnôt v závislosti od vykonávania sa algoritmu. V stĺpci *Popis*. sa snažím zachytiť stručne slovne čo sa deje. Ďalej v učebnici budem používať túto zjednodušenú formu overovania správnosti algoritmu na rozdiel od detailného krokovania, ktoré som použila vyššie, resp. v prvom dieli učebnice.

Hodnoty sledovaných premenných				**Popis**
n	**i**	**pole[i]**	**p_min**	
5				Načítanie vstupnej premennej *n* – veľkosti poľa
	0	−5		*i = 0, 4* → postupné načítanie prvkov poľa a ich ukladanie do poľa na indexy 0 ÷ 4
	1	12		
	2	0		
	3	−8		
	4	−9		
	5			*i* = 5 sme mimo hraníc cyklu → koniec cyklu
			0	Inicializácia premennej *p_min*, ktorá predstavuje pozíciu nášho minima
	1			Inicializácia riadiacej premennej cyklu *i* = 1 a testovanie splnenia podmienky cyklu
				Porovnanie *pole[i] < pole[p_min]* → 12 nie je menšie ako −5
	2			Inkrementácia riadiacej premennej *i* a opätovne testovanie splnenia podmienky cyklu
				Porovnanie *pole[i] < pole[p_min]* → 0 nie je menšie ako −5
	3			Inkrementácia riadiacej premennej *i* a opätovne testovanie splnenia podmienky cyklu
				Porovnanie *pole[i] < pole[p_min]* → −8 je menšie ako −5
			3	Uloženie indexu *i*, ktorý predstavuje minimum do premennej *p_min*
	4			Inkrementácia riadiacej premennej *i* a opätovne testovanie splnenia podmienky cyklu
				Porovnanie *pole[i] < pole[p_min]* → −9 je menšie ako −8
			4	Uloženie indexu *i*, ktoré predstavuje minimum do premennej *p_min*

	5			Inkrementácia riadiacej premennej *i* a opätovne testovanie splnenia podmienky cyklu → i = 5 sme mimo hraníc cyklu → koniec cyklu
				Výstup obsahu premennej *pole[p_min]* = −9

Jedna z možných implementácií 1.1.2_Pr_3.c:

```c
#include <stdio.h>
#define MAX 100 // maximalna velkost pola

int main(void)
{
    float pole[MAX];
    int n, i, p_min = 0;
    // nacitanie poctu prvkov pola
    do
    {
        printf("Zadaj pocet prvkov pola: \n");
        scanf("%d", &n);
    }
    while (n <= 0 || n > MAX);

    // nacitanie prvkov pola
    for (i = 0; i < n; i++)
    {
        printf("\nZadaj %d. cislo: \n", i + 1);
        scanf("%f", &pole[i]);
    }
    // algoritmus hladania minima
    for (i = 1; i < n; i++)
    {
        if (pole[i] < pole[p_min])
        {
            p_min = i;
        }
    }
    // vypis vysledku
    printf("\nMinimalny prvok zo zadaneho pola je prvok s hodnotou %f.\n", pole[p_min]);

    return 0;
}
```

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 412 B/812 B = *n* (*int* = 4 B) + *pole[100]* (*float* = 4 B/*double* = 8 B . 100 = 400 B/800 B) + *i* (*int* = 4 B) + *p_min* (*int* = 4 B).

Domnievam sa, že vlastnosti algoritmu boli v prvom diele tejto učebnice dostatočne objasnené, preto sa pri ich posudzovaní zameriam už len na fakty, ktoré sú iné, resp. nové v súvislosti s riešenými problémami.

Nutné vlastnosti algoritmu:

- **Determinovanosť –** splnená.
- **Konečnosť –** každý algoritmus musí skončiť po vykonaní konečného počtu krokov. Riešenie pozostáva v podstave len zo sekvencie, cyklu *for* na načítanie prvkov poľa a cyklu *for* na hľadania extrému (minima), v ktorom je použitá podmienka na hľadanie tohto extrému. V tomto prípade je konečnosť načítavania prvkov poľa zabezpečená cyklom s pevným počtom opakovaní. Časť algoritmu, ktorá rieši hľadanie extrému (minima) je opätovne zabezpečená cyklom s pevným počtom opakovaní. Správne navrhnutou podmienkou hľadania extrému (minima) dostaneme vždy správny výsledok. Môžeme teda konštatovať, že konečnosť je splnená.

  Rezultatívnosť – splnená. Správnosť nájdenia extrému je zabezpečená správnou inicializáciou premennej *min*, resp. *p_min*, ako aj podmienkou jeho hľadania.

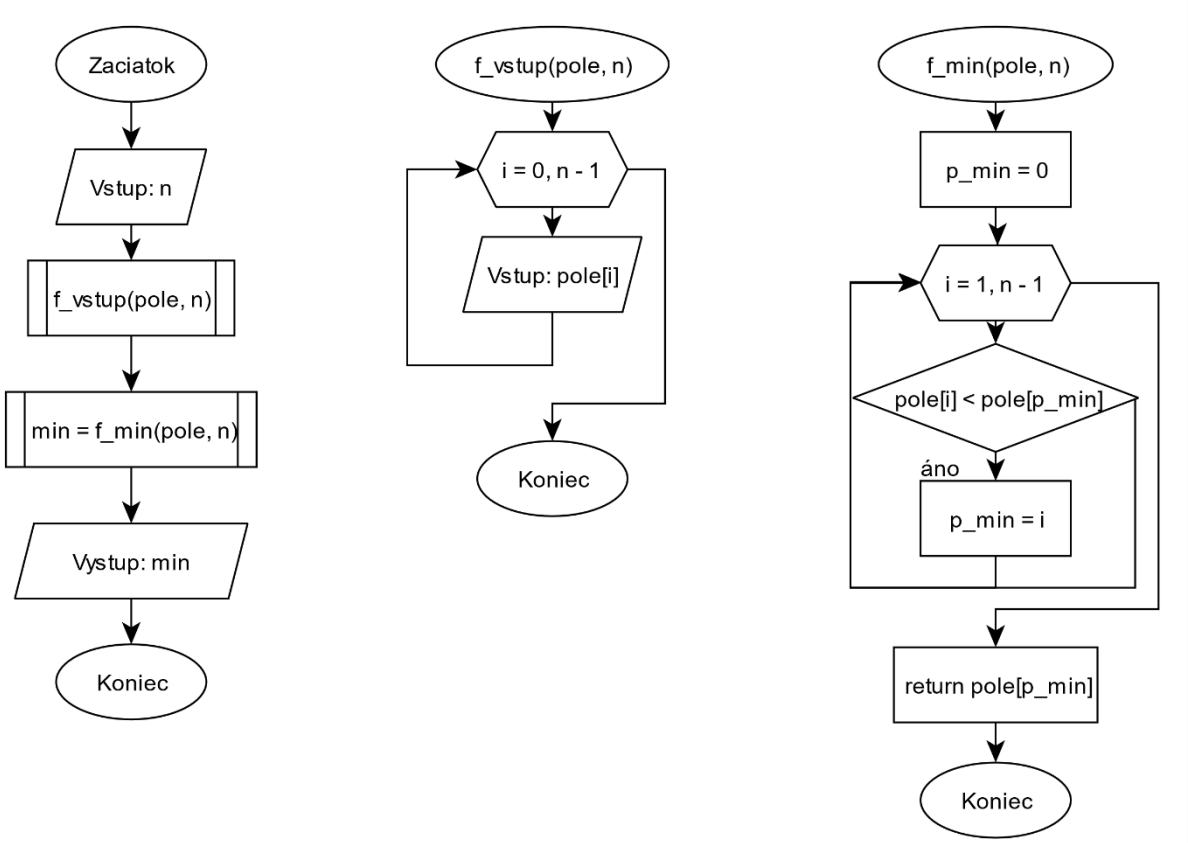
  Pozn. autora: Vzhľadom na to, že pracujeme so statickým poľom a aby sme neporušili pri implementácii zásady štruktúrovaného programovania, ktoré hovoria, že použité premenné by mali byť deklarované hneď na začiatku programu, zadefinovali sme si konštantu *MAX*, ktorá predstavuje maximálne možnú veľkosť poľa. Z tohto dôvodu je nevyhnutné pri načítavaní veľkosti poľa zobrať do úvahy aj túto skutočnosť. Inak by nastal zápis hodnôt mimo vyhradenej pamäte, dôsledkom čoho by sme nemuseli dostávať korektné hodnoty.

Odporúčané/očakávané vlastnosti algoritmu:

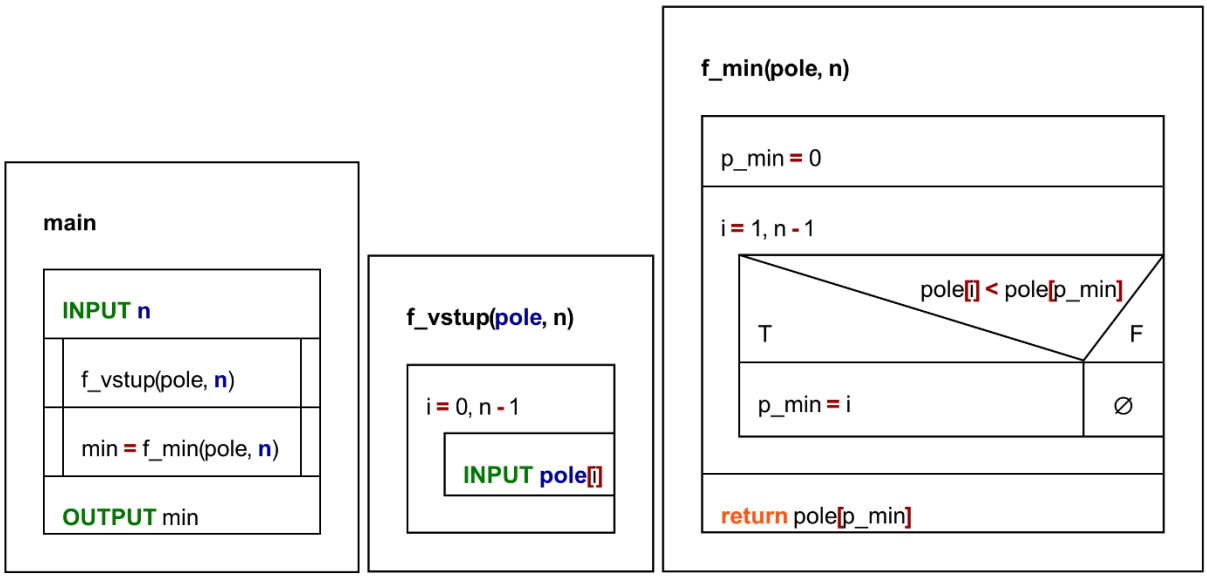
- **Hromadnosť** – splnená.
- **Elementárnosť** – splnená.
- **Efektívnosť** – môžeme konštatovať, že aj táto vlastnosť je splnená. Momentálne algoritmus vyžaduje použitie štyroch premenných. Priestorové nároky v závislosti od hodnôt vstupných premenných sú počas vykonávania algoritmu nemenné, z čoho vyplýva, že priestorová zložitosť je konštantná → $S(1)$.

  Časová zložitosť je lineárna → $O(n)$, pretože počet vykonaní príkazov tela cyklu tak pri načítavaní prvkov poľa, ako aj pri hľadaní minima je závislý od hodnoty počtu prvkov poľa n. Počet iterácií cyklu pri načítavaní prvkov poľa je teda n-krát. Pri hľadaní minima je počet iterácii cyklu $n - 1$-krát, vzhľadom na predpoklad, že prvý prvok poľa je naše minimum. Je teda zrejmý vzťah medzi nami definovaným vstupom a určenou časovou zložitosťou.

Pri posudzovaní efektívnosti by sme sa mali pozrieť aj na vlastnosti modifikovateľnosti a štruktúrovanosti, ktoré hovoria o tom, aby bol algoritmus navrhnutý tak, aby poskytol možnosť jednoduchej zmeny (úpravy) toku údajov, aj samotného algoritmu a aby pozostával zo samostatných, ale logicky previazaných celkov. Aby nastalo úplné naplnenie tejto vlastnosti, mali by sme kľúčové činnosti realizovať cez funkcie. V takomto prípade by reprezentácia riešenia mohla vyzerať takto:



Obr. 11 Vývojový diagram na riešenie problému hľadania minima v zadanom poli na základe pozície s funkciami.



Obr. 12 NS diagram na riešenie problému hľadania minima v zadanom poli na základe pozície s funkciami.

Pozn. autora: Treba si uvedomiť, že v prípade reprezentácie algoritmu graficky, nie je možné zachytiť pri používaní funkcií také skutočnosti, ako je odovzdávanie skutočných parametrov hodnotou alebo odkazom. V tomto prípade je parameter *pole* odovzdaný odkazom a parameter *n* hodnotou. Viac v kapitolách 2 a 4.

V prípade, ak nastane načítanie prvkov poľa s pomocou funkcie a hľadaniu minima tiež s pomocou funkcie, tak priestorové aj časové nároky by sme mali vyjadriť trochu inak:

Priestorová zložitosť	main()	$S(1)$ = konštantná Používajú sa tri premenné: *n*, *pole[100]* a *min*.
	f_vstup()	$S(1)$ = konštantná Funkcia má dva parametre *pole* a *n* a jednu lokálnu premennú *i*.
	f_min()	$S(1)$ = konštantná Funkcia má dva parametre *pole* a *n* a dve lokálne premennú *p_min* a *i*.
Časová zložitosť	main()	$O(1)$ = konštantná
	f_vstup()	$O(n)$ = lineárna
	f_min()	$O(n)$ = lineárna

Jedna z možných implementácií 1.1.2_Pr_4.c:

```c
#include <stdio.h>
#define MAX 100 // maximalna velkost pola

// deklaracie funkcii
void f_vstup(float *pole, int n);
int f_min(float *pole, int n);

int main(void)
{
    float pole[MAX];
    int n, p_min;
    // nacitanie poctu prvkov pola
    do
    {
        printf("Zadaj pocet prvkov pola: \n");
        scanf("%d", &n);
    }
    while (n <= 0 || n > MAX);

    // volanie funkcie na nacitanie prvkov pola
    f_vstup(pole,n);

    // vypis vysledku s pomocou funkcie
    printf("\nMinimalny prvok je prvok s hodnotou %f.\n", pole[f_min(pole,n)]);

    return 0;
}

// definicie funkcii
void f_vstup(float *pole, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        printf("\nZadaj %d. cislo: \n", i + 1);
        scanf("%f", &pole[i]);
    }
}

int f_min(float *pole, int n)
{
    int i, p_min = 0;
    for (i = 1; i < n; i++)
    {
        if (pole[i] < pole[p_min])
        {
            p_min = i;
        }
    }
    return p_min;
}
```

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných a vziať do úvahy aj použitie funkcií, tak by sme tieto mali vyjadriť trochu inak:

Priestorové nároky	main()	*n* (*int* = 4 B) + *pole[100]* (*float* = 4 B/*double* = 8 B . 100 = 400 B/800 B)) + *min* (*float* = 4 B/*double* = 8 B) = **408/812 B**
	f_vstup()	*pole* (* *float* = 4 B/8 B/* *double* = 4 B/8 B) + *n* (*int* = 4 B) + *i* (*int* = 4 B) = **12/16 B**
	f_min()	*pole* (* *float* = 4 B/8 B/* *double* = 4 B/8 B) + *n* (*int* = 4 B) + *p_min* (*float* = 4 B/*double* = 8 B) + *i* (*int* = 4 B) = **16/24 B**

Treba si uvedomiť, že priestorové nároky funkcií *f_vstup()* a *f_min()* vznikajú, keď nastane ich vykonávanie a zanikajú v momente, keď nastane ich ukončenie. Priestorové nároky funkcie *main()* sú trvalé od začiatku až po koniec programu.

Pozn. autora: Premenná typu pointer zaberá buď 4/8 B v závislosti od operačného systému, či je 32-bitový, alebo 64-bitový. Je vhodné si všimnúť, že pri určovaní priestorovej zložitosti pri funkciách nie je pamäť pre celé pole opätovne alokovaná. Je to tak práve z dôvodu, že sa *pole* ako parameter odovzdáva odkazom.

1.2 Dvojrozmerné pole

Dvojrozmerné pole sa dá prirovnať k tabuľke alebo k matici. Tabuľka aj matica je špecifikovaná riadkami a stĺpcami. Preto dva rozmery, ktoré sa zadávajú pri deklarácii dvojrozmerného poľa, sa dajú interpretovať ako riadok a stĺpec. Syntax deklarácie dvojrozmerného poľa vyzerá takto:

```
TYP pole[riadky][stlpce];
```

Technicky je možná aj deklarácia:

```
TYP pole[stlpce][riadky];
```

Záleží od používateľa ako sa na pole pozrie. Odporúčame prvý spôsob, avšak v prípade ak, by používateľ zvolil druhý, odporúčame zachovávať rovnaký spôsob práce s maticami v celom algoritme/programe. Je viac než nevhodné meniť koncept v rámci jedného algoritmu/programu.

Inicializácia viacrozmerného poľa môže byť realizovaná podobne ako pri jednorozmernom poli. Riadky môžu/nemusia byť v samostatnej zátvorke. Nepovinný je len prvý rozmer. Druhý a ďalšie (v prípade viacrozmerných polí) musia mať určenú veľkosť. Príklady:

```
int matica[2][3] = { 5,2,7,8,6,3 };
int matica[2][3] = { {5,2,7}, {8,6,3} };
// pocet riadkov sa urci podla poctu inicializatorov
int matica[][3] = { {5,2,7}, {8,6,3}, {5,7,6}, {2,8,7} };
```

Dvojrozmerné statické pole môžeme inicializovať podobnými spôsobmi, akými sme inicializovali jednorozmerné statické pole:

- Inicializáciou jednotlivých prvkov poľa
- Inicializačným zoznamom
- Vnoreným cyklom

```
#include <stdio.h>
int main(void)
{
  int pole[2][3];
  pole[0][0] = 1;
  pole[0][1] = 2;
  pole[0][2] = 3;
  pole[1][0] = 4;
  pole[1][1] = 5;
  pole[1][2] = 6;
  return 0;
}
```

```
#include <stdio.h>
int main(void)
{
  int pole[2][2] = {
      {1, 2, 3},
      {4, 5, 6} };
  return 0;
}
```

```
#include <stdio.h>
int main(void)
{
  int pole[2][3];
  int i, j, x = 1;
  for (i = 0; i < 2; i++) {
   for (j = 0; j < 3; j++){
       pole[i][j] = x++;
   }
}
  return 0;
}
```

Pozn. autora: V prípade načítavania prvkov s pomocou vnoreného cyklu si môžeme všimnúť, že prvky matice sa načítavajú po riadkoch. Vhodnou zmenou je možné realizovať načítavanie aj po stĺpcoch. Ako som uviedla vyššie, technicky to možné je, ale nie je to konvenčné riešenie. Ak sa však autor tak rozhodne, treba dodržiavať túto konvenciu v celom algoritme/programe, t. j. nemiešať pri načítavaní prvkov matice prístup, že raz sa budú načítavať po riadkoch a druhýkrát po stĺpcoch.

1.2.1 Príklad – súčet dvoch matíc

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte rešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém súčtu dvoch matíc.

Pozn. autora: Často sa študenti pri takto formulovanom zadaní pýtajú, či majú rozmery aj prvky matice ponechať na zadanie od používateľa. Jednoznačná odpoveď je áno. Následne sa pýtajú, prečo to nie je zadané. Odpoveď je, že to nie je nevyhnutné. Ak chceme splniť vlastnosť hromadnosti, tak riešenie ktoré by realizovalo súčet matíc pevných rozmerov, prípadne aj konkrétnych matíc, by túto vlastnosť nesplnilo.

Vstup (vstupné premenné):

m, n – rozmery matice A a matice B, počet riadkov a stĺpcov

$A[m][n]$ – vstupná matica A

$B[m][n]$ – vstupná matica B

Výstup (výstupné premenné):

$C[m][n]$ – výstupná matica daná súčtom matice A a matice B

Vstupné podmienky:

$m, n \in \{2, 3, 4 \dots MAX\} \wedge A[i][j] \in \mathrm{R} \wedge B[i][j] \in \mathrm{R}$

Výstupné podmienky:

$C[i][j] \in \mathrm{R} \wedge C[i][j] = A[i][j] + B[i][j]$

Pomocné premenné:

i – riadiaca premenná, $i \in \langle 0, m)$

j – riadiaca premenná, $j \in \langle 0, n)$

MAX – konštanta na určenie maximálnej veľkosti matice (pre úplnosť uvádzame aj túto premennú, je však zrejmé, že z pohľadu algoritmizácie nie je toto vôbec potrebné). Pri implementácii sa jej použitiu vieme tiež vyhnúť. V takom prípade by sme však pri implementácii museli využívať dynamické pole, ktoré bude vysvetlené až v kapitole 4.6.

Pozn. autora: Vzhľadom na to, že problém ktorý riešime je súčet matíc a platí, že vieme sčítať len matice rovnakých rozmerov, stačia nám len dve premenné, ktoré reprezentujú tieto rozmery. Často si študenti túto skutočnosť neuvedomia a definujú štyri (v horšom prípade šesť) premenné reprezentujúce rozmery daných matíc, s ktorými pracujú. Skutočnosť, že si neuvedomia, že súčet matíc je definovaný len pre matice rovnakých rozmerov a že rozmery výslednej matice môžu odvodiť od rozmerov vstupných matíc, je dôležitá. Zbytočne tak navyšujú priestorovú zložitosť riešenia a okrem toho často zabúdajú na definovanie podmienok, bez ktorých by riešenie neposkytovalo korektné výsledky, teda na fakt, že rozmery vstupných

matíc musia byť rovnaké ($r1 = r2 \wedge s1 = s2$), v prípade ak by sme definovali zvlásť premenné pre rozmery vstupných matíc. Rozmery matíc sme definovali na množine čísiel {2, 3, 4 … *MAX*} z toho dôvodu, že matica o rozmere $m = 1$ a $n = 1$ je v podstate číslo. Matica ktorá má m riadkov a n stĺpcov a počet riadkov je rovný 1 je špeciálna matica, tzv. riadkový vektor, resp. ak počet stĺpcov je rovný 1 hovoríme o tzv. stĺpcovom vektore.

Situácia, kde by sme použili rôzne premenné na reprezentáciu rozmerov vstupných matíc nie je až taká kritická (chybná). Dôvodom je skutočnosť, že ak sa snažíme navrhnúť algoritmus tak, že rátame s jeho budúcim rozšírením, je tento prístup možno aj vhodnejší. Predstavte si, že by sme daný problém rozšírili aj o súčin matíc. V takom prípade platí, že vieme násobiť len také matice, kde je počet stĺpcov ľavej matice rovnaký ako počet riadkov pravej matice, t. j. rozmery vstupných matíc môžu byť rôzne. Tu je ilustrované, ako by vyzerala definícia vstupov a výstupov a ich podmienok za tohto predpokladu:

Vstup (vstupné premenné):

r1, s1, r2, s2 – rozmery matice *A* a matice *B*, počet riadkov a stĺpcov

A[r1][s1] – vstupná matica *A*

B[r2][s2] – vstupná matica *B*

Výstup (výstupné premenné):

C[r1][s1] – výstupná matica, daná súčtom matice *A* a matice *B*

Vstupné podmienky:

$r1, s1, r2, s2 \in \{2, 3, 4 \ldots MAX\} \wedge r1 = r2 \wedge s1 = s2 \wedge A[i][j] \in \mathrm{R} \wedge B[i][j] \in \mathrm{R}$

Výstupné podmienky:

$C[i][j] \in \mathrm{R} \wedge C[i][j] = A[i][j] + B[i][j]$

Pomocné premenné:

i – riadiaca premenná, $i \in \langle 0, r1)$ alebo $i \in \langle 0, r2)$

j – riadiaca premenná, $j \in \langle 0, s1)$ alebo $j \in \langle 0, s2)$

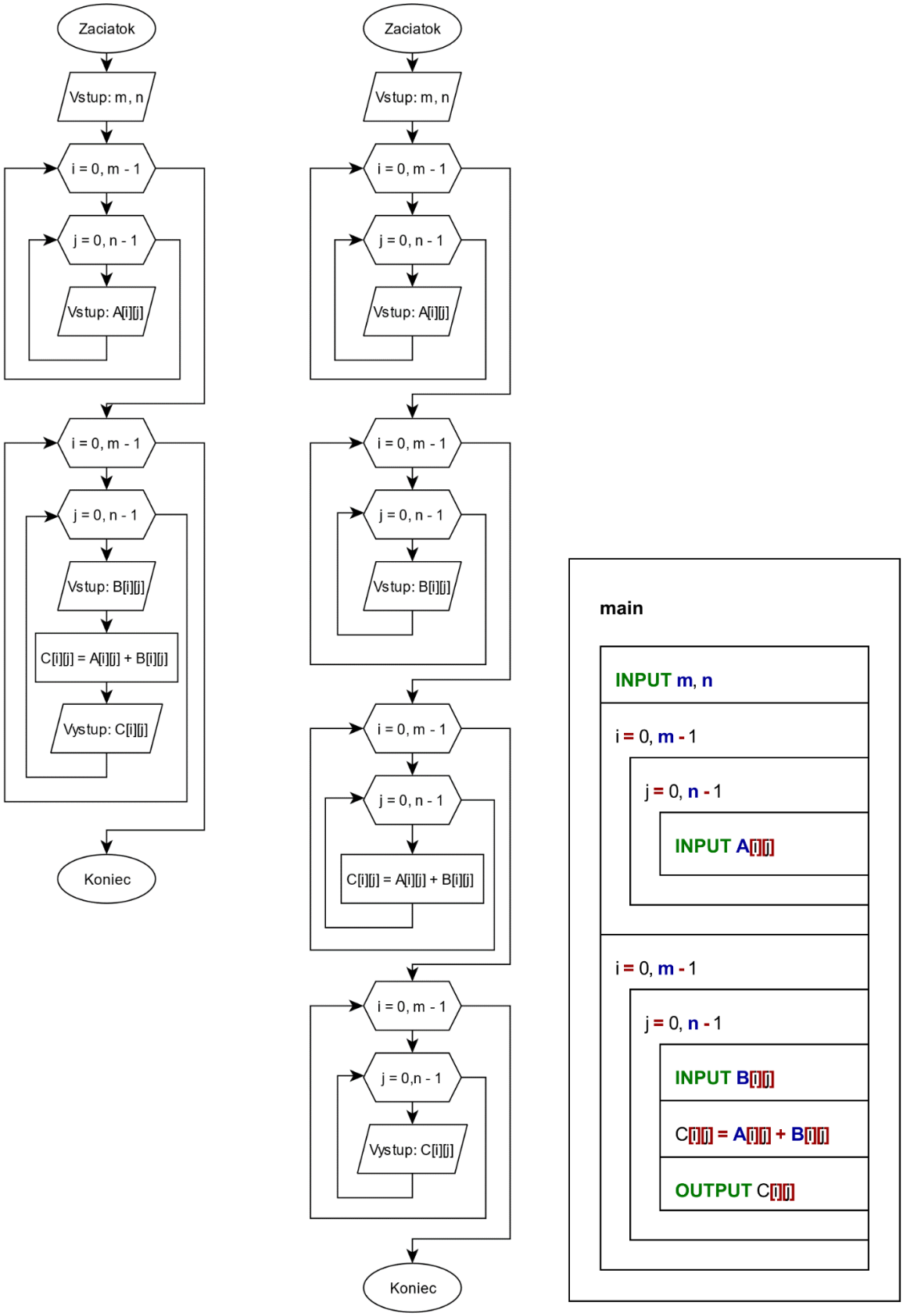
Rozbor úlohy: Maticu môžeme definovať ako určitú množinu čísel (tzv. prvkov matice) usporiadaných do pravidelných riadkov a stĺpcov vyznačujúcich sa tým, že každý výpočtový úkon vykonávaný s maticou sa dotýka každého prvku tvoriaceho maticu. Ak má matica m riadkov a n stĺpcov, hovoríme o matici typu m krát n. Prvky matice *A* zvyčajne označujeme ako a_{ij}, pričom *i* je číslo riadka a *j* stĺpca (Madaras, 2011). V prvom rade preto potrebujeme

poznať rozmery vstupných matíc, aby sme vedeli, koľko prvkov treba načítať. Sčítavanie matíc môže prebiehať len vtedy, ak tieto dve matice majú rovnaký rozmer. Sčítavajú sa čísla na rovnakých pozíciách. Napríklad:

$$\begin{bmatrix} 1 & 3 & 2 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 & 2+5 \\ 1+7 & 0+5 & 0+0 \\ 1+2 & 2+1 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 7 \\ 8 & 5 & 0 \\ 3 & 3 & 3 \end{bmatrix}$$

Je zrejmé, že nie je nevyhnutné definovať rozmery pre výstupnú maticu. Tie sú rovnaké ako majú vstupné matice. Na načítanie/výpis prvkov matice, aj na spracovanie prvkov matice je možné použiť vnorený cyklus s pevným počtom opakovaní. Vnorený cyklus potrebujeme z toho dôvodu, že prvky matice sú určené dvomi indexmi. Ako vieme každý *for*-cyklus je možné reprezentovať s pomocou cyklu s podmienkou na začiatku. Táto grafická reprezentácia však nebude taká prehľadná ako pri použití *for*-cyklov a často vedie študentov k neúmyselným chybám, ktoré však majú fatálne dôsledky. Bližšie objasníme na príklade.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 13 Vývojový a NS diagram na súčet matíc s pomocou *for*-cyklov.

Pozn. autora: Obidve riešenia zakreslené s pomocou vývojových diagramov na obrázku 13 sú správne. Priestorová zložitosť oboch riešení je rovnaká. Časová zložitosť tiež – je kvadratická $O(n^2)$. Z vizuálneho hľadiska sa zdá prívetivejšie prvé riešenie, pretože je kratšie, to však nemôže byť jediným meradlom. Ak sa zamyslíme nad procesom vykonávania algoritmu, tak druhé riešenie bude dvakrát viac inicializovať riadiace premenné *i* a *j*. Vyhodnocovanie podmienky vykonávania cyklu pri načítaní prvkov druhej matice pri spracovaní a tlači výstupnej matice sa tiež niekoľkokrát znásobí. Ako sme však uviedli, to nezmení rád rýchlosti rastu funkcie určujúcej časovú zložitosť, preto obe riešenia sú pre nás akceptovateľné. Na strane druhej, ak chceme splniť vlastnosť modifikovateľnosti do úplnej miery, riešenie by sme mali poskytnúť s pomocou funkcií, kde sa viacnásobnému použitiu vnorených cyklov nevyhneme. Tiež si treba uvedomiť, že nie všetky operácie, ktoré sa s maticami vykonávajú sa dajú realizovať takto, s pomocou jedného vnoreného cyklu. Ak pri súčte matíc poznáme hodnotu prvého prvku druhej matice, tak môžeme realizovať operáciu súčtu a výstup na obrazovku. Avšak napr. pri výpočte súčinu matíc by toto nebolo možné, pretože na výpočet prvého prvku výstupnej matice potrebujeme poznať všetky prvky prvého riadka vstupnej matice a všetky prvky prvého stĺpca druhej vstupnej matice.

Overenie správnosti algoritmu:

Realizácia algoritmu na príklade $m = 3$, $n = 3$ a prvkoch matíc, tak ako ilustruje obrázok:

$$\begin{bmatrix} 1 & 3 & 2 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 & 2+5 \\ 1+7 & 0+5 & 0+0 \\ 1+2 & 2+1 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 7 \\ 8 & 5 & 0 \\ 3 & 3 & 3 \end{bmatrix}$$

***m* = 3, *n* = 3**					
Hodnoty sledovaných premenných					**Popis**
i	**j**	**A[m][n]**	**B[m][n]**	**C[m][n]**	
					Načítanie veľkosti dvojrozmerného poľa = matice → *m* = 3 a *n* = 3
0					Inicializácia riadiacej premennej cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
		1			Načítanie prvku matice *A[0][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2

		3			Načítanie prvku matice *A[0][1]*
	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → *2 ≤ 2*
		2			Načítanie prvku matice *A[0][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu
1					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 1 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky vnútorného cyklu → 0 ≤ 2
		1			Načítanie prvku matice *A[1][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
		0			Načítanie prvku matice *A[1][1]*
	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
		0			Načítanie prvku matice *A[1][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu
2					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 2 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky vnútorného cyklu → 0 ≤ 2
		1			Načítanie prvku matice *A[2][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
		2			Načítanie prvku matice *A[2][1]*
	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
		2			Načítanie prvku matice *A[2][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu

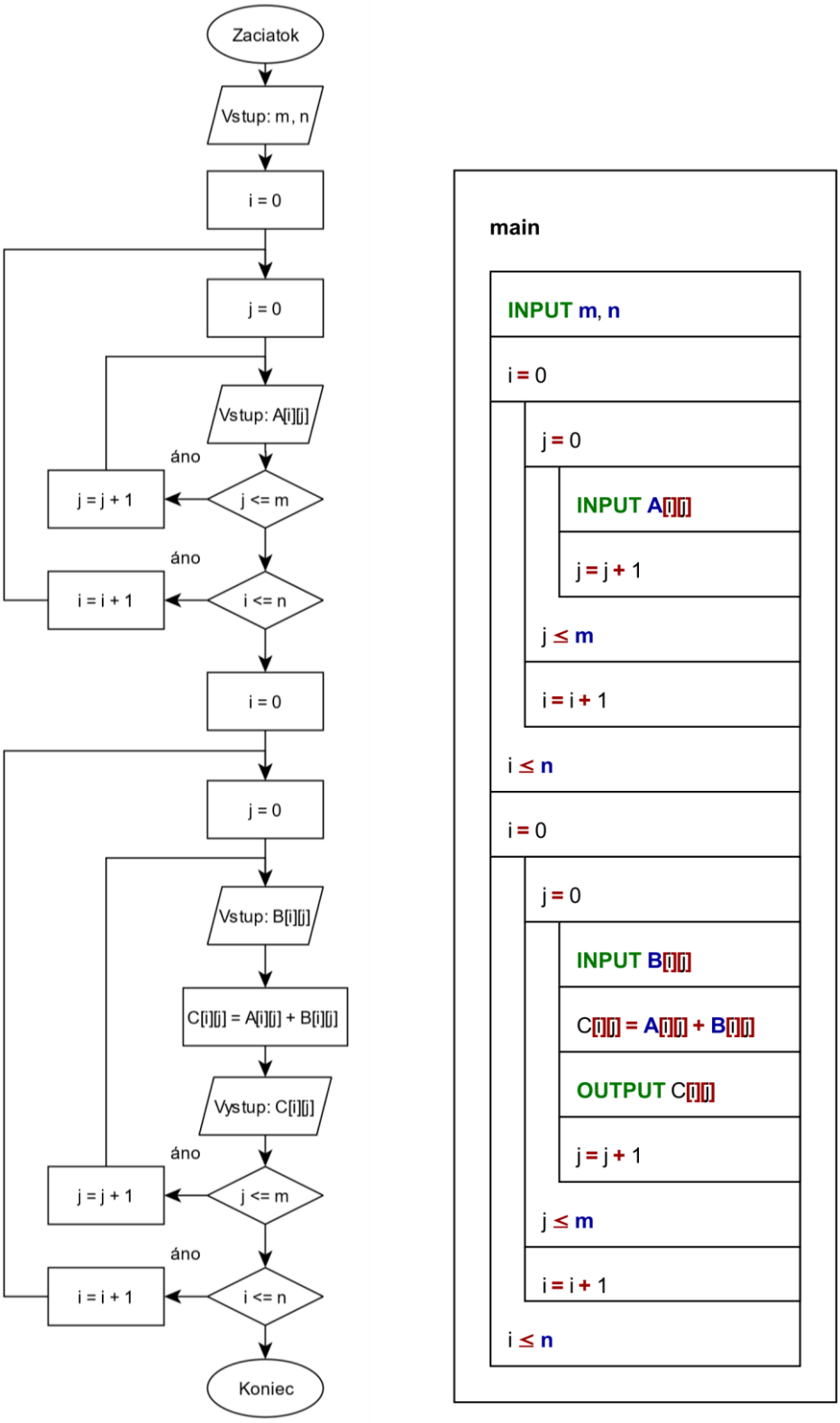
3					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 3 ≤ 2 → podmienka neplatí → koniec vonkajšieho cyklu
0					Inicializácia riadiacej premennej cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
			0		Načítanie prvku matice *B[0][0]*
				1	*C[0][0] = A[0][0] + B[0][0]* = 1 + 0 = 1
					Výstup na obrazovku prvku *C[0][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
			0		Načítanie prvku matice *B[0][1]*
				3	*C[0][1] = A[0][1] + B[0][1]* = 3 + 0 = 3
					Výstup na obrazovku prvku *C[0][1]*
	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
			5		Načítanie prvku matice *B[0][2]*
				7	*C[0][2] = A[0][2] + B[0][2]* = 2 + 5 = 7
					Výstup na obrazovku prvku *C[0][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu
1					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 1 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
			7		Načítanie prvku matice *B[1][0]*
				8	*C[1][0] = A[1][0] + B[1][0]* = 1 + 7 = 8
					Výstup na obrazovku prvku *C[1][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
			5		Načítanie prvku matice *B[1][1]*
				5	*C[1][1] = A[1][1] + B[1][1]* = 0 + 5 = 5
					Výstup na obrazovku prvku *C[1][1]*

	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
			0		Načítanie prvku matice *B[1][2]*
				0	*C[1][2] = A[1][2] + B[1][2]* = 0 + 0 = 0
					Výstup na obrazovku prvku *C[1][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu
2					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 2 ≤ 2
	0				Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
			2		Načítanie prvku matice *B[2][0]*
				3	*C[2][0] = A[2][0] + B[2][0]* = 1 + 2 = 3
					Výstup na obrazovku prvku *C[2][0]*
	1				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
			1		Načítanie prvku matice *B[2][1]*
				3	*C[2][1] = A[2][1] + B[2][1]* = 2 + 1 = 3
					Výstup na obrazovku prvku *C[2][1]*
	2				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
			1		Načítanie prvku matice *B[2][2]*
				3	*C[2][2] = A[2][2] + B[2][2]* = 2 + 1 = 3
					Výstup na obrazovku prvku *C[2][2]*
	3				Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vnútorného cyklu
3					Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → podmienka neplatí → koniec vonkajšieho cyklu (koniec algoritmu)

Detailné overenie správnosti druhej verzie algoritmu ilustrovaného s pomocou vývojového diagramu na obrázku 13 ponechávam na čitateľovi.

1.2.1.1 Riešenie s pomocou cyklov s podmienkou na konci

Riešenie toho istého algoritmu s pomocou cyklov s podmienkou na konci, namiesto použita cyklov s presne definovaným počtom opakovaní:



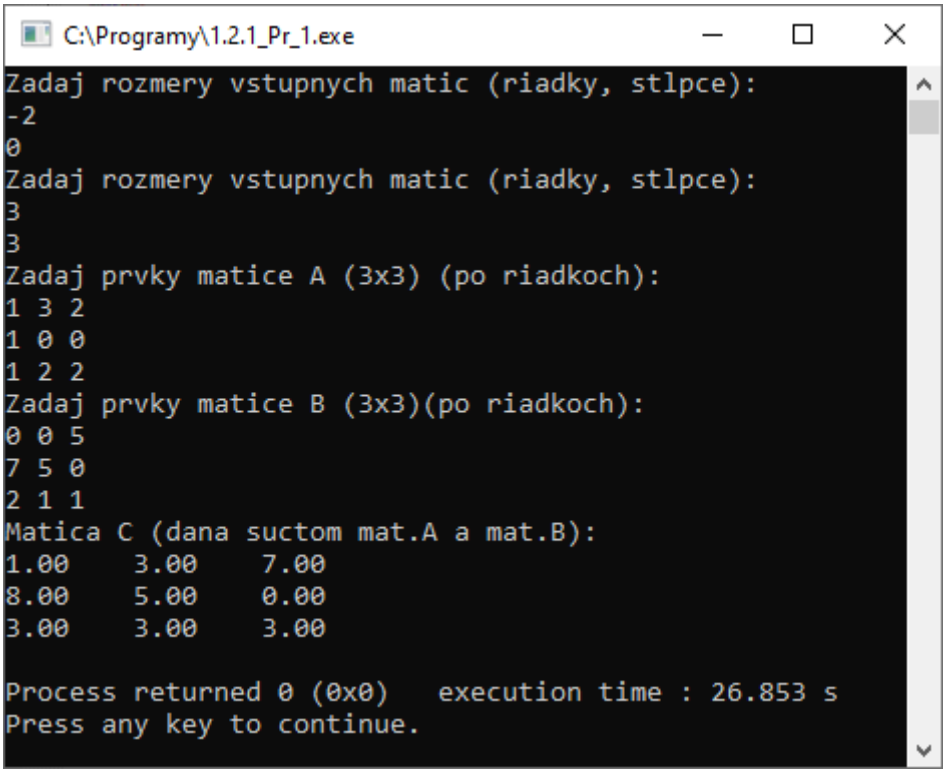
Obr. 14 Vývojový a NS diagram na súčet matíc s pomocou cyklov s podmienkou na konci.

Pozn. autora: Na obrázku 14 sú diagramy predchádzajúceho algoritmu prekreslené s pomocou cyklu s podmienkou na konci. Takéto zakreslenie je úplne korektné, avšak neodporúčame ho. Nielen z dôvodu, že sa grafická reprezentácia zbytočne rozširuje, ale aj z dôvodu, že v takomto prípade študenti často robia neúmyselné chyby. Časté chyby sú, že zabudnú nastaviť korektne počiatočné hodnoty riadiacich premenných *i* a *j*, prípadne zabudnú zakresliť, ako sa riadiaca premenná modifikuje.

Jedna z možných implementácií 1.2.1_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    const int MAX = 10;
    float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    int m, n, i, j;
    // nacitanie rozmerov matic
    do
    {
        printf("Zadaj rozmery vstupnych matic (riadky, stlpce): \n");
        scanf("%d %d", &m, &n);
    }
    while (m <= 0 || m > MAX || n <= 0 || n > MAX);

    // nacitanie prvkov matic
    printf("Zadaj prvky matice A (%d x %d) po riadkoch: \n", m, n);
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%f", &A[i][j]);
        }
    }
    printf("Zadaj prvky matice B (%d x %d) po riadkoch: \n", m, n);
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%f", &B[i][j]);
        }
    }

    // scitanie matic a vypis vyslednej matice
    printf("Matica C (dana suctom mat. A a mat. B):\n");
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            C[i][j] = A[i][j] + B[i][j];
            printf("%.2f\t", C[i][j]);
        }
        printf("\n");
    }

```

```
46      return 0;
47  }
```



Obr. 15 Konzolový výstup programu 1.2.1_Pr_1.c.

Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 1 220 B = *MAX* (*int* = 4 B) + *A[10][10]* (*float* = 4 B . 100) + *B[10][10]* (*float* = 4 B . 100) + *C[10][10]* (*float* = 4 B . 100) + *m* (*int* = 4 B) + *n* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B).

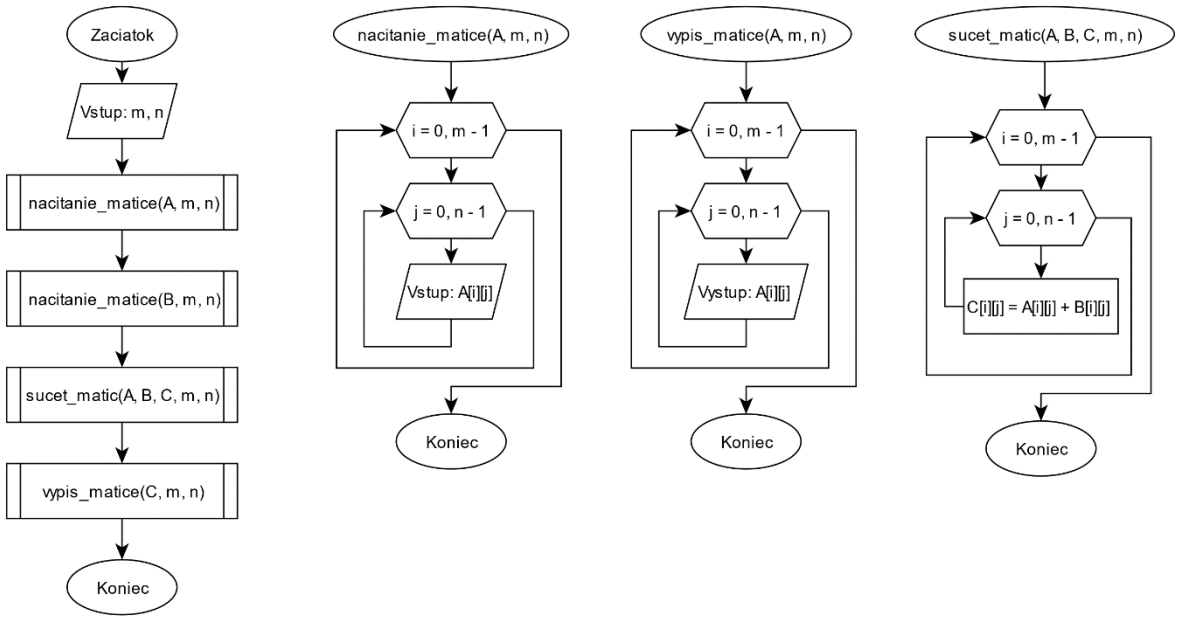
Pozn. autora: Z programu je zrejmé, že na riadku 5 sme definovali konštantu *MAX*, ktorá predstavuje maximálny rozmer matíc. Je to z toho dôvodu, že stále pracujeme so statickým poľom a ak nechceme, aby nastalo porušenie odporúčanej štruktúry programu pri štruktúrovanom programovaní (aby premenné boli deklarované hneď na začiatku programu), tak nemáme viac-menej inú možnosť. V takomto prípade nezabudnite na ošetrenie pri načítavaní rozmerov matíc aj vzhľadom na túto konštantu.

Ak by sme porušili zásady štruktúrovaného programovania, tak by začiatok programu mohol vyzerať takto:

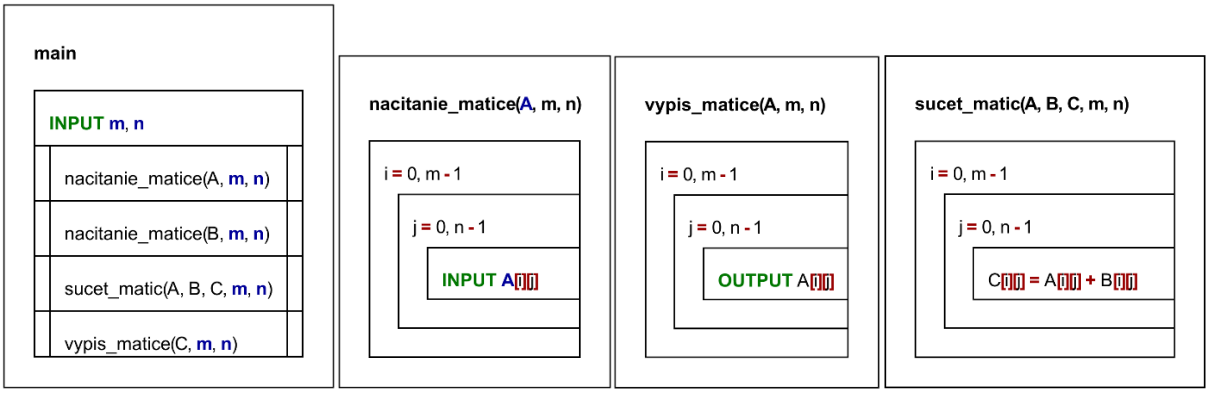
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int m, n, i, j;
6       // nacitanie rozmerov matic
```

```
7       do
8       {
9           printf("Zadaj rozmery vstupnych matic (riadky, stlpce): \n");
10          scanf("%d %d", &m, &n);
11      }
12      while (m <= 0 || n <= 0);
13
14      float A[m][n], B[m][n], C[m][n];
```

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu s použitím funkcií:



Obr. 16 Vývojový diagram na súčet matíc s pomocou funkcií.



Obr. 17 NS diagram na súčet matíc s pomocou funkcií.

Jedna z možných implementácií 1.2.1_Pr_2.c:

```c
#include <stdio.h>

const int MAX = 10;    // globalna konstanta

// deklaracie funkcii
void nacitanie_matice(float A[][MAX], int m, int n);
void sucet_matic(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n);
void vypis_matice(float A[][MAX], int m, int n);

int main(void)
{
    float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    int m, n;
    // nacitanie rozmerov matic
    do
    {
        printf("Zadaj rozmery vstupnych matic (riadky, stlpce): \n");
        scanf("%d %d", &m, &n);
    }
    while (m <= 0 || m > MAX || n <= 0 || n > MAX);

    // nacitanie prvkov matic
    nacitanie_matice(A, m, n);
    nacitanie_matice(B, m, n);

    // scitanie matic
    sucet_matic(A, B, C, m, n);

    // vypis matic
    vypis_matice(A, m, n);
    printf("\n\t+\t\n");
    vypis_matice(B, m, n);
    printf("\n\t=\t\n");
    vypis_matice(C, m, n);

    return 0;
}

// definicie funkcii
void nacitanie_matice(float A[][MAX], int m, int n)
{
    int i, j;
    printf("Zadaj prvky matice (%d x %d) po riadkoch: \n", m, n);
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%f", &A[i][j]);
        }
    }
}

void sucet_matic(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
```

```
57      {
58          for (j = 0; j < n; j++)
59          {
60              C[i][j] = A[i][j] + B[i][j];
61          }
62      }
63  }
64
65  void vypis_matice(float A[][MAX], int m, int n)
66  {
67      int i, j;
68      for (i = 0; i < m; i++)
69      {
70          for (j = 0; j < n; j++)
71          {
72              printf("%.2f\t", A[i][j]);
73          }
74          printf("\n");
75      }
76  }
```

```
C:\Programy\1.2.1_Pr_2.exe
Zadaj rozmery vstupnych matic (riadky, stlpce):
-2
0
Zadaj rozmery vstupnych matic (riadky, stlpce):
3
3
Zadaj prvky matice (3x3) (po riadkoch):
1 3 2
1 0 0
1 2 2
Zadaj prvky matice (3x3) (po riadkoch):
0 0 5
7 5 0
2 1 1
1.00    3.00    2.00
1.00    0.00    0.00
1.00    2.00    2.00

        +
0.00    0.00    5.00
7.00    5.00    0.00
2.00    1.00    1.00

        =
1.00    3.00    7.00
8.00    5.00    0.00
3.00    3.00    3.00

Process returned 0 (0x0)   execution time : 20.300 s
Press any key to continue.
```

Obr. 18 Konzolový výstup programu 1.2.1_Pr_2.c.

Vyjadrenie priestorových nárokov riešenia s funkciami:

Priestorové nároky	main()	*MAX* (*int* = 4 B) + *m* (*int* = 4 B) + *n* (*int* = 4 B) + *A[10][10]* (*float*/*double* = 4/8 B . 10 . 10) + *B[10][10]* (*float*/*double* = 4/8 B . 10 . 10) + *C[10][10]* (*float*/*double* = 4/8 B . 10 . 10) = **1 212/2 412 B**
	nacitanie_matice()	*A* (*float* ** = 4/8 B) + *m* (*int* = 4 B) + *n* (*int* = 4 B) + *i* (int = 4 B) + *j* (int = 4 B) = **20/24 B**
	vypis_matice()	*A* (*float* ** = 4/8 B) + *m* (*int* = 4 B) + *n* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B) = **20/24 B**
	sucet_matic()	*A* (*float* ** = 4/8 B) + *B* (*float* ** = 4/8 B) + *C* (*float* ** = 4/8 B) + *m* (*int* = 4 B) + *n* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B) = **28/40 B**

Netreba zabudnúť, že priestorové nároky funkcií vznikajú, keď nastáva ich vykonanie a zanikajú v momente, keď nastane ich ukončenie. Priestorové nároky funkcie *main()* sú trvalé od začiatku až po koniec programu.

Nutné vlastnosti algoritmu:

- **Determinovanosť –** táto vlastnosť algoritmu je splnená. Po načítaní prvkov vstupných matíc sa algoritmus realizuje predpísaným spôsobom, pričom v každom kroku je zrejmé, čo sa má realizovať a kadiaľ pokračuje tok riadenia.
- **Konečnosť –** každý algoritmus sa musí skončiť po vykonaní konečného počtu krokov. V tomto prípade je konečnosť zabezpečená s pomocou cyklov s pevným počtom opakovaní ako pri načítaní, spracovaní, tak aj pri výpise.
- **Rezultatívnosť** – aj táto vlastnosť je splnená. Vstupná podmienka $m, n \in \{2, 3, \ldots MAX\}$ v prípade tvorby programu bude zabezpečená deklaráciou premenných požadovaného dátového typu: *int* a ošetrením pri načítavaní. Vstupné podmienky $A[i][j] \in \mathrm{R} \wedge B[i][j] \in \mathrm{R}$, ktoré hovoria o tom, že prvky matice sú definované na množine *R*, budú zabezpečené deklarovaním polí požadovaného dátového typu (*float*/*double*). Výstupná podmienka $C[i][j] \in \mathrm{R}$ bude zabezpečená deklarovaním premennej typu pole požadovaného dátového typu (*float*/*double*) a druhá časť podmienky $C[i][j] = A[i][j] + B[i][j]$ bude zabezpečená správne navrhnutým a zrealizovaným algoritmom.

Odporúčané/očakávané vlastnosti algoritmu:

- **Hromadnosť** – môžeme konštatovať, že dané riešenie spĺňa túto vlastnosť vzhľadom na zadanie problému. Algoritmus je platný nielen pre konkrétne hodnoty, ale pre celú prípustnú množinu hodnôt, čo je zadefinované vstupnou podmienkou (rozmery matíc môžu byť ľubovoľné, avšak musí platiť, že rozmery vstupných matíc sú totožné a prvky matíc sú definované na celej množine reálnych čísiel).
- **Elementárnosť** – vlastnosť je splnená, algoritmus je zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, v ktorom sa algoritmus bude realizovať (implementovať).
- **Efektívnosť** – môžeme konštatovať, že aj táto vlastnosť je splnená. Momentálne algoritmus vyžaduje použitie siedmych premenných. Štyri premenné sú jednoduchých dátových typov, tri premenné sú štruktúrované a predstavujú matice. Priestorové nároky v závislosti od hodnôt vstupných premenných sú počas vykonávania algoritmu nemenné, z čoho vyplýva, že priestorová zložitosť je konštantná → $S(1)$.

  Časová zložitosť je kvadratická → $O(n^2)$, pretože počet vykonaní iterácií tela vonkajšieho cyklu je závislý od hodnoty vstupu *m*. Počet vykonaní iterácií tela vnútorného cyklu je závislý od hodnoty vstupu *n*. Počet iterácií je teda *m* . *n*-krát. Ak teda určujeme časovú zložitosť pre veľké hodnoty vstupov, ktoré by mohli byť označené kľudne *n*, je zrejmý vzťah medzi nami definovaným vstupom a určenou časovou zložitosťou.

  V prípade, prvého algoritmu na obrázku 13 sme dokonca zlúčili operácie, ktoré je možné vyhodnocovať súčasne, čo vedie k optimalizácii. To však nemôžeme využiť v prípade riešenia s použitím funkcií, ak chceme naplniť vlastnosti modifikovateľnosti a štruktúrovanosti. Ak použijeme na výpočet súčtu funkciu alebo keby sme načítali alebo vypísali prvky matice s použitím funkcie, mali by sme priestorové a časové nároky vyjadriť trochu inak:

Priestorová zložitosť	main()	$S(1)$ = konštantná Používa sa päť premenných: *m, n, A[m][n], B[m][n]* a *C[m][n]*.
	nacitanie_matice()	$S(1)$ = konštantná Funkcia má tri parametre: premennú, ktorá reprezentuje maticu *A* a jej rozmery *m* a *n*.

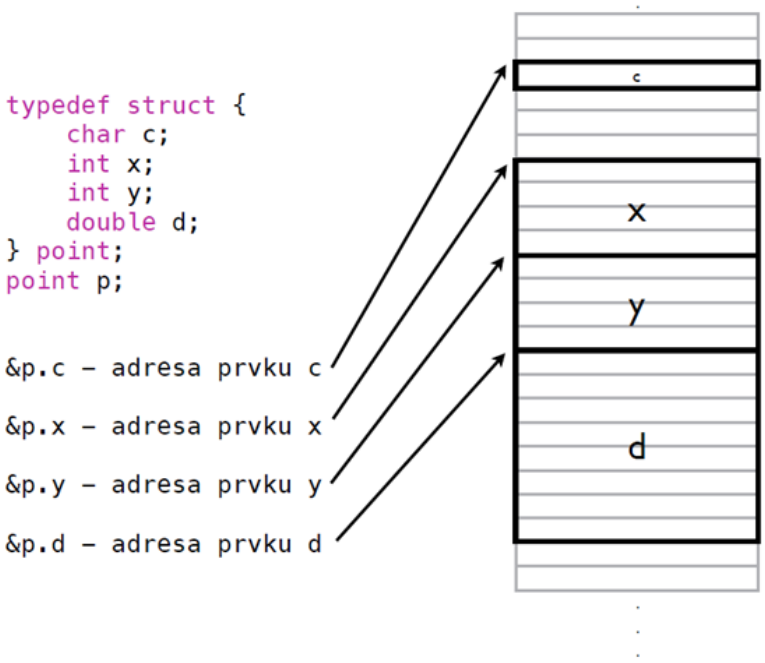
		Disponuje dvomi lokálnymi premennými *i* a *j*.
	vypis_matice()	*S*(1) = konštantná Funkcia má tri parametre: premennú, ktorá reprezentuje maticu *A* a jej rozmery *m* a *n*. Disponuje dvomi lokálnymi premennými *i* a *j*.
	sucet_matic()	*S*(1) = konštantná Funkcia má päť parametrov: tri premenné, ktoré reprezentujú matice *A*, *B* a *C* a ich rozmery *m* a *n*. Disponuje dvomi lokálnymi premennými *i* a *j*.
Časová zložitosť	main()	*O*(1) = konštantná
	nacitanie_matice()	$O(n^2)$ = kvadratická
	vypis_matice()	$O(n^2)$ = kvadratická
	sucet_matic()	$O(n^2)$ = kvadratická

1.3 Štruktúra

Štruktúra je heterogénny, používateľom definovaný údajový typ, t. j. prvky (položky, členy) štruktúry môžu mať rôzny dátový typ. Každý člen štruktúry je nejakého vlastného dátového typu. Štruktúry pomáhajú efektívnejšie organizovať komplikovanejšie dátové entity, keďže dovoľujú pracovať so skupinou premenných ako s jedným dátovým objektom. Typickým príkladom použitia štruktúry môže byť:

- zamestnanec – charakterizovaný svojím menom, priezviskom, adresou, mzdou a pod.,
- výrobok – určený identifikačným číslom, názvom, skladbou komponentov a pod.

Graficky môžeme premennú typu štruktúry v pamäti reprezentovať:



Obr. 19 Grafická reprezentácia štruktúry v pamäti.

Skutočnú veľkosť, ktorú zaberá premenná *p* v pamäti si vieme zistiť v jazyku *C s* pomocou operátora *sizeof(),* ktorý vracia veľkosť v bajtoch. V tomto prípade sa *sizeof(p)* vyhodnotí na hodnotu **20 B**. V literatúre sa dočítate, že veľkosť premennej typu štruktúry je daná súčtom veľkostí jednotlivých položiek. Jednoduchým matematickým sčítaním veľkostí jednotlivých položiek by sme dostali hodnotu 17 B, avšak ako uvádzame, reálna veľkosť danej štruktúry je 20 B. Je tak z dôvodu, že jednotlivé prvky v štruktúre môžu byť kompilátorom zarovnané (*allignment*) tak, aby začinali na adresách deliteľných napr. 2, 4 alebo 8, z dôvodu, že to môže umožniť kompilátoru preklad do efektivnejšieho binárneho kódu. Viac o tejto problematike je možné dočítať sa napríklad na GeeksforGeeks (2019).

Existuje až šesť rôznych spôsobov definície štruktúry a premennej typu štruktúry. Najčastejšie sa používa definícia pomenovanej štruktúry s oddelenou definíciou premenných, ktorej syntax je:

```
struct meno
{
   datovy_typ prvok1;
   datovy_typ prvok2;
   ...
   datovy_typ prvokN;
},
struct meno zoznam_premennych;
```

Jednotlivé spôsoby definovania štruktúry a premennej/premenných typu štruktúry (Herout, 2010):

1. **Definícia nepomenovanej štruktúry**

   Táto nepomenovaná štruktúra sa nedá v programe už nikde opakovane využiť, môžeme pracovať len s premennými *osoba1*, *osoba2*, *osoba3*. Ak by sme potrebovali viac premenných takéhoto typu, tak musíme rozšíriť zoznam premenných. Nebudeme používať.

```
struct
{
  int vyska;
  float vaha;
} osoba1, osoba2, osoba3;
```

2. **Definícia pomenovanej štruktúry**

   V tomto prípade môžeme využiť štruktúru opakovane a definovať v prípade potreby ďalšie premenné daného typu. Budeme používať.

```
struct miery
{
  int vyska;
  float vaha;
} osoba1, osoba2, osoba3;
struct miery osoba4;
```

3. **Definícia pomenovanej štruktúry s oddelenou definíciou premenných**

   V tomto prípade sa robí definícia štruktúry oddelene od deklarácie premenných. Nezabudnite pri definícii premenných použiť kľúčové slovo *struct*. Budeme používať.

```
struct miery
{
  int vyska;
  float vaha;
};
struct miery osoba1;
struct miery osoba2, osoba3;
```

4. **Definícia typu nepomenovanej štruktúry**

   Definícia nového typu sa realizuje s pomocou kľúčového slova *typedef*. Štruktúra nie je pomenovaná. Pozor, pri deklarácii premenných nie je použité kľúčové slovo *struct*.

```
typedef struct
{
  int vyska;
  float vaha;
}MIERY;
```

   Identifikátor *MIERY* je názov nového dátového typu a premenné typu tejto štruktúry sa deklarujú takto:

```
MIERY osoba1, osoba2, osoba3;
```

5. **Definícia typu pomenovanej štruktúry**

   Najčastejšie sa používa pri realizácii zoznamov (list-ov), keď štruktúra odkazuje sama na seba. Budeme používať.

```
typedef struct miery
{
  int vyska;
  float vaha;
} MIERY;
MIERY osoba1, osoba2, osoba3;
```

Použitie štruktúry vo všeobecnosti prebieha v štyroch krokoch:

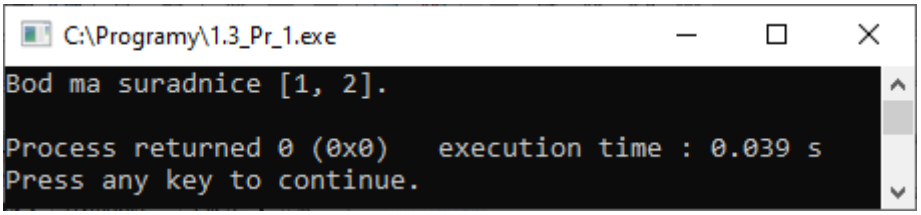
1. Definícia štruktúry
2. Deklarácia premennej typu štruktúry
3. Inicializácia premennej typu štruktúry s pomocou bodkovej (.) notácie
4. Použitie premennej typu štruktúry

Pozn.: K jednotlivým položkám štruktúry pristupujeme cez **operátor bodky '.'**. V samotnej štruktúre môže existovať položka, ktorá predstavuje pole, ale môže existovať aj pole štruktúr. Príklad ilustruje pole typu *MIERY*, kde každá položka poľa je reprezentovaná jednou štruktúrovanou položkou typu štruktúry *MIERY*.

```
MIERY a[10];
a[5].vyska = 187;   // priradenie vysky 6-temu prvku pola
```

Ilustrácia použitia štruktúry na príklade 1.3_Pr_1.c:

```c
#include <stdio.h>
// definicia struktury
struct bod
{
    int x;
    int y;
};

int main(void)
{
    // deklaracia premennej bod1 strukturovaneho udajoveho typu
    struct bod bod1;

    // inicializacia premennej
    bod1.x = 1;
    bod1.y = 2;

    // pouzitie premennej typu struktury
    printf("Bod ma suradnice [%d, %d].\n", bod1.x, bod1.y);

    return 0;
}
```

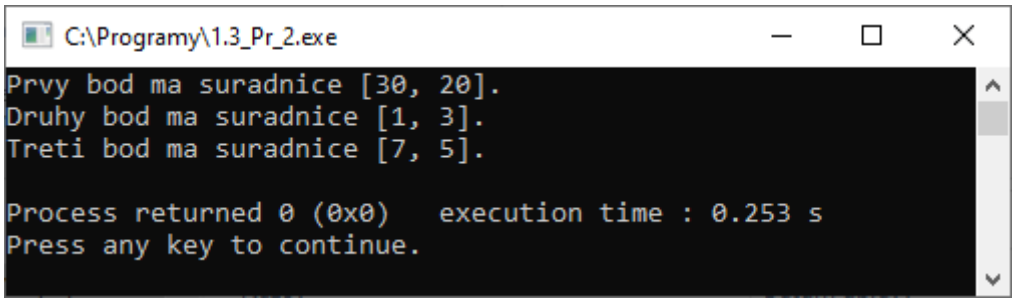


Obr. 20 Konzolový výstup programu 1.3_Pr_1.c.

Ilustrácia použitia štruktúry a poľa 1.3_Pr_2.c:

```c
#include <stdio.h>
// definicia typu struktury
typedef struct bod
{
    int x;
    int y;
} BOD;

int main(void)
{
    // definicia pola, prvky su struktury
    BOD pole_bodov[]= {{1,1}, {3,3},{7,5}};

    // editovanie suradnic prveho bodu
    pole_bodov->x = 30;
    pole_bodov->y = 20;

    // editovanie suradnice x druheho bodu
    pole_bodov[1].x = 1;

    // pouzitie premennej typu struktury
```

```
22      printf("Prvy bod ma suradnice [%d, %d].\n", pole_bodov[0].x, pole_bodov->y);
23      printf("Druhy bod ma suradnice [%d, %d].\n", pole_bodov[1].x, pole_bodov[1].y);
24      printf("Treti bod ma suradnice [%d, %d].\n", pole_bodov[2].x, pole_bodov[2].y);
25
26      return 0;
27  }
```



Obr. 21 Konzolový výstup programu 1.3_Pr_2.c.

Pozn. autora: Celé pole môžeme naplniť (ale len pri deklarácii) spôsobom, ktorý je vidieť v programe, t. j. s pomocou inicializačného zoznamu – riadok 12. V prípade, že pristupujeme k jednotlivým položkám štruktúry cez premennú typu smerník, mení sa bodková notácia na šípkovú, čiže namiesto operátora bodky '**.**' sa použije operátor šípky '−>', čo je znak zložený zo znakov „mínus“ a „väčšie“. Keďže premenná *pole_bodov* je typu pole a vieme, že názov poľa je zároveň smerníkom, ktorý ukazuje na prvú položku poľa, môžeme k položkám štruktúry prvého bodu pristupovať tak cez operátor šípky, ako aj bodky (napr. riadok 19). Aj Belan (2011) deklaruje, že sa pri volaní funkcií častejšie ako parameter používa smerník na štruktúru, než celá štruktúra a že je tento druhý spôsob používaný relatívne často. Viac o pointeroch a práci s nimi sa dočítate v kapitole 4. Z tohto dôvodu pristupujeme k ďalším položkám poľa štandardným spôsobom, s pomocou indexov a bodkovej notácie.

1.3.1 Príklad – určenie vzájomnej polohy bodu a kružnice

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte riešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém určenia vzájomnej polohy bodu a kružnice v dvojrozmernom priestore.

Vstup (vstupné premenné):

Ax – *x*-ová súradnica bodu *A*

Ay – *y*-ová súradnica bodu *A*

Sx – *x*-ová súradnica stredu kružnice *S*

Sy – *y*-ová súradnica stredu kružnice *S*

r – polomer kružnice

Výstup (výstupné premenné):

Výpisy:

Bod leží v časti roviny ohraničenej kružnicou.

Bod leží na kružnici.

Bod leží mimo kružnice.

Bod je totožný so stredom kružnice.

Vstupné podmienky:

$Ax, Ay, Sx, Sy \in \mathrm{R} \wedge r \in \mathrm{R}^{+}$

Výstupné podmienky:

Bod leží v časti roviny ohraničenej kružnicou. Ak $vzd < r$

Bod leží na kružnici. Ak $vzd = r$

Bod leží mimo kružnice. Ak $vzd > r$

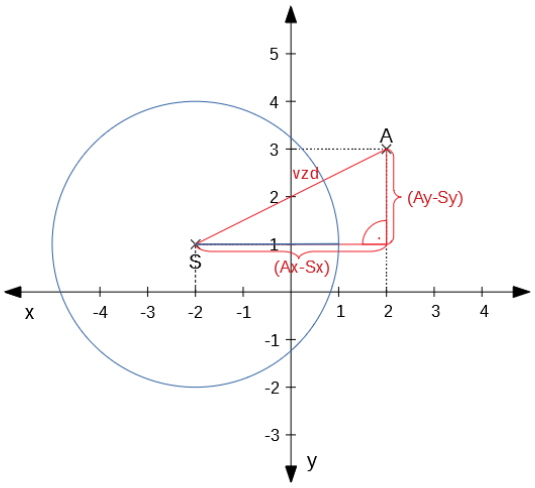
Bod je totožný so stredom kružnice. Ak $vzd = 0$

Pomocné premenné:

vzd – vzdialenosť bodu *A* od stredu kružnice *S*

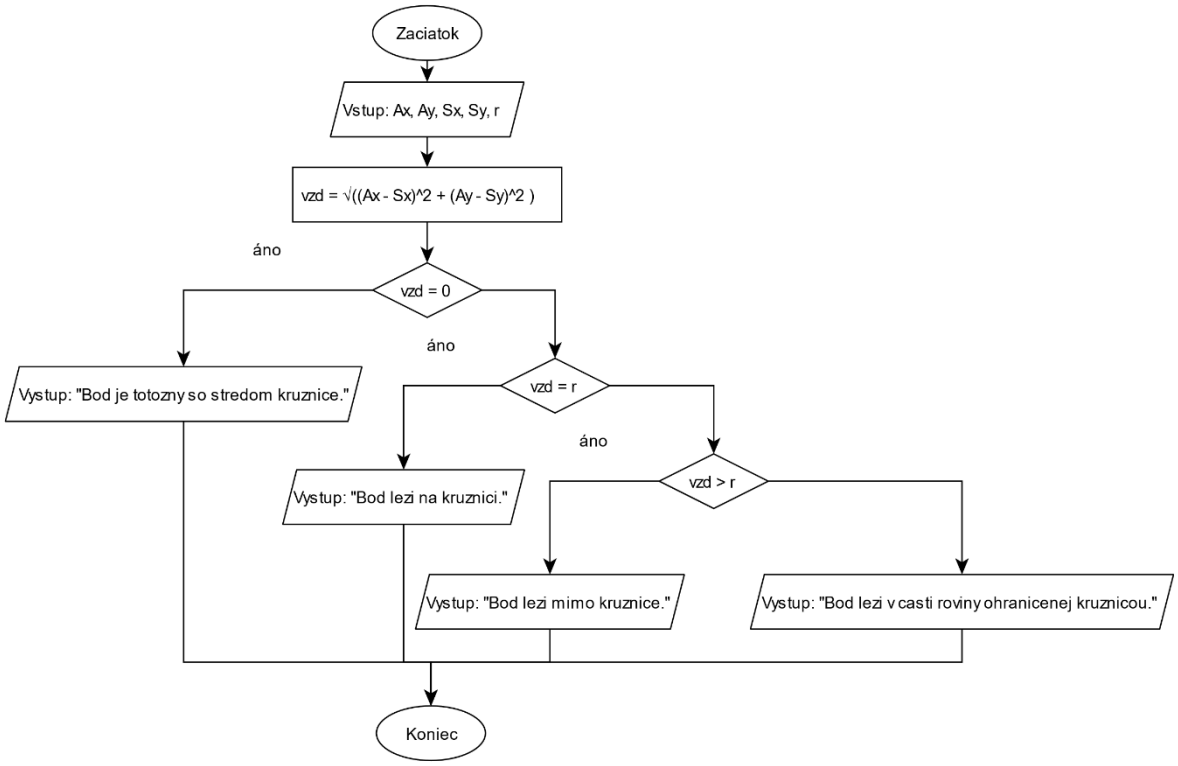
$vzd \in \mathrm{R}_0^{+} \wedge vzd = \sqrt{(Ax - Sx)^2 + (Ay - Sy)^2}$

Rozbor úlohy: Na určenie vzájomnej polohy bodu a kružnice je v prvom rade potrebné poznať vstupné údaje pri dodržaní vstupných podmienok. Následne je možné na základe Pytagorovej vety vypočítať vzdialenosť bodu od stredu kružnice, ktorá vlastne predstavuje preponu. Na základe tejto hodnoty podľa definovaných výstupných podmienok, informovať používateľa o výsledku.

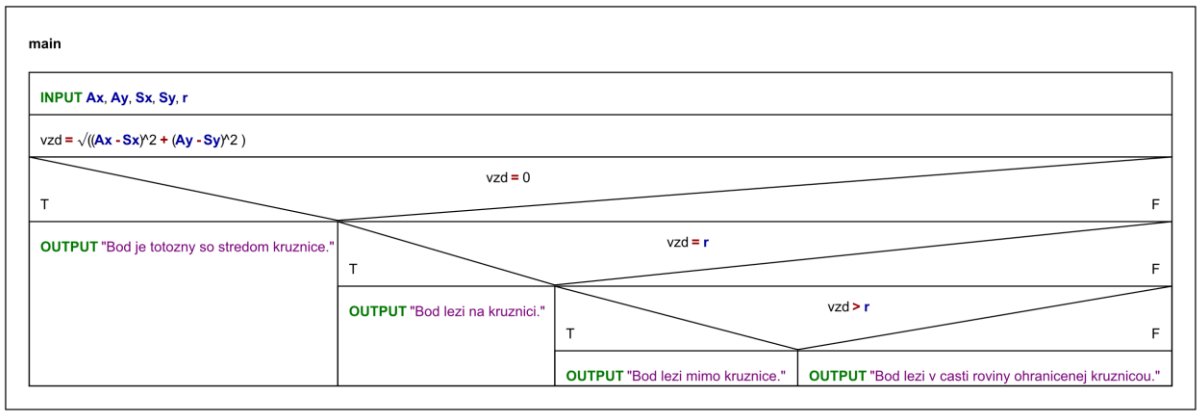


Obr. 22 Ilustrácia výpočtu vzdialenosti bodu *A* od stredu kružnice *S*.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 23 Vývojový diagram na určenie vzájomnej polohy bodu a kružnice.

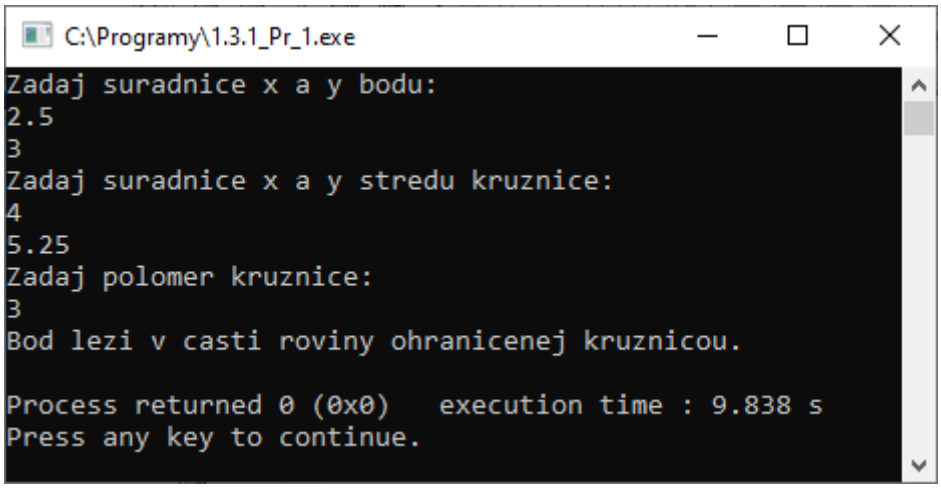


Obr. 24 NS diagram na určenie vzájomnej polohy bodu a kružnice.

Overenie správnosti algoritmu aj posúdenie vlastností ponechávam na čitateľa.

Jedna z možných implementácií 1.3.1_Pr_1.c:

```c
#include <stdio.h>
#include <math.h>   // sqrt()

int main(void)
{
    float Ax, Ay, Sx, Sy, r, vzd;   // deklaracia premennych

    // nacitanie hodnot
    printf("Zadaj suradnice x a y bodu:\n");
    scanf("%f %f", &Ax, &Ay);

    printf("Zadaj suradnice x a y stredu kruznice:\n");
    scanf("%f %f", &Sx, &Sy);

    do
    {
        printf("Zadaj polomer kruznice:\n");
        scanf("%f", &r);
    }
    while (r <= 0);

    // vypocet vzdialenosti
    vzd = sqrt((Ax - Sx) * (Ax - Sx) + (Ay - Sy) * (Ay - Sy));

    // vyhodnotenie vysledkov
    if (vzd == 0)
        printf("Bod je totozny so stredom kruznice.\n");
    else if (vzd == r)
        printf("Bod lezi na kruznici.\n");
    else if (vzd > r)
        printf("Bod lezi mimo kruznice.\n");
    else
        printf("Bod lezi v casti roviny ohranicenej kruznicou.\n");

    return 0;
}
```



Obr. 25 Konzolový výstup programu 1.3.1_Pr_1.c.

Pozn. autora: Ako dátový typ pre premenné reprezentujúce bod a kružnicu je možné použiť jednoduché dátové typy, ako sme ilustrovali v riešení úlohy z príkladu vyššie. Ak by sme sa však zamysleli takým smerom, že by bolo nevyhnutné v budúcnosti problém rozšíriť aj o určovanie vzájomnej polohy kružníc, prípadne priamky a pod., počet premenných by značne narastal, čo by v konečnom dôsledku znižovalo čitateľnosť programu. V takomto prípade sa zdá byť použitie štruktúry viac ako vhodné nielen pri implementácii, ale aj pri algoritmickom riešení. Vhodné by bolo zaviesť dve štruktúry, ktoré reprezentujú použité geometrické útvary. V takomto prípade sa definícia vstupov a výstupov zmení takto:

Zavádzame dva typy pre bod a kružnicu:

```
typedef struct bod
{
    float x;
    float y;
} BOD;
```

```
typedef struct kruznica
{
    BOD s;
    float r;
} KRUZNICA;
```

Vstup (vstupné premenné):

A – bod

k – kružnica

Výstup (výstupné premenné):

Výpisy:

Bod leží v časti roviny ohraničenej kružnicou.

Bod leží na kružnici.

Bod leží mimo kružnice.

Bod je totožný so stredom kružnice.

Vstupné podmienky:

$A.x, A.y, k.S.x, k.S.y \in \mathrm{R} \wedge k.r \in \mathrm{R}^{+}$

Výstupné podmienky:

Bod leží v časti roviny ohraničenej kružnicou. Ak *vzd* < *k.r*

Bod leží na kružnici. Ak *vzd* = *k.r*

Bod leží mimo kružnice. Ak *vzd* > *k.r*

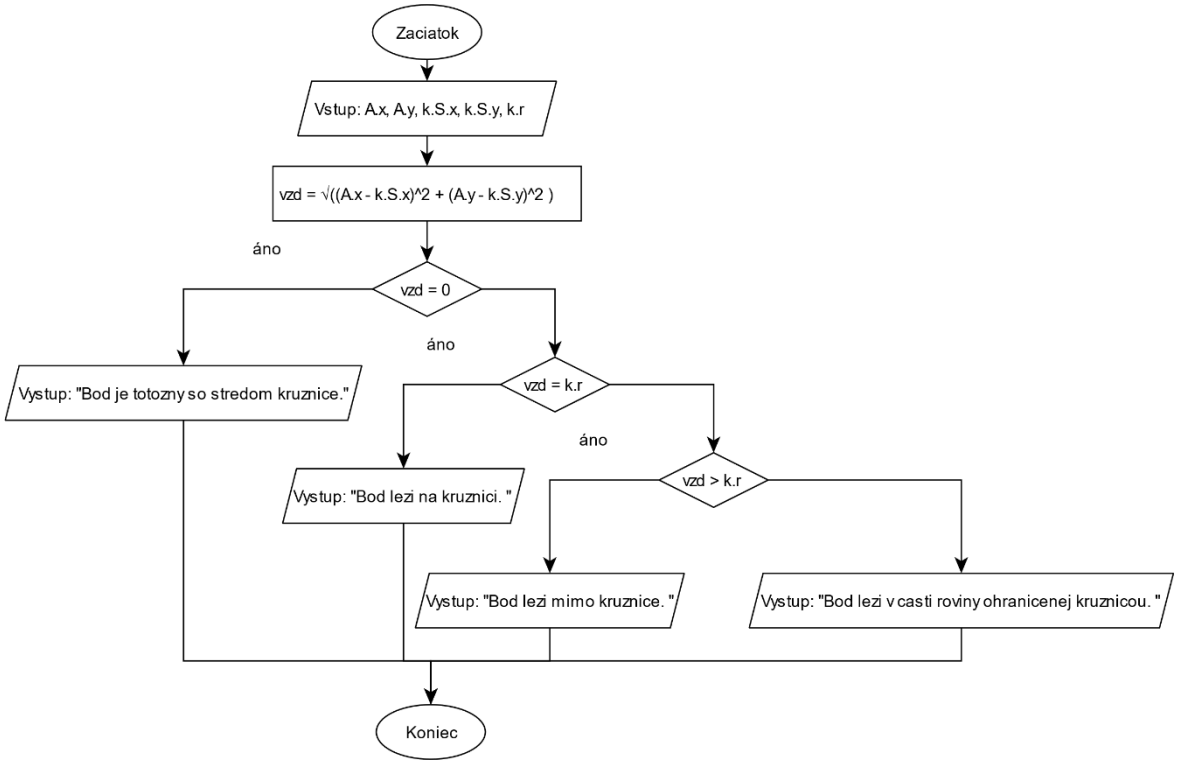
Bod je totožný so stredom kružnice. Ak *vzd* = 0

Pomocné premenné:

vzd – vzdialenosť bodu od stredu kružnice

$vzd \in \mathrm{R}_0^{+} \wedge vzd = \sqrt{(A.x - k.S.x)^2 + (A.y - k.S.y)^2}$

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 26 Vývojový diagram na určenie vzájomnej polohy bodu a kružnice s použitím štruktúry.

Niekto by mohol povedať, že takáto reprezentácia porušuje vlastnosť elementárnosti, pretože využívame priamo bodkovú notáciu známu z jazyka *C* na prácu so štruktúrou. Avšak na strane druhej, predstavte si, že by skutočne bolo nevyhnutné mať v programe deklarované premenné,

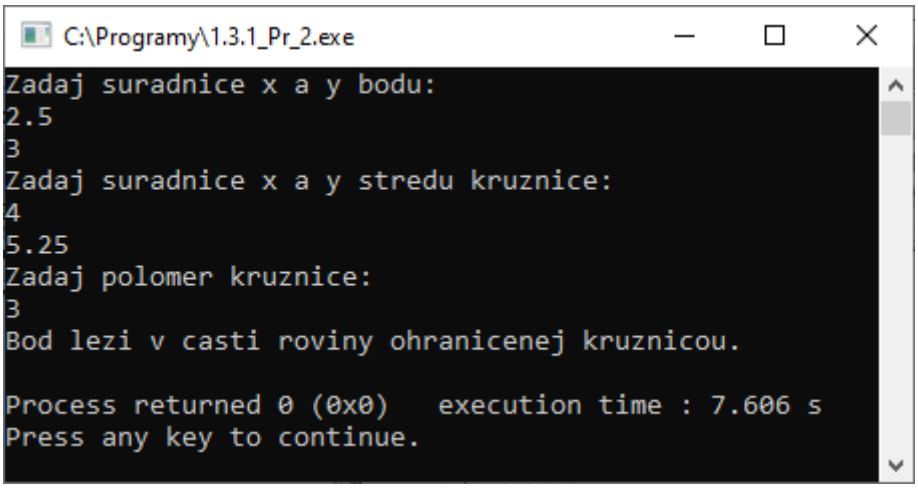
napríklad pre tri kružnice, päť bodov a dve priamky. V prípade použitia jednoduchých dátových typov by bolo nevyhnutné deklarovať minimálne 27 premenných (každá kružnica je reprezentovaná stredom a polomerom, t. j. 3 . 3 = 9 premenných, bod je reprezentovaný v rovine dvomi súradnicami, t. j. 5 . 2 = 10 a pri priamke potrebujeme pracovať s dvomi bodmi, ktoré na nej ležia pre každú priamku, t. j. 2 . 2 . 2 = 8).

Použiť štruktúru pri implementácii je samozrejme možné aj bez zohľadnenia tejto skutočnosti počas algoritmickéo riešenia problému.

Jedna z možných implementácií 1.3.1_Pr_2.c:

```c
#include <stdio.h>
#include <math.h>   // sqrt()

// definicie typov struktury
typedef struct bod
{
    float x;
    float y;
} BOD;

typedef struct kruznica
{
    BOD s;
    float r;
} KRUZNICA;

int main(void)
{
    // deklaracia premennych
    BOD a;
    KRUZNICA k;
    float vzd;

    // naciatanie hodnot
    printf("Zadaj suradnice x a y bodu:\n");
    scanf("%f %f", &a.x, &a.y);

    printf("Zadaj suradnice x a y stredu kruznice:\n");
    scanf("%f %f", &k.s.x, &k.s.y);

    do
    {
        printf("Zadaj polomer kruznice:\n");
        scanf("%f", &k.r);
    }
    while (k.r <= 0);

    // vypocet vzdialenosti
    vzd = sqrt((a.x - k.s.x) * (a.x - k.s.x) + (a.y - k.s.y) * (a.y-k.s.y));

    // vyhodnotenie vysledkov
    if (vzd == 0)
        printf("Bod je totozny so stredom kruznice.\n");
```

```
44      else if (vzd == k.r)
45          printf("Bod lezi na kruznici.\n");
46      else if (vzd > k.r)
47          printf("Bod lezi mimo kruznice.\n");
48      else
49          printf("Bod lezi v casti roviny ohranicenej kruznicou.\n");
50
51      return 0;
52  }
```



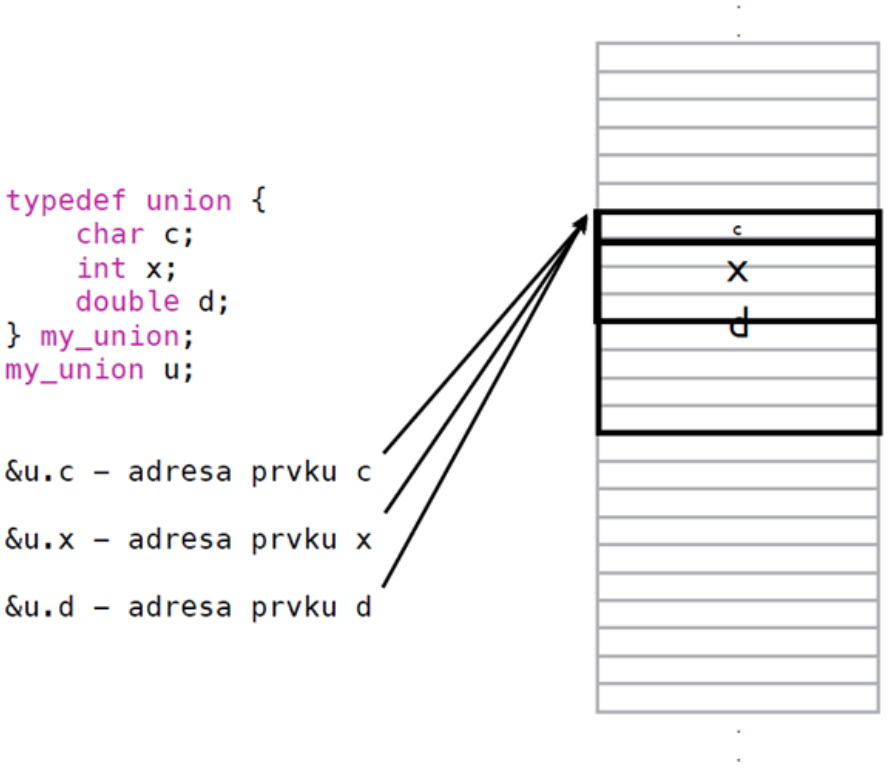
Obr. 27 Konzolový výstup programu 1.3.1_Pr_2.c.

1.4 Union

Union je nehomogénny dátový typ, ktorý na rozdiel od štruktúry má veľkosť rovnú veľkosti najväčšej položky. Všetky položky unionu sa začínajú na rovnakej adrese, ktorá je zhodná s adresou premennej. K položkám unionu sa pristupuje tiež cez bodkovú notáciu, prípadne šípkovú. Definícia sa robí podobným spôsobom ako u štruktúry:

```
union meno
{
  datovy_typ prvok 1;
  datovy_typ prvok 2;
  ...
  datovy_typ prvok N;
}zoznam-premennych;
```

Graficky môžeme premennú typu union v pamäti reprezentovať:



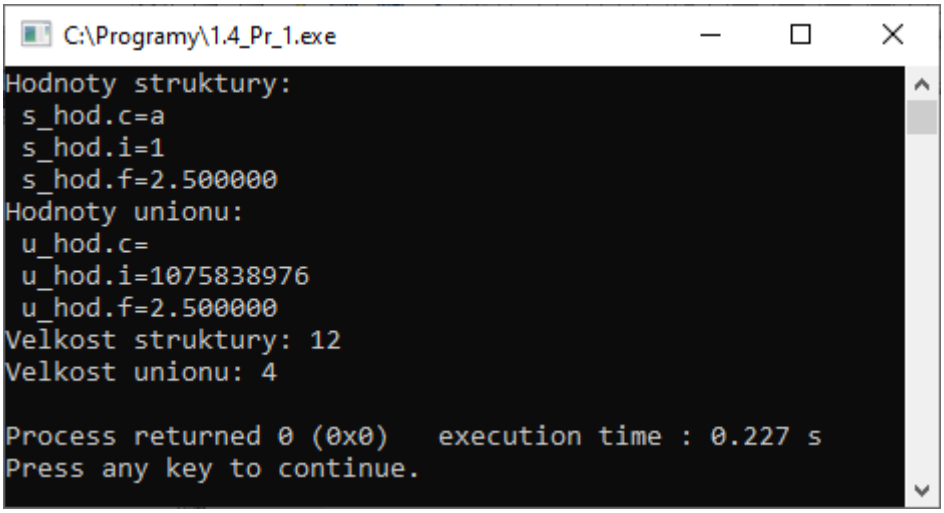
Obr. 28 Grafická reprezentácia unionu v pamäti.

Všetky položky unionu sa začínajú na rovnakej adrese, ktorá je v tomto príklade zhodná s adresou premennej *u* ktorá je typu *my_union*. Veľkosť tejto premennej je rovná veľkosti najväčšej položky, teda *sizeof(double) = 8 B*.

Príklad nižšie ilustruje použitie premennej typu štruktúry a unionu a rozdiel medzi nimi 1.4_Pr_1.c:

```c
#include <stdio.h>

// definicia typu struktury
typedef struct
{
    char c;
    int i;
    float f;
} SH;

// definicia typu unionu
typedef union
{
    char c;
    int i;
    float f;
} UH;

int main(void)
{
```

```c
    // deklaracia premennych
    SH s_hod;
    UH u_hod;

    // inicializacia premennych
    s_hod.c = 'a';
    s_hod.i = 1;
    s_hod.f = 2.5;

    u_hod.c = 'a';
    u_hod.i = 1;
    u_hod.f = 2.5;

    // vyhodnotenie vysledkov
    printf("Hodnoty struktury:\n");
    printf(" s_hod.c = %c\n s_hod.i = %d\n s_hod.f = %f\n", s_hod.c, s_hod.i, s_hod.f);
    printf("Hodnoty unionu:\n");
    printf(" u_hod.c = %c\n u_hod.i = %d\n u_hod.f = %f\n", u_hod.c, u_hod.i, u_hod.f);
    printf("Velkost struktury: %d\n", sizeof(s_hod));
    printf("Velkost unionu: %d\n", sizeof(u_hod));

    return 0;
}
```



Obr. 29 Konzolový výstup programu 1.4_Pr_1.c.

Pozn. autora: Dátový typ *struct* má veľkosť rovnú 12 B, aj keď jednoduchým sčítaním veľkostí jednotlivých položiek dostaneme 9 B, *sizeof(char) + sizeof(int) + sizeof(float) = 9 B*. O tomto probléme sme pojednávali vyššie. Jednotlivé položky sa začínajú na rôznych adresách a ležia v pamäti za sebou, preto sa všetky hodnoty zobrazia tak ako sme ich naplnili. Veľkosť unionu sa vyhodnocuje podľa veľkosti najväčšej položky, t. j. *sizeof(float) = 4 B*, pričom všetky položky sa začínajú na jednej a tej istej adrese. Táto rozdielna interpretácia vedie k tomu, že do premennej typu *unionu* sú postupne ukladané všetky položky, ktoré sa postupne prepisujú a ostáva zachovaná len naposledy vložená, teda hodnota typu *float*.

1.5 Úlohy na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problém, ktorý načíta do poľa (resp. matice – dvojrozmerného poľa) postupnosť čísiel a následne vypočíta priemer z prvkov poľa.
2. Algoritmicky riešte problém, ktorý načíta do poľa (resp. matice – dvojrozmerného poľa) postupnosť čísiel a zistí maximálny a minimálny prvok, zistí početnosť ich výskytov, ako aj indexy na ktorých sa tieto prvky nachádzajú v poli.
3. Algoritmicky riešte problém, ktorý načíta do poľa (resp. matice – dvojrozmerného poľa) postupnosť čísiel a utriedi tieto prvky (pre maticu v jednotlivých stĺpcoch, prípadne riadkoch matice).
4. Algoritmicky riešte problém, ktorý načíta do poľa postupnosť znakov a následne vypíše znaky v opačnom poradí, ako boli zadané.
5. Algoritmicky riešte problém, ktorý s pomocou algoritmu Eratosthenovho sita určí a vypíše všetky prvočísla po zadanú hranicu.
6. Algoritmicky riešte problém, ktorý načíta do poľa postupnosť znakov a pre každý znak určí, či je toto písmeno veľké, malé, či ide o číslo, alebo iný špeciálny znak.
7. Algoritmicky riešte problém, ktorý načíta do poľa postupnosť čísiel, porovná každý prvok s každým a označí vzťah medzi nimi (<, >, =).
8. Algoritmicky riešte problém, ktorý pre zadávané písmená spočíta, koľkokrát bolo každé písmeno zadané. Zadávanie ukončite zadaním zvoleného znaku.
9. Algoritmicky riešte problém, ktorý načíta do matice – dvojrozmerného poľa postupnosť čísiel a následne vypočíta a vypíše súčet prvkov na jednotlivých riadkoch (stĺpcoch) danej matice.
10. Algoritmicky riešte problém, ktorý umožní zadanie výšky a váhy desiatich osôb. Následne zistite osobu s najväčšou váhou a vypíšte údaje o tejto osobe.
11. Algoritmicky riešte problém, ktorý umožní riešiť súčet zadaných zlomkov. Na reprezentáciu zlomkov použite štruktúrovaný typ. Výsledok uvádzajte v základnom tvare. Napr. 4/3 a −2/6 je 6/6 t. j. 1.

2 Funkcie – základná terminológia

Funkcia je nezávislá časť programového kódu, ktorá vykonáva určitú (spravidla jednu) úlohu (určenú skupinou príkazov) a má pridelené meno (identifikátor), za ktorým nasledujú bez medzery okrúhle zátvorky, v ktorých sú argumenty (formálne parametre). Je dôležité voliť výstižné, ale zároveň krátke meno pre funkciu. Vo väčšine prípadov ak má vývojár s týmto problém, tak sa snaží, aby funkcia robila viac než jednu činnosť, čo svedčí o nie vhodnej dekompozícii problému. Funkciu môžeme chápať aj ako podprogram, ktorý má vlastné deklarácie a príkazy. Volanie funkcie, ktoré predstavuje určenie danej funkcie jej menom a odovzdaním skutočných argumetov v programe zabezpečuje vykonanie tela funkcie.

Program v jazyku *C* obsahuje jednu alebo viacero definícií funkcií, z ktorých jedna sa vždy musí volať *main()*. Spracovanie programu sa začína volaním funkcie *main()* a končí opustením tejto funkcie (Horovčák a Podlubný, 1997).

Hlavička funkcie informuje prekladač, aké údaje (počet a typ parametrov) funkcia prijíma z volajúceho programu a aký údaj (typ návratovej hodnoty) funkcia vráti volajúcemu programu. Hlavička funkcie nie je ukončená bodkočiarkou ';'. Má tvar:

```
navratovy_typ meno_funkcie(deklaracia_parametrov)
```

- Deklarácia parametrov je zoznam dvojíc: dátový typ parametra a jeho označenie, oddelené čiarkami. Nazývame ich **formálne parametre** a pri volaní funkcie sú nahradené skutočnými hodnotami nazývanými **aktuálne (skutočné) parametre.** Napríklad takáto definícia nie je možná: `double max(double x, y);` je nevyhnutné opraviť na: `double max(double x, double y);` Funkcia, ktorá nemá žiadne parametre, musí byť definovaná vrátané oboch okrúhlych zátvoriek, v ktorých by mal byť uvedený typ *void*.
- Formálne parametre, ktoré sú odovzdané hodnotou, sa správajú ako lokálne premenné funkcie a sú prístupné (viditeľné) iba vo funkcii, t. j. sú skryté zvonka tejto funkcie. Časový interval počas ktorého program pracuje s týmito parametrami sa často označuje ako doba života, alebo existencia parametra.

Deklaráciu funkcie = prototyp funkcie určuje len hlavička funkcie (typ návratovej hodnoty, meno funkcie a prípadne typ a počet parametrov) ukončená bodkočiarkou ';'.

```
navratovy_typ meno_funkcie(typ p1, typ p2);
int max(int a, int b);
```

Definíciu funkcie určuje hlavička funkcie spolu s jej telom.

- Telo funkcie je uzavreté do zložených zátvoriek { } a môže obsahovať príkazy aj deklarácie/definície premenných.
- Každá funkcia v jazyku *C* má maximálne jeden výstup (návratovú hodnotu), vrátenú s pomocou príkazu *return*. Tento príkaz ukončuje funkciu. Je možné z funkcie vrátiť aj viacero hodnôt – s pomocou smerníkov, resp. globálnych premenných. Návratový typ nemôže byť pole. V prípade ak je typ funkcie (návratová hodnota) vynechaný, bolo zvykom v jazyku *C*, že táto návratová hodnota bola určená implicitne na typ *int*, napr. pri zápise deklarácie funkcie: *max(int a, int b);* (Herout, 2010). Väčšina prekladačov na túto skutočnosť upozorní s pomocou *warnings*. Toto však nie je zaručené. Preto v prípade, ak funkcia nemá vrátiť žiadnu hodnotu, použijeme *void*. Odporúčame vždy uvádzať typ a nespoliehať sa na implicitné nastavenia. Príkaz *return* musí mať každá funkcia čo nevracia *void*, napr.:

```
return 0;                   // navrat konstanty 0
return a * a + b;           // navrat vyhodnoteneho vyrazu
return n == 10 ? 0 : 1;     // navrat hodnoty podmieneneho vyrazu
return status;              // navrat premennej "status"
return (status);            // mozu byt pouzite zatvorky, ale aj nemusia
return;                     // navrat z funkcie co vracia void, pouzity skokovy prikaz
```

Odporúčaná štrúktúra definície funkcie:

```
navratovy_typ meno_funkcie(formalne_parametre)
{
  deklaracie;
  prikazy;
  return navratova_hodnota;
}
```

Definícia funkcie na nájdenie maximálnej hodnoty z dvoch hodnôt by mohla vyzerať takto:

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Vo funkcii nemôže byť vnorená ďalšia funkcia, t. j. jedna funkcia nemôže obsahovať vo svojom tele definíciu druhej funkcie, ale **funkcia môže byť vo vnútri inej funkcie volaná** (Herout, 2010)**.**

Použitie funkcie v konečnom dôsledku prináša prehľadnejší zdrojový kód, t. j. ich použitie vedie k členeniu zdrojového kódu. Taktiež vplýva na elimináciu duplicitného kódu, čiže

opakujúce sa časti kódu je vhodné umiestňovať do funkcií. A tiež znovupoužiteľnosť, teda možnosť použiť už naprogramované funkcie z iných projektov.

Funkcia sa volá uvedením jej mena a vstupnými argumentmi, ak nejaké má:

```
meno_funkcie(vstupne_argumenty);
```

Napr.:

```
printf("Zadaj 2 cisla:\n");
scanf("%d %d", &a, &b);
printf("Maximum z %d a %d je %d.\n", a, b, max(a, b));
```

Ak funkcia nemá parametre tak je nevyhnutné ju volať vrátane zátvoriek: `say_hello();`, volanie `say_hello;` by bolo chybné. Ak funkcia vracia hodnotu, môže byť použitá ako výraz: `d = max(a, b) + 10.5;` Pri volaní funkcie môže dôjsť aj k stavu, že jej výsledok nie je potrebný, nemusí byť použitý/zapamätaný, v takom prípade funkciu voláme napr. v tvare: `max(a, b);`

2.1 Procedúry v jazyku *C*

Procedúra je v podstate podprogram, ktorý vykonáva špecifickú úlohu. Je to v podstate časť kódu vo vnútri väčšieho programu, ktorá môže byť relatívne nezávisla od ostatného kódu, t. j. procedúru tvoria príkazy vyčlenené z hlavného programu uzatvorené do jedného celku.

Formálne, procedúry v jazyku *C* neexistujú, t. j. jazyk *C* nerozlišuje medzi funkciami a procedúrami. Podľa Herouta (2010) existujú však dva spôsoby, ako toto tvrdenie obísť:

1. Funkcia síce návratovú hodnotu vráti, ale nikto ju nechce. Typickým príkladom je čakanie na stlačenie klávesy s pomocou funkcie *getchar()* deklarovanej v hlavičkovom súbore *stdio.h*, ktorá pri normálnom použití vráti stlačený znak. Novšie prekladače vyžadujú v takejto situácii explicitné pretypovanie na typ *void*, aby bolo jasné, že programátor návratovú hodnotu skutočne nepotrebuje. Príklad volania takejto funkcie: `getchar();` alebo `(void)getchar();`
2. Funkcia sa definuje ako funkcia vracajúca typ *void*. Vtedy nie je nevyhnutné uvádzať ani príkaz *return* v tele funkcie. Príkaz *return* sa použije iba v prípade, že je nevyhnutné na základe nejakej podmienky ukončiť funkciu pred dosiahnutím jej konca. Napr.:

```
void pozdrav(void)
{
    printf("Ahoj ludia!\n");
}
```

Nižšie uvedená funkcia má návratový typ *void* – nevracia nič a ako parameter má celočíselnú premennú, ktorá sa vo funkcii iba vytlačí.

```
void tlac_znak(int i)
{
    printf("Znak %c.\n", i);
}
```

2.2 Odporúčané umiestnenia častí funkcie

Deklarácia funkcie sa umiestňuje pred hlavnú funkciu *main()*, alebo ešte lepšie do prislúchajúcich hlavičkových súborov. Definícia funkcie sa obvykle udáva až za hlavnou funkciou. Kód funkcie je možné uvádzať aj pred hlavnou funkciou *main()*, ale z hľadiska prehľadnosti je prvý spôsob vhodnejší. Opierajúc sa o názor Belana (2011), ak chcete použiť nejakú funkciu, máte dve možnosti. Buď ju zadefinujete pred miestom použitia, alebo až potom. V druhom prípade ale nastáva problém. Keď kompilátor číta program, narazí na použitie funkcie, pričom netuší, čo to má byť za funkciu a vypíše chybu. Preto musíte povedať dopredu, ako tá funkcia vyzerá, t. j. musíte ju deklarovať aby kompilátor pri použití vedel, že tá funkcia je v poriadku. Napríklad:

```
void strasidlo(int i);  // tu je deklaracia funkcie = prototyp funkcie
int main(void)
{
    strasidlo(5);   // volanie funkcie
    return 0;
}
// tu je definicia funkcie
void strasidlo(int i)
{
    int j;
    for (j = 0; j < i; j++)
        printf("Bubu.\n");
}
```

Obdobný problém môže nastať aj v prípade, ak jedna funkcia je volaná z vnútra inej. Problém môže nastať, keď je volaná funkcia definovaná až za definíciou funkcie, ktorá túto funkciu volá. Volajúca funkcia v tomto prípade nemá doposiaľ žiadne informácie o volanej funkcii. Teda prekladaču je nevyhnutné oznámiť návratový typ, meno volanej funkcie a počet a typ parametrov volanej funkcie ešte pred jej volaním. Je teda potrebné zadeklarovať volanú funkciu s pomocou *prototypu funkcie* ešte pred samotným volaním funkcie.

2.3 Odovzdávanie parametrov funkcii

Odovzdávanie parametrov funkcii sa v jazyku *C* vykonáva len **hodnotou (passed by value)**, avšak hodnotou môže byť aj adresa. Pri tomto spôsobe prenášania parametrov sa hodnoty aktuálnych (skutočných) parametrov kopírujú cez zásobník do formálnych parametrov funkcie. Pri prenášaní parametrov hodnotou, skutočné parametre nemôžu byť vo funkcii zmenené, pretože vo funkcii sa pracuje s kópiou skutočných parametrov. Teda akákoľvek zmena parametrov vo vnútri funkcie je iba dočasná a po opustení funkcie sa stráca. Typ skutočného parametru by mal byť rovnaký ako typ formálneho parametru. Inak sa prevedie typová konverzia na typ uvedený v prototype funkcie (Herout, 2010).

Príklad: Na základe uvedeného prototypu funkcie je zrejmé, že funkcia *max()* očakáva dva parametre, ktorých dátový typ je reálne číslo (*double*). Pri volaní funkcie sú však odovzdané celé čísla (*int*), dôjde teda k typovej konverzii (*int* → *double*). Návratovou hodnotou funkcie je opätovne typ *double*, avšak táto hodnota je uložená pri volaní funkcie do premennej *c*, ktorej dátový typ je *int*. Dôjde teda opätovne k typovej konverzii z typu *double* na *int*.

```
double max(double x, double y);

int main(void)
{
    int a = 10;
    int b = -12;
    int c;
    c = max(a, b);

    return 0;
}

// definicia funkcie max()
...
```

Pozn. autora: Takáto typová konverzia, nie je až tak nebezpečná, ako keby tomu bolo opačne, teda v prípade, ak by skutočné parametre boli typu reálneho a formálne parametre typu celočíselného. Vtedy by mohlo dôjsť k strate informácií.

Ak potrebujeme vo funkcii zmeniť hodnoty skutočných parametrov – urobiť trvalú zmenu, musíme funkcii dodať namiesto hodnoty parametra jeho adresu. Potom už môže funkcia zmeniť hodnotu parametra, pretože bude meniť obsah na dodanej adrese v pamäti. Tento spôsob riešenia sa často nazýva **volanie odkazom**. Na prenos adresy sa využívajú smerníky. Viac o tejto problematike sa dočíta čitateľ v kapitole 4.

2.4 Príklad – funkcia na výpočet faktoriálu – volanie hodnotou

Príklad ilustruje výpočet faktoriálu s pomocou funkcie, kde parametre odovzdávame hodnotou. V matematike sa pojmom faktoriál kladného celého čísla *n* označuje súčin všetkých kladných celých čísel menších alebo rovných *n*. Zapisuje sa *n*!, pričom platí 0! = 1.

Napríklad: 5! = 1 . 2 . 3 . 4 . 5 = 120

Jedna z možných implementácií 2.4_Pr_1.c:

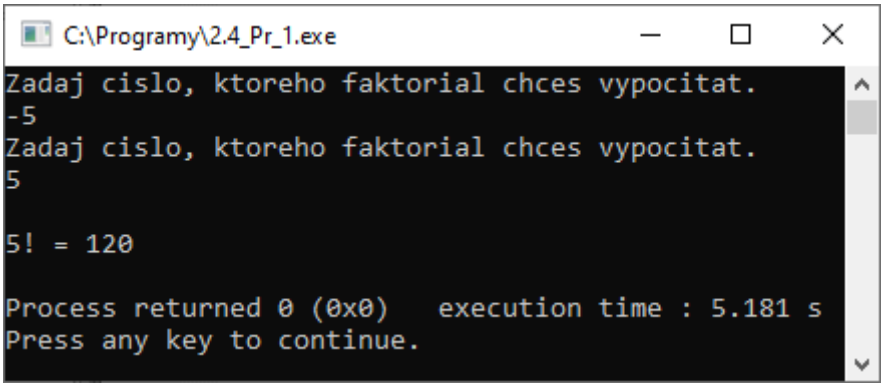
```c
#include <stdio.h>

int faktorial(int n);  // deklaracia funkcie

int main(void)
{
    int a;
    do
    {
        printf("Zadaj cislo, ktoreho faktorial chces vypocitat.\n");
        scanf("%d", &a);
    }
    while (a < 0);

    printf("\n%d! = %d \n", a, faktorial(a));

    return 0;
}

// definicia funkcie
int faktorial(int n)
{
    int i, fakt = 1;
    for (i = 1; i <= n; i++)
    {
        fakt = i * fakt;
    }
    return fakt;
}
```

Prototyp funkcie

Volanie funkcie

Formálny parameter *n* získa kópiu odovzdaného (skutočného) parametra *a*.

Definícia funkcie



Obr. 30 Konzolový výstup programu 2.4_Pr_1.c.

Ako prebieha vyhodnocovanie programu?

Vyhodnotenie každého programu sa začína od hlavnej funkcie, t. j. *main()*. V tomto prípade je používateľ vyzvaný k zadaniu jedného celého kladného čísla (realizované formátovaným načítaním premennej s pomocou funkcie *scanf()* – riadok 11), ktoré je uložené do premennej *a*, deklarovanej na riadku 7. Následne je volaná funkcia *faktorial(a)*, ktorej je táto hodnota premennej odovzdaná ako skutočný parameter. Tok riadenia skočí do funkcie, kde sa skutočná hodnota premennej *a* nakopíruje do lokálnej premennej *n* – formálneho parametra funkcie. Vo funkcii sa pracuje s touto lokálnou kópiou premennej, čiže k žiadnej zmene skutočného parametra *a* nemôže dôjsť. Na riadku 23 okrem deklarácie riadiacej premennej *i* na účely cyklu definujeme aj premennú *fakt*, ktorej hodnotu inicializujeme na 1. Cyklus s pevne určeným počtom opakovaní sa vykoná toľkokrát, koľko je hodnota premennej *n* a v každom cykle sa aktuálna hodnota premennej *fakt* vynásobí hodnotou riadiacej premennej *i*, ktorá sa po každom cykle navýši o 1. Po ukončení cyklu hodnotu premennej *fakt* vrátime s pomocou príkazu *return*, čím dôjde k ukončeniu funkcie. Tok riadenia sa vráti do miesta volania funkcie – riadok 15 a dôjde k vypísaniu formátovaných informácií na obrazovku. Následne program končí.

2.5 Pole ako parameter funkcie

Ak je jednorozmerné pole parametrom funkcie, tak skutočný parameter sa odovzdáva iba odkazom, teda s pomocou pointera. Napríklad v prípade funkcie, ktorá nájde minimum v zadanom poli, definícia funkcie by vyzerala:

```
int minimum(int pole[], int pocet)
{
    int i, min = pole[0];
    for (i = 1; i < pocet; i++)
       if (pole[i] < min)
         min = pole[i];
    return min;
}
```

Hlavička funkcie by mohla byť uvedená teda aj v tvare:

```
int minimum(int * pole, int pocet);
```

Volanie funkcie v tele programu by potom mohlo vyzerať:

```
#define POCET 100
int main(void)
{
  int pp[POCET];
  …
  x = minimum(pp, POCET); // volanie funkcie
  …
  return 0;
}
```

Pozn. autora: Pri volaní funkcie je odovzdaný len názov poľa (identifikátor), pričom vieme, že tento je zároveň ukazovateľom na prvý prvok poľa, t. j. odovzdáva sa adresa. Totožný príkaz by sme mohli napísať aj v prípade, ak by sme priamo špecifikovali adresu prvého prvku (tento zápis sa však často nepoužíva): `x = minimum(&pp[0], POCET);`

Obdobne tomu je aj pri dvojrozmernom poli. Definícia funkcie, ktorá vráti najväčšiu hodnotu dvojrozmerného poľa by mohla vyzerať:

```
float maximum(float p[][5], int riadky)
{
    float max = FLT_MAX;      // float max = p[0][0];
    int i, j;
    for (i = 0; i < riadky; i++)
       for (j = 0; j < 5; j++)
          if (p[i][j] > max)
                max = p[i][j];
    return max;
}
```

Pozn. autora: Ak by sme použili inicializáciu premennej *max*, ako je uvedené v komentári, tak by pri prvom vykonávaní tela vnoreného cyklu boli prvé prvky porovnávané samé so sebou. Z tohto dôvodu sme zvolili inicializáciu na konštantu *FLT_MAX*.

Pozn.: Počet prvkov u jednorozmerného poľa nemusí byť uvedený, lebo kompilátor si ho sám doplní. U dvojrozmerných polí nemusí byť uvedený počet riadkov, ale počet stĺpcov sa musí uviesť.

2.6 Globálne verzus lokálne premenné

Vychádzajúc z publikácie Herouta (2010), deklarácie, resp. definície premenných sa môžu vyskytovať buď vonku (mimo) funkcie, vtedy hovoríme o **globálnej premennej**, alebo vo vnútri funkcie a vtedy takúto premennú nazývame **lokálna premenná.**

Globálne premenné sú viditeľné od miesta ich deklarácie/definície do konca zdrojového súboru. Existujú teda od spustenia programu až po ukončenie programu. Keďže globálnu premennú môže meniť ľubovoľná funkcia, snažíme sa vyhýbať používaniu takýchto premenných. Nie je totižto jednoduché zistiť v prípade problémov, ktorá funkcia zmenila obsah tejto premennej. Ich používaním sa zároveň znižuje modularita programu.

Statická globálna premenná má rovnaký význam ako globálna premenná, ale je špecifikovaná kľúčovým slovom *static*, ktoré obmedzuje jej viditeľnosť len na časť modulu (súboru) v ktorej je deklarovaná/definovaná (toto platí v prípade, ak používame oddelený preklad programu).

Lokálna premenná existuje a je viditeľná iba zvnútra funkcie alebo bloku, v ktorom je deklarovaná/definovaná. Rozsah platnosti lokálnej premennej je od miesta jej deklarácie/definície až po koniec funkcie/bloku, v ktorej je premenná deklarovaná/definovaná. Prekladač ju uloží do zásobníka (*stacku*). S ukončením spracovávania funkcie/bloku lokálna premenná zaniká a pri opätovnom vstupe do funkcie je jej hodnota nedefinovaná (ak nie je inicializovaná). Ak vo funkcii zadefinujeme premennú s rovnakým názvom, ako je názov globálnej premennej, prekryjeme globálnu premennú definíciou lokálnej premennej a teda v tejto funkcii už nemáme prístup ku globálnej premennej. Neplatí to, ak vo funkcii zadefinujeme premennú s rovnakým názvom, ako je názov inej lokálnej premennej. V takomto prípade ide o dve nezávislé premenné s rovnakým názvom.

Statická lokálna premenná existuje od vstupu do funkcie, v ktorej je definovaná do skončenia programu, pričom je viditeľná iba z funkcie, v ktorej je definovaná. Prekladač ju ukladá do dátovej oblasti (*data area*) pamäti. Lokálne premenné nie sú automaticky inicializované, čiže ich hodnota je vždy náhodná. Globálne premenné sú implicitne inicializované na nulu, teda *int* premenné majú hodnotu 0, *float* hodnotu 0.0, *char* hodnotu '\0', a pod. Je však dobrým zvykom nespoliehať sa na túto skutočnosť a v prípade potreby inicializáciu uviesť.

Príklad 1:

```
float pocet;          // globalna premenna programu
void funkcia()
{
    float pocet;      // lokalna premenna funkcie, zakryje globalnu premennu
    ...
}
int main(void)
{
    // lokalne premenne programu, pocet je ina premenna ako vo funkcii
    float pocet, sucet;
    ...
}
```

Príklad 2:

```
int p_1;              // deklaracia globalnej premennej
void funkcia_A()
{
    ...               // tu je viditelna globalna premenna p_1
}
int p_2;              // deklaracia globalnej premennej
void funkcia_B()
{
    ...               // tu su viditelne globalne premenne p_1 aj p_2
}
void funkcia_C()
{
    ...               // tu su viditelne globalne premenne p_1 aj p_2
}
```

Príklad 3:

```
int p_1, p_2;          // deklaracie globalnych premennych p_1 a p_2
void funkcia_A()
{
    int p_1;           // deklaracia lokalnej premennej zakryje globalnu p_1
     ...               // tu je viditelna lokalna p_1 a globalna p_2
}
int p_3;               // deklaracia globalnej premennej p_3

void funkcia_B()
{
    int p_1, p_2;      // deklaracia lokalnych premennych (zakryvaju globalne)
                       // tu su viditelne lokalne p_1 a p_2 a globalna p_3
}
```

2.6.1 Pamäťové triedy – modifikátory pamäťovej triedy

Okrem rôznych dátových typov môžu byť premenné uvedené aj v rôznych pamäťových triedach (Herout, 2010). Tie určujú, v ktorej časti pamäti bude premenná kompilátorom umiestnená a tiež kde všade bude premenná viditeľná. Rozširujú tak možnosti viditeľnosti premenných, ktoré boli doteraz len globálne a lokálne.

Jazyk *C* rozoznáva tieto modifikátory pamäťovej triedy: *auto*, *extern*, *static* a *register*.

Trieda *auto*

O týchto premenných sa často hovorí ako o automatických. Je to implicitná pamäťová trieda pre lokálne premenné. Ak je premenná definovaná vo vnútri funkcie bez určenia pamäťovej triedy, je jej implicitný typ práve typ *auto* a premenná je uložená v zásobníku. Premenná typu *auto* existuje od vstupu do funkcie a zaniká pri výstupe z funkcie. Pri každom vstupe do funkcie má náhodnú hodnotu, čiže nie je implicitne inicializovaná na 0 a ani si nezachováva svoju hodnotu medzi dvomi volaniami funkcie. Napr.:

```
funkcia()
{
  auto int i;
  auto int j = 1;
  …
}
```

je to isté, ako

```
funkcia()
{
  int i;
  int j = 1;
  …
}
```

Trieda *extern*

Je to implicitná pamäťová trieda pre globálne premenné. Tieto premenné sú uložené v dátovej oblasti. Kľúčové slovo *extern* sa používa pri oddelenom preklade súborov, kde je potrebné, aby dva alebo viac súborov sa delili o tú istú premennú. Táto globálna premenná je v jednom súbore

definovaná bez kľúčového slova *extern* a vo všetkých ostatných musí byť deklarovaná s použitím *extern*, napr.:

súbor S1.c
```
int rozmer;      // deklaracia
```

súbor S2.c
```
extern int rozmer;   // deklaracia
```

Trieda *static*

Pre túto pamäťovú triedu neexistuje žiadna implicitná definícia. Kľúčové slovo *static* musí byť pri deklarácii/definícii vždy uvedené. Premenné tejto triedy sú uložené v dátovej oblasti. Má dve oblasti použitia:

1. Pre globálne premenné alebo funkcie, čo má taký význam, že sú viditeľné iba v module (súbore) v ktorom sú definované.
2. Pre lokálne premenné, kde pamäťovú triedu *static* využívajú lokálne premenné, ktoré si ponechávajú svoju hodnotu aj medzi jednotlivými volaniami tejto funkcie. Napr.:

```
void funkcia(void)
{
  int x = 2;
  static int i = 1;
  printf("Funkcia bola volana %d-krat, x = %d\n", i, x);
  i++;
  x++;
}
```

Statická lokálna premenná existuje od prvého volania prislúchajúcej funkcie až do doby ukončenia programu, ale ako lokálna premenná nie je samozrejme prístupná zvonku funkcie. To znamená, že premenná *x* je lokálna automatická premenná a premenná *i* je lokálna statická premenná. Vždy, keď je *funkcia()* zavolaná, môže byť miesto pre premennú *x* alokované v inej časti zásobníku a vždy je explicitne inicializovaná na hodnotu 2. Premenná *i* je inicializovaná na 1 pri prvom vstupe do funkcie *funkcia()* a svoju hodnotu si zachováva aj medzi jednotlivými volaniami, čiže inicializácia na 1 sa pri ďalších volaniach funkcie *funkcia()* už neuskutoční. Napr. pre volanie:

```
for (j = 0; j < 3; j++) funkcia();
```

bude výstup:

Funkcia bola volana 1-krat, x = 2

Funkcia bola volana 2-krat, x = 2

Funkcia bola volana 3-krat, x = 2

Pozn.: Ak potrebujeme viac statických premenných jedného typu, je vhodné definovať každú premennú samostatným príkazom. Niektoré prekladače totiž deklaráciu:

```
static int i, j;
```

spracujú tak, že statická premenná bude iba premenná *i* a premenná *j* bude mať implicitnú pamäťovú triedu *auto*. Obe premenné samozrejme budú typu *int*.

Trieda *register*

Pretože jazyk *C* je jazyk strednej úrovne, programátor môže požadovať, aby niektorá premenná nebola uložená v operačnej pamäti, ale iba v registri počítača. To má výhodu omnoho rýchlejšieho prístupu k premennej a teda aj rýchlejšieho programu. Označenie premennej ako *register* neviaže túto premennú na určitý konkrétny register počítača – to plne záleží na prekladači. Deklarácia premennej:

```
register int i;
```

znamená iba to, že premenná môže byť uložená do registra, ak je nejaký voľný a ak to je z rôznych systémových dôvodov možné.

Ak označíme všetky použité premenné ako *register*, tak skutočne v registroch budú umiestnené iba niektoré. Z toho vyplýva, že ako registrovú premennú je vhodné označiť napr. premennú jednoduchého cyklu, alebo často používaný formálny parameter. U registrových premenných sa nerobí žiadna implicitná inicializácia a používajú sa výhradne ako lokálne premenné, t. j. nie je možné mať globálnu premennú typu *register*.

Pozn.: Pre definíciu viacerých premenných rovnakého typu v pamäťovej triede *register* platí to isté, ako pre *static*, odporúčame používať definíciu každej premennej samostatne. Dôležitou skutočnosťou je, že nie je možné získať adresu registrovej premennej.

2.6.2 Alokácia pamäte

Každá premenná musí mať počas svojej existencie pridelený pamäťový priestor, ktorý veľkosťou zodpovedá typu premennej. Meno premennej (identifikátor) je vlastne symbolická adresa tohto pamäťového priestoru. Akcia, ktorá vyhradzuje pamäťový priestor sa nazýva **alokácia**, pričom rozoznávame dva základné typy alokácie – statická a dynamická (Herout, 2010).

2.6.2.1 Statická alokácia

Statická alokácia sa používa častejšie a to vtedy, ak vieme prekladaču dopredu presne povedať, aké budeme mať v programe pamäťové nároky. Prekladač sám určí požiadavky na pamäť na

zásobníku (*stacku*) pre všetky definované premenné a *linker* (*loader*) alokuje túto pamäť v čase začiatku spustenia programu.

V priebehu spustenia programu sa nerealizuje žiadna manipulácia s prideľovaním pamäte. Existencia staticky alokovaných globálnych premenných je od začiatku programu až do jeho konca, teda do momentu kým nie je odovzdané riadenie späť operačnému systému, ktorý všetky alokácie zrealizované pred spustením programu zruší.

Statická alokácia je účinná, ale v niektorých prípadoch nedostačujúca. Napríklad pri rekurzívnom volaní funkcie. Každé rekurzívne volanie totiž spôsobí novú alokáciu potrebnej pamäte a prekladač nie je schopný určiť, koľkokrát bude funkcia rekurzívne volaná. Prípadne, ak potrebujeme do pamäte načítať obsah celého súboru, nemôžeme v dobe prekladu vedieť, či to bude krátky alebo dlhý súbor (Herout, 2010).

2.6.2.2 Dynamická alokácia na hromade (heape)

Alokáciu realizujeme s pomocou špeciálneho príkazu (bližšie v kapitole 4), kde až v priebehu vykonávania programu špecifikujeme veľkosť požadovanej pamäte. Táto pamäť nie je pomenovaná (nemá identifikátor) a pristupuje sa k nej s pomocou pointera.

2.6.2.3 Alokácia pamäte v zásobníku (stacku) pri volaní funkcií

Tento spôsob alokácie pamäte sa realizujeme automaticky pri volaní funkcie s využitím informácií odovzdaných prekladačom. Pre väčšinu lokálnych premenných (definovaných vo vnútri funkcie) je použitý tento druh alokácie. Existencia lokálnych premenných sa začína pri vstupe do funkcie a končí pri výstupe z tejto funkcie. Potom môže byť pamäť použitá vo funkcii pre túto premennú na iné účely. Ak je však funkcia volaná znovu, premenná má pri vstupe do funkcie nedefinovanú hodnotu. Z toho vyplýva, že premenná, ktorá je síce potrebná iba vo vnútri funkcie, ale musí si uchovať svoju hodnotu medzi volaniami tejto funkcie (statická lokálna premenná), nemôže mať pamäť alokovanú v zásobníku (Herout, 2010).

2.7 Direktíva *#define* a makrá

#define je (podobne ako *#include*) direktívou preprocesora, ktorá slúži na definíciu symbolických konštánt a makier. Konštantu sme doteraz deklarovali ako premennú s modifikátorom *const*:

```
const float PI = 3.14f;
```

V jazyku *C* však môžeme konštantnú hodnotu definovať aj s pomocou symbolickej konštanty, ktorú vytvoríme prostredníctvom direktívy *#define*:

```
#define MenoKonstanty HodnotaKonstanty
```

Makrá môžu byť:

- bez parametrov,
- s parametrami.

Makrá bez parametrov sú skôr známe pod menom *symbolické konštanty*. Používajú sa pomerne často, pretože nahrádzajú v programe konštanty. Väčšinou sú definované na začiatku programu (modulu). Náhrada konštanty skutočnou hodnotou sa nazýva rozvoj (expanzia) makra alebo tiež substitúcia makra. Názvy konštánt sa píšu veľkými písmenami. Hodnota je od mena oddelená aspoň jednou medzerou. Nepíše sa za nimi bodkočiarka „;“.

```
#include <stdio.h>
#define MAX 1000
#define PI 3.14f
#define DVE_PI (2*PI) // ak je symbolickou konstantou vyraz odporuca sa uzatvorit do ()
#define MOD %
#define MENO_SUBORU "textak.txt"
```

Makrá sú nahrádzané v dobe prekladu svojim originálom. Zrušenie definície makra:

```
#undef meno_makra
```

Napr.:

```
#undef MAX
```

Niekedy sa makro používa ako skrytá časť programu, napr.:

```
#define ERROR {printf("Chyba v datach.\n");}
```

Pri použití nesmie byť toto makro ukončené bodkočiarkou.

```
if (x == 0)
  ERROR
else
 y = y / x;
```

Makrá s parametrami sa niekedy nazývajú aj *vkladané funkcie* (*inline functions*), pretože na rozdiel od skutočných funkcií sa makrá s parametrami nevolajú, ale pred predkladom nahradí preprocesor meno makra konkrétnym textom. Môže nahradiť funkciu, ktorá je veľmi krátka, alebo realizuje jednoduchý výpočet. Takéto riešenie často vedie k optimalizácii rýchlosti programu, pretože odpadajú administratívne operácie, ktoré sa viažu na použitie klasickej funkcie, t. j. odovzdanie parametrov, uchovanie návratovej hodnoty, skok do funkcie, návrat z funkcie do miesta volania a pod. Samozrejme je to na úkor rozsahu programu. To znamená, že praktické použitie je len v prípade krátkej akcie, kedy by administratíva funkcie trvala

porovnateľnú dobu s vlastným výpočtom funkcie. Netreba však zabudnúť, že pri makrách s parametrami nie je možné používať rekurziu na rozdiel od funkcií. Syntax makra s parametrami je nasledujúca:

```
#define meno_makra(arg1, arg2, ..., argN) hodnota makra
```

Na rozdiel od makier bez parametrov sa makrá s parametrami píšu malými písmenkami (ako u funkcií). Syntax volania makra je:

```
meno_makra(arg1, ..., argN)
```

Pozor! Medzi menom makra a otváracou okrúhlou zátvorkou nesmie byť medzera. Argumenty by tak boli považované za hodnotu makra (Herout, 2010).

Príklady použitia:

```
// test velkeho pismena
#define je_velke(c) ((c) >= 'A' && (c) <= 'Z'))
// volanie v zdrojovom subore
ch = je_velke(ch) ? ch + ('a' - 'A') : ch;
// rozvinutie makra po spracovani preprocesorom
ch = ((ch) >= 'A' && (ch) <= 'Z') ? ch + ('a' - 'A') : ch;

// vypocet druhej mocniny cisla x
#define na_2(x) ((x) * (x))
```

Pozor! Tiež si všimnite, že tak ako argument aj hodnota makra sú uzatvorené v okrúhlych zátvorkách. Ak by sme to tak nespravili, mohlo by byť makro nesprávne rozvité a vznikali by chyby. Napr. definícia:

```
#define na_2(x) x * x
```

po volaní:

```
na_2(a + b)
```

by sa rozvinula do:

```
a + b * a + b
```

Správna definícia teda má byť:

```
#define na_2(x) ((x) * (x))
```

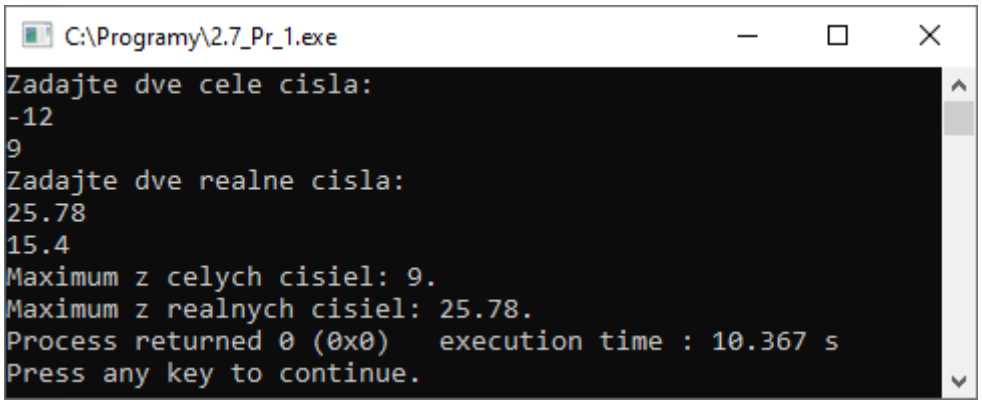
ktorá sa rozvinie do:

```
((a + b) * (a + b))
```

Ukážka definície makra na určenie maximálnej hodnoty z dvoch zadaných hodnôt 2.7_Pr_1.c:

```
1  #include <stdio.h>
2  #include <conio.h>
3  #define MAX(x, y) ((x) > (y) ? (x) : (y))    // makro s parametrami
4
5  int main(void)
6  {
7      int a, b;
```

```
8       float a1, b1;
9       printf("Zadajte dve cele cisla:\n");
10      scanf("%d %d", &a, &b);
11      printf("Zadajte dve realne cisla:\n");
12      scanf("%f %f", &a1, &b1);
13      printf("Maximum z celych cisiel: %d.\n", MAX(a, b));     // volanie makra
14      printf("Maximum z realnych cisiel: %.2f.", MAX(a1, b1));  // volanie makra
15
16      return 0;
17  }
```



Obr. 31 Konzolový výstup programu 2.7_Pr_1.c.

V jazyku *C* existujú aj **preddefinované makrá** (obsah hlavičkových súborov *stdio.h* a *ctype.h*). Makrá definované v súbore *ctype.h* sú delené do dvoch skupín:

1. makrá na určenie typu znaku – tieto sa začínajú písmenami *is*, napr. *isdigit(c)* testuje, či *c* je číslo, ak áno, tak vráti hodnotu *1* (*true*), inak *0* (*false*)
2. makrá na konverziu znaku – tieto začínajú písmenami *to*, napr. *b = tolower(c)* realizuje konverziu znaku uloženého v *c* na malé písmeno, znak uložený v *c* necháva bezo zmeny (Herout, 2010).

2.8 Príklad – výpočet koreňov kvadratickej rovnice s pomocou funkcie

V prvom diele tejto učebncie, v kapitole 1.7.1 sme riešili problém výpočtu koreňov kvadratickej rovnice. Uviedli sme aj riešenie s pomocou procedúry, avšak toto riešenie nebolo dostatočne univerzálne. Na návrh univerzálneho riešenia sme potrebovali nadobudnúť vedomosti ohľadom odovzdávania parametrov odkazom. S pomocou návratovej hodnoty funkcie vieme vrátiť len jednu hodnotu, čo v tomto prípade nie je dostatočné (kvadratická rovnica môže mať dva rôzne reálne korene). Návrh funkcie na výpočet koreňov kvadratickej rovnice v množine *R* navrhujeme:

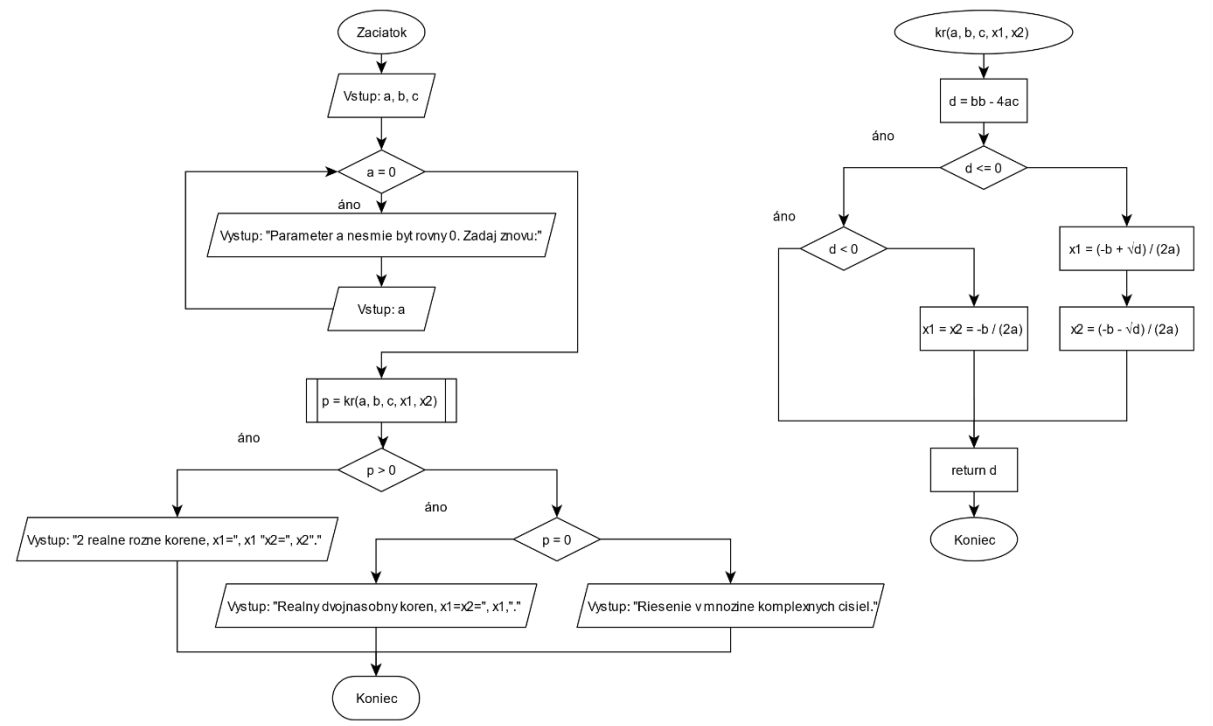
```
float kr(float a, float b, float c, float *x1, float *x2);
```

kde s pomocou návratovej hodnoty odovzdáme hodnotu diskriminantu, podľa ktorého vieme rozhodnúť, aké je riešenie kvadratickej rovnice a vypísať v tomto prípade adekvátne výsledky.

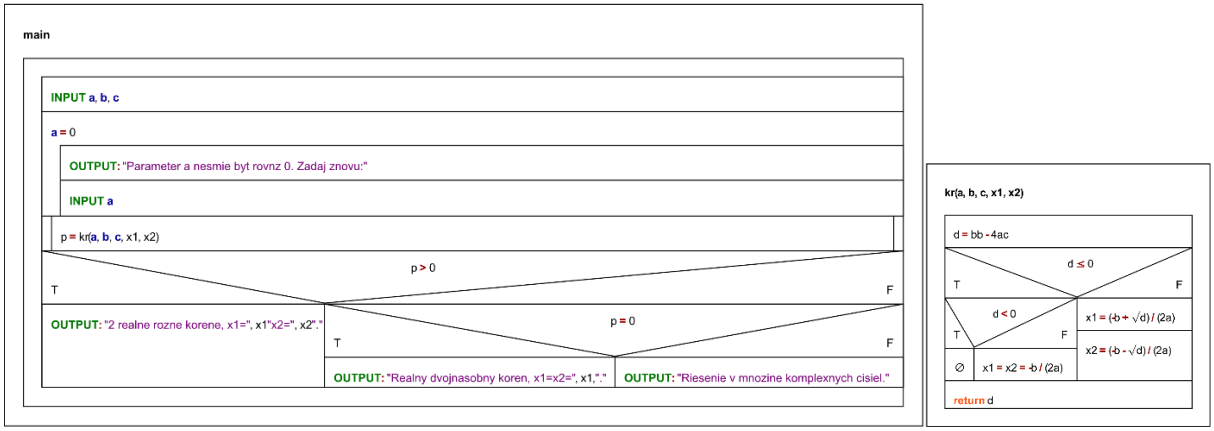
Koeficienty kvadratickej rovnice stačí odovzdať hodnotou. Slúžia len k výpočtu a nie je cieľom modifikovať ich hodnoty. Parametre reprezentujúce korene kvadratickej rovnice musia byť odovzdané odkazom, aby zmena bola trvalá a mohli sme v *maine* s nimi pracovať podľa potreby. V tomto prípade ich len vypisujeme používateľovi.

Jedna z možných implementácií 2.8_Pr_1.c:

```c
#include <stdio.h>
#include <math.h>

float kr(float a, float b, float c, float *x1, float *x2);   // prototyp funkcie

int main(void)
{
    float a, b, c, x1, x2, p;
    printf("Zadaj koeficienty kvadratickej rovnice a, b, c: \n");
    scanf("%f %f %f", &a, &b, &c);
    while (a == 0)
    {
        printf("Toto nie je spravny parameter; a musi byt rozne od 0.\n");
        printf("Znovu zadaj parameter a:\n");
        scanf("%f", &a);
    }
    p = kr(a, b, c, &x1, &x2);  // volanie funkcie s navratovou hodnotou
    if (p > 0)
        printf("Dva realne rozne korene x1 = %.2f, x2 = %.2f.\n", x1, x2);
    else if (p == 0)
        printf("Realny dvojnasobny koren x1 = x2 = %.2f.\n", x1);
    else
        printf("Riesenie v mnozine komplexnych cisiel.\n");

    return (0);
}

// definicia funkcie
float kr(float a, float b, float c, float *x1, float *x2)
{
    float d;
    d = b * b - 4 * a * c;
    if (d <= 0)
        if (d < 0)
            return d;
        else
        {
            *x1 = *x2 = -b / (2 * a);
            return d;
        }
    else
    {
        *x1 = (-b + sqrt(d)) / (2 * a);
        *x2 = (-b - sqrt(d)) / (2 * a);
        return d;
    }
}
```



Obr. 32 Vývojový diagram na riešenie kvadratickej rovnice s pomocou funkcie.



Obr. 33 NS diagram na riešenie kvadratickej rovnice s pomocou funkcie.

2.9 Príklad – súčin dvoch matíc

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte riešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém súčinu dvoch matíc.

Vstup (vstupné premenné):

r1, s1, r2, s2 – rozmery matice *A* a matice *B*, počet riadkov a stĺpcov

A[r1][s1] – vstupná matica *A*

B[r2][s2] – vstupná matica *B*

Výstup (výstupné premenné):

C[r1][s1] – výstupná matica, daná súčinom matice *A* a matice *B*

Vstupné podmienky:

$r1, s1, r2, s2 \in \mathrm{Z}^{+} \wedge r1 = s2 \wedge A[i][j] \in \mathrm{R} \wedge B[i][j] \in \mathrm{R}$

Výstupné podmienky:

$C[i][j] \in \mathrm{R} \wedge C[i][j] = A[i, 1]B[1, j] + A[i, 2]B[2, j] + \ldots + A[i, n]B[m, j]$

Pomocné premenné:

i – riadiaca premenná, $i \in \langle 0, r1)$ alebo $i \in \langle 0, r2)$

j – riadiaca premenná, $j \in \langle 0, s1)$ alebo $j \in \langle 0, s2)$

k – riadiaca premenná použitá pri výpočte súčinu matíc, $k \in \langle 0, s1)$

MAX – konštanta na určenie maximálnej veľkosti matice (pre úplnosť uvádzame aj túto premennú. Je však zrejmé, že jej použitiu sa vieme vyhnúť. V takom prípade by sme však pri implementácii museli využívať dynamické pole, ktoré bude vysvetlené až v kapitole 4.6.)

Rozbor úlohy: Násobenie matíc môže prebiehať len vtedy, ak je počet stĺpcov ľavej matice rovnaký ako počet riadkov pravej matice. Ak *A* je $m \times n$ matica a *B* je $n \times r$ matica, tak ich maticový produkt *AB* má rozmery $m \times r$ (*m* počet riadkov (ako v prvej matici) × *r* počet stĺpcov (ako v druhej matici)). Výsledná hodnota na pozícií *[i, j]* je:

$(AB)[i, j] = A[i, 1]B[1, j] + A[i, 2]B[2, j] + \ldots + A[i, n]B[m, j]$

pre každé *i* a *j* (Matica, 2020). Napríklad:

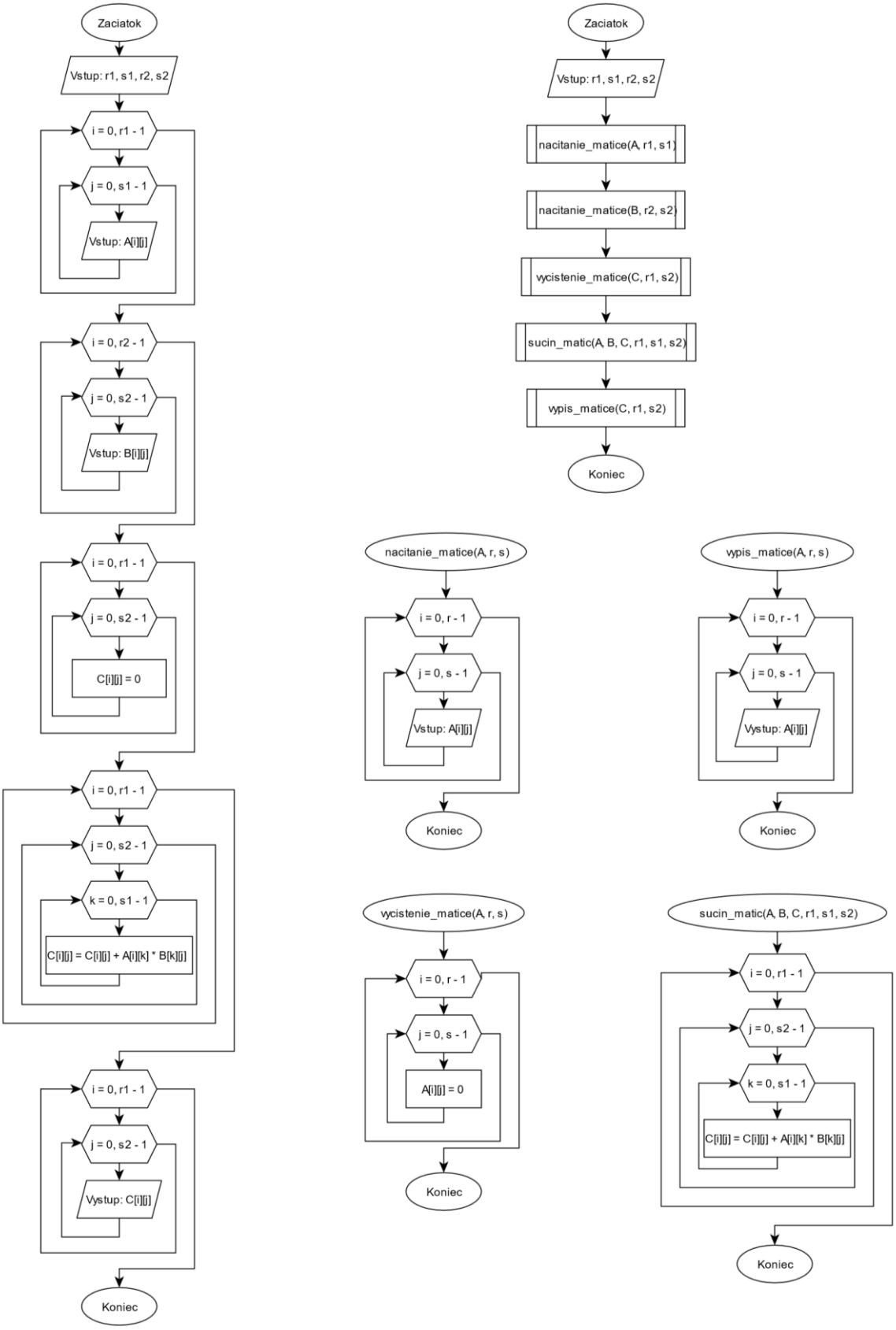
$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1 \times 3 + 0 \times 2 + 2 \times 1) & (1 \times 1 + 0 \times 1 + 2 \times 0) \\ (-1 \times 3 + 3 \times 2 + 1 \times 1) & (-1 \times 1 + 3 \times 1 + 1 \times 0) \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Pričom nie je jedno, v akom poradí sa matice násobia, napríklad:

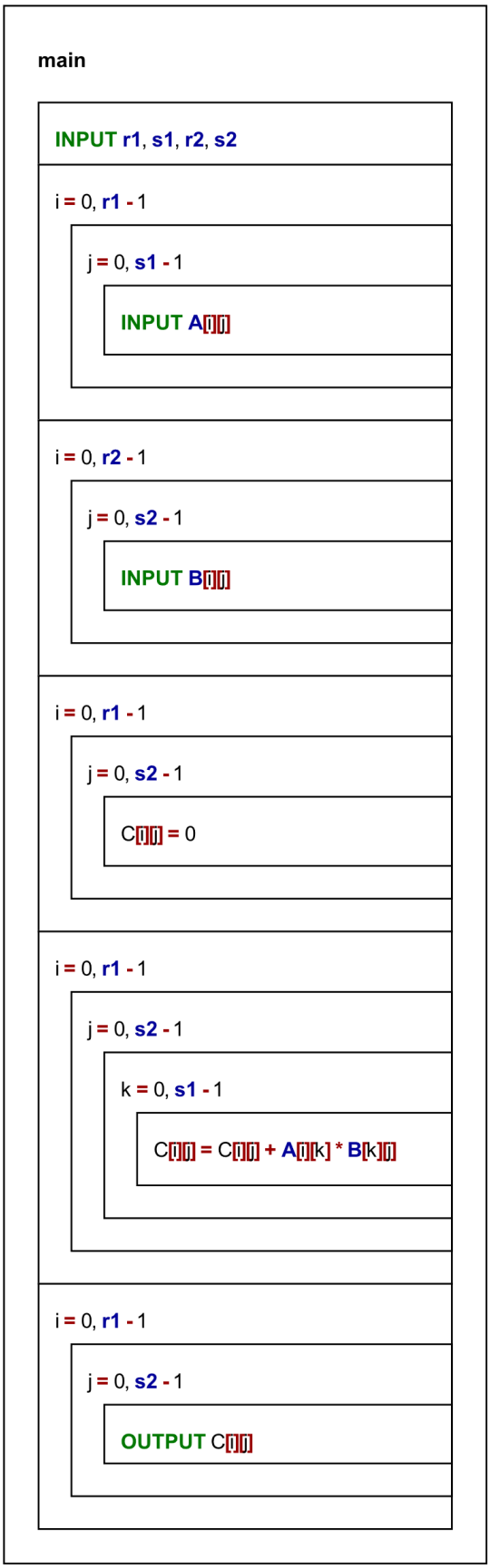
$$\begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} (3\times 1+1\times -1) & (3\times 0+1\times 3) & (3\times 2+1\times 1) \\ (2\times 1+1\times -1) & (2\times 0+1\times 3) & (2\times 2+1\times 1) \\ (1\times 1+0\times -1) & (1\times 0+0\times 3) & (1\times 2+0\times 1) \end{bmatrix} = \begin{bmatrix} 2 & 3 & 7 \\ 1 & 3 & 5 \\ 1 & 0 & 2 \end{bmatrix}$$

Môžete vidieť, že dokonca ani rozmer výsledných matíc nemusí byť rovnaký pri vymenenom poradí ich súčinu. Je zrejmé, že je možné násobiť matice rôznych rozmerov. Preto, v prvom rade potrebujeme poznať rozmery vstupných matíc a otestovať, či je možné matice takýchto rozmerov násobiť. Až následne má zmysel sa postarať o ich naplnenie. Na načítanie/výpis prvkov matice je možné použiť dva vnorené cykly s pevným počtom opakovaní. Na súčin matíc budeme potrebovať už tri vnorené cykly. Z dôvodu, že prvky matice *C* vznikajú súčtom súčinov jednotlivých prvkov vstupných matíc, je dôležité aby výstupná matica *C* bola pred operáciou súčinu vynulovaná. V opačnom prípade by sme nedostávali korektné výsledky.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 34 Vývojový diagram na súčin matíc bez funkcií (riešenie vľavo) a s funkciami (riešenie vpravo).



Obr. 35 NS diagram na súčin matíc bez funkcií.

Overenie správnosti algoritmu bez funkcií pre takéto vstupy:

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1\times 3+0\times 2+2\times 1) & (1\times 1+0\times 1+2\times 0) \\ (-1\times 3+3\times 2+1\times 1) & (-1\times 1+3\times 1+1\times 0) \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Pozn. autora: Predpokladám, že čitateľ má už dostatočne osvojenú prácu s maticami a preto niektoré kroky pri overovaní správnosti algoritmu v nasledujúcej tabuľke uvádzam v stručnejšej podobe.

***r1* = 2, *s1* = 3, *r2* = 3, *s2* = 2**						
Hodnoty sledovaných premenných						**Popis**
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	
						Načítanie rozmerov matíc *A* a *B* → *r1* = 2, *s1* = 3, *r2* = 3 a *s2* = 2
0						Inicializácia riadiacej premennej cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1
	0					Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
			1			Načítanie prvku matice *A[0][0]*
	1					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
			0			Načítanie prvku matice *A[0][1]*
	2					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
			2			Načítanie prvku matice *A[0][2]*
	3					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → koniec vnútorného cyklu
1						Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 1 ≤ 1
	0					Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky vnútorného cyklu → 0 ≤ 2
			−1			Načítanie prvku matice *A[1][0]*
	1					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 2
			3			Načítanie prvku matice *A[1][1]*

	2					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 2
			1			Načítanie prvku matice *A[1][2]*
	3					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 3 ≤ 2 → koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 2 ≤ 1 → koniec vonkajšieho cyklu
0						Inicializácia riadiacej premennej cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2
	0					Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1
				3		Načítanie prvku matice *B[0][0]*
	1					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 1
				1		Načítanie prvku matice *B[0][1]*
	2					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
1						Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 1 ≤ 2
	0					Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky vnútorného cyklu → 0 ≤ 1
				2		Načítanie prvku matice *B[1][0]*
	1					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 1
				1		Načítanie prvku matice *B[1][1]*
	2					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 2 ≤ 2
	0					Inicializácia riadiacej premennej cyklu *j* = 0 a testovanie splnenia podmienky vnútorného cyklu → 0 ≤ 1
				1		Načítanie prvku matice *B[2][0]*

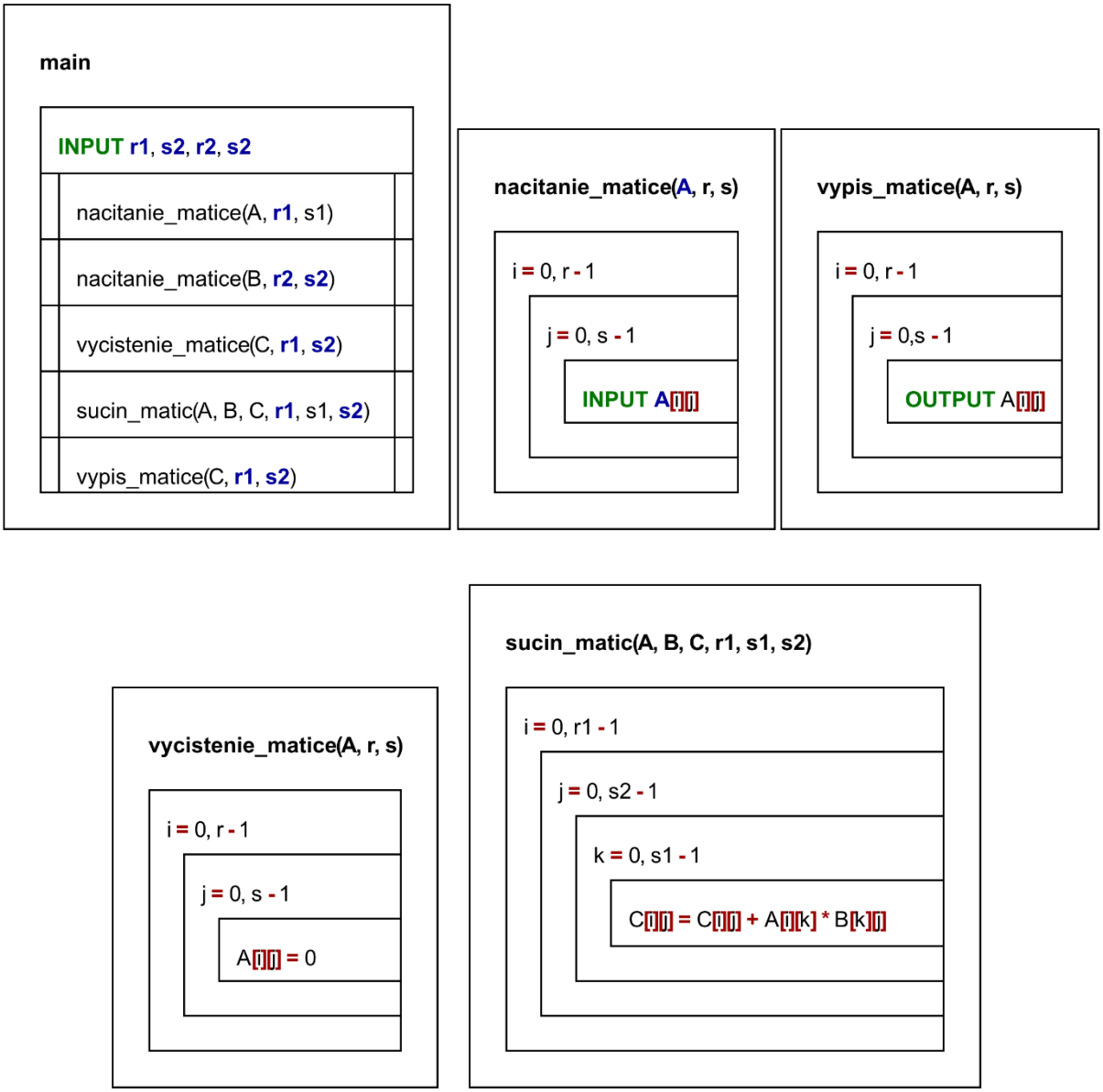
	1					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 1 ≤ 1
				0		Načítanie prvku matice *B[2][1]*
	2					Inkrementácia riadiacej premennej *j* a opätovné testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
3						Inkrementácia riadiacej premennej *i* a opätovné testovanie splnenia podmienky vonkajšieho cyklu → 3 ≤ 2 → koniec vonkajšieho cyklu
0	0				0	Inicializácia riadiacej premennej vonkajšieho cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 Následne vynulovanie prvku matice *C[0][0]*
	1				0	Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1 Následne vynulovanie prvku matice *C[0][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1→ koniec vnútorného cyklu
1	0				0	Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1, inicializáica riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 Následne vynulovanie prvku matice *C[1][0]*
	1				0	Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1 Následne vynulovanie prvku matice *C[1][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1→ koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1→ koniec vonkajšieho cyklu
0	0	0				Inicializácia riadiacej premennej prvého cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1, inicializácia riadiacej premennej druhého cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 a inicializácia riadiacej premennej tretieho cyklu *k* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2 → realizácia výpočtu *C[0][0] = C[0][0] + A[0][0] . B[0][0]* = 0 + 1 . 3 +

		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[0][1] \,.\, B[1][0] = + 0 \,.\, 2 +$
		2			5	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[0][2] \,.\, [2][0] = + 2 \,.\, 1 = 5$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
0	1	0				Inkrementácia riadiacej premennej druhého cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1$ a inicializácia riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 2 \rightarrow$ realizácia výpočtu $C[0][1] = C[0][1] + A[0][0] \,.\, B[0][1] = 0 + 1 \,.\, 1 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[0][1] \,.\, B[1][1] = + 0 \,.\, 1 +$
		2			1	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[0][2] \,.\, B[2][1] = + 2 \,.\, 0 = 1$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	2					Inkrementácia riadiacej premennej druhého cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec druhého cyklu
1	0	0				Inkrementácia riadiacej premennej prvého cyklu $i = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1,$ inicializácia riadiacej premennej druhého cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 1$ a inicializácia riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 2 \rightarrow$ realizácia výpočtu $C[1][0] = C[1][1] + A[1][0] \,.\, B[0][0] = 0 + (-1) \,.\, 3 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[1][1] \,.\, B[1][0] = + 3 \,.\, 2 +$

		2			4	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[1][2] \,.\, B[2][0] = + 1 \,.\, 1 = 4$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	1	0				Inkrementácia riadiacej premennej druhého cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1$ a inicializácia riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 2 \rightarrow$ realizácia výpočtu $C[1][1] = C[1][1] + A[1][0] \,.\, B[0][1] = 0 + (-1) \,.\, 1 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[1][1] \,.\, B[1][1] = + 3 \,.\, 1 +$
		2			2	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[1][2] \,.\, B[2][1] = + 1 \,.\, 0 = 2$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	2					Inkrementácia riadiacej premennej druhého cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec druhého cyklu
2						Inkrementácia riadiacej premennej prvého cyklu $i = 2$ a testovanie splnenia podmienky cyklu $2 <= 1 \rightarrow$ koniec prvého cyklu
0	0					Inicializácia riadiacej premennej prvého cyklu $i = 0$ a testovanie splnenia podmienky cyklu $\rightarrow \ 0 \leq 1$ a inicializácia riadiacej premennej druhého cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow \ 0 \leq 1 \rightarrow$ Výstup na obrazovku prvku $C[0][0]$
	1					Inkrementácia riadiacej premennej vnútorného cyklu $j = 1$ a testovanie splnenia podmienky cyklu $1 \leq 1 \rightarrow$ Výstup na obrazovku prvku $C[0][1]$
	2					Inkrementácia riadiacej premennej vnútorného cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec vnútorného cyklu
1	0					Inkrementácia riadiacej premennej vonkajšieho cyklu $i = 1$ a testovanie splnenia podmienky cyklu $1 \leq 1$,

						inicializácia riadiacej premennej vnútorného cyklu $j = 0$ a testovanie splnenia podmienky cyklu $0 \leq 1 \rightarrow$ Výstup na obrazovku prvku *C[1][0]*
	1					Inkrementácia riadiacej premennej vnútorného cyklu $j = 1$ a testovanie splnenia podmienky cyklu $1 \leq 1 \rightarrow$ Výstup na obrazovku prvku *C[1][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej vonkajšieho cyklu $i = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec vonkajšieho cyklu (koniec programu)

Algoritmus riešenia – grafická reprezentácia s pomocou NS diagramu s použitím funkcií:



Obr. 36 NS diagram na súčin matíc s funkciami.

Overenie správnosti algoritmu s funkciami:

***r1* = 2, *s1* = 3, *r2* = 3, *s2* = 2**						
Načítanie rozmerov matíc *A* a *B* → *r1* = 2, *s1* = 3, *r2* = 3 a *s2* = 2						
Volanie funkcie *nacitanie_matice(A, r1, s1)*						
Odovzdanie adresy pamäťového priestoru pre maticu *A* referenčnej premennej funkcie a vytvorenie lokálnych kópií vo funkcii *r* = *r1* = 2, *s* = *s1* = 3						
Hodnoty sledovaných premenných						**Popis**
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	
0	0		1			Inicializácia riadiacej premennej vonkajšieho cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2 → načítanie prvku matice *A[0][0]*
	1		0			Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 2 → načítanie prvku matice *A[0][1]*
	2		2			Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 2 → načítanie prvku matice *A[0][2]*
	3					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 3 a testovanie splnenia podmienky cyklu → 3 ≤ 2 → koniec vnútorného cyklu
1	0		−1			Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2 → načítanie prvku matice *A[1][0]*
	1		3			Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 2 → načítanie prvku matice *A[1][1]*
	2		1			Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 2 → načítanie prvku matice *A[1][2]*
	3					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 3 a testovanie splnenia podmienky cyklu → 3 ≤ 2 → koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vonkajšieho cyklu
						Návrat z funkcie, uvoľnenie priestoru pre lokálne premenné funkcie.

Návrat do *mainu*						
Volanie funkcie *nacitanie_matice(B, r2, s2)*						
Odovzdanie adresy pamäťového priestoru pre maticu *B* referenčnej premennej funkcie a vytvorenie lokálnych kópií vo funkcii *r = r2* = 3, *s = s2* = 2						
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	**Pozn.**
0	0			3		Inicializácia riadiacej premennej vonkajšieho cyklu *i* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 2, inicializáica riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 → načítanie prvku matice *B[0][0]*
	1			1		Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1 → načítanie prvku matice *B[0][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
1	0			2		Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 2, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 → načítanie prvku matice *B[1][0]*
	1			1		Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1 → načítanie prvku matice *B[1][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
2	0			1		Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 2, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podmienky cyklu → 0 ≤ 1 → načítanie prvku matice *B[2][0]*
	1			0		Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu → 1 ≤ 1 → načítanie prvku matice *B[2][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu → 2 ≤ 1 → koniec vnútorného cyklu
3						Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 1 a testovanie splnenia podmienky cyklu → 3 ≤ 2→ koniec vonkajšieho cyklu

						Návrat z funkcie, uvoľnenie priestoru pre lokálne premenné funkcie.
Návrat do *mainu*						
Volanie funkcie *vycistenie_matice(C, r1, s2)*						
Odovzdanie adresy pamäťového priestoru pre maticu *C* referenčnej premennej funkcie a vytvorenie lokálnych kópií vo funkcii $r = r1 = 2$, $s = s2 = 2$						
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	**Pozn.**
0	0				0	Inicializácia riadiacej premennej vonkajšieho cyklu $i = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1$, inicializácia riadiacej premennej vnútorného cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1 \rightarrow$ vynulovanie prvku matice *C[0][0]*
	1				0	Inkrementácia riadiacej premennej vnútorného cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1 \rightarrow$ vynulovanie prvku matice *C[0][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu $j = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 1 \rightarrow$ koniec vnútorného cyklu
1	0				0	Inkrementácia riadiacej premennej vonkajšieho cyklu $i = 1$ a testovanie splnenia podmienky cyklu $\rightarrow\ 1 \leq 1$, inicializácia riadiacej premennej vnútorného cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1 \rightarrow$ vynulovanie prvku matice *C[1][0]*
	1				0	Inkrementácia riadiacej premennej vnútorného cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1 \rightarrow$ vynulovanie prvku matice *C[1][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu $j = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 1 \rightarrow$ koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej vonkajšieho cyklu $i = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 1 \rightarrow$ koniec vonkajšieho cyklu
						Návrat z funkcie, uvoľnenie priestoru pre lokálne premenné funkcie.
Návrat do *mainu*						
Volanie funkcie *sucin_matic(A, B, C, r1, s1, s2)*						
Odovzdanie adries pamäťového priestoru pre maticu *A, B* a *C* referenčným premenným funkcie a vytvorenie lokálnych kópií vo funkcii $r1 = r1 = 2$, $s1 = s1 = 3$, $s2 = s2 = 2$						
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	**Pozn.**
0	0	0				Inicializácia riadiacej premennej prvého cyklu $i = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1$,

						inicializácia riadiacej premennej druhého cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1$a inicializácia riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 2 \rightarrow$ realizácia výpočtu $C[0][0] = C[0][0] + A[0][0] \,.\, B[0][0] = 0 + 1 \,.\, 3 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[0][1] \,.\, B[1][0] = + 0 \,.\, 2 +$
		2			5	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[0][2] \,.\, B[2][0] = + 2 \,.\, 1 = 5$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
0	1	0				Inkrementácia riadiacej premennej druhého cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow\ 1 \leq 1$ a inicializáca riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 2 \rightarrow$ realizácia výpočtu $C[0][1] = C[0][1] + A[0][0] \,.\, B[0][1] = 0 + 1 \,.\, 1 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[0][1] \,.\, B[1][1] = + 0 \,.\, 1 +$
		2			1	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[0][2] \,.\, B[2][1] = + 2 \,.\, 0 = 1$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	2					Inkrementácia riadiacej premennej druhého cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec druhého cyklu
1	0	0				Inkrementácia riadiacej premennej prvého cyklu $i = 1$ a testovanie splnenia podmienky cyklu $\rightarrow\ 1 \leq 1$, inicializácia riadiacej premennej druhého cyklu $j = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 1$ a inicializácia riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow\ 0 \leq 2 \rightarrow$ realizácia výpočtu $C[1][0] = C[1][1] + A[1][0] \,.\, B[0][0] = 0 + (-1) \,.\, 3 +$

		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[1][1] \,.\, B[1][0] = + 3 \,.\, 2 +$
		2			4	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[1][2] \,.\, B[2][0] = + 1 \,.\, 1 = 4$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	1	0				Inkrementácia riadiacej premennej druhého cyklu $j = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 1$ a inicializáca riadiacej premennej tretieho cyklu $k = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 2 \rightarrow$ realizácia výpočtu $C[1][1] = C[1][1] + A[1][0] \,.\, B[0][1] = 0 + (-1) \,.\, 1 +$
		1				Inkrementácia riadiacej premennej tretieho cyklu $k = 1$ a testovanie splnenia podmienky cyklu $\rightarrow 1 \leq 2 \rightarrow$ realizácia výpočtu $+ A[1][1] \,.\, B[1][1] = + 3 \,.\, 1 +$
		2			2	Inkrementácia riadiacej premennej tretieho cyklu $k = 2$ a testovanie splnenia podmienky cyklu $\rightarrow 2 \leq 2 \rightarrow$ realizácia výpočtu a zápis hodnoty $+ A[1][2] \,.\, B[2][1] = + 1 \,.\, 0 = 2$
		3				Inkrementácia riadiacej premennej tretieho cyklu $k = 3$ a testovanie splnenia podmienky cyklu $3 \leq 2 \rightarrow$ koniec tretieho cyklu
	2					Inkrementácia riadiacej premennej druhého cyklu $j = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec druhého cyklu
2						Inkrementácia riadiacej premennej prvého cyklu $i = 2$ a testovanie splnenia podmienky cyklu $2 \leq 1 \rightarrow$ koniec prvého cyklu
						Návrat z funkcie, uvoľnenie priestoru pre lokálne premenné funkcie.
Návrat do *mainu*						
Volanie funkcie *vypis_matice(C, r1, s2)*						
Odovzdanie adresy pamäťového priestoru pre maticu *C* referenčnej premennej funkcie a vytvorenie lokálnych kópií vo funkcii $r = r1 = 2$, $s = s2 = 2$						
i	**j**	**k**	**A[r1][s1]**	**B[r2][s2]**	**C[r1][s2]**	**Pozn.**
0	0					Inicializácia riadiacej premennej prvého cyklu $i = 0$ a testovanie splnenia podmienky cyklu $\rightarrow 0 \leq 1$ a inicializácia riadiacej premennej druhého cyklu $j = 0$

						a testovanie splnenia podmienky cyklu → 0 ≤ 1 → Výstup na obrazovku prvku *C[0][0]*
	1					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu 1 ≤ 1 → Výstup na obrazovku prvku *C[0][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu 2 ≤ 1→ koniec vnútorného cyklu
1	0					Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 1 a testovanie splnenia podmienky cyklu 1 ≤ 1, inicializácia riadiacej premennej vnútorného cyklu *j* = 0 a testovanie splnenia podminky cyklu 0 ≤ 1→ Výstup na obrazovku prvku *C[1][0]*
	1					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 1 a testovanie splnenia podmienky cyklu 1 ≤ 1→ Výstup na obrazovku prvku *C[1][1]*
	2					Inkrementácia riadiacej premennej vnútorného cyklu *j* = 2 a testovanie splnenia podmienky cyklu 2 ≤ 1→ koniec vnútorného cyklu
2						Inkrementácia riadiacej premennej vonkajšieho cyklu *i* = 2 a testovanie splnenia podmienky cyklu 2 ≤ 1→ koniec vonkajšieho cyklu
						Návrat z funkcie, uvoľnenie priestoru pre lokálne premenné funkcie.
Návrat do *mainu* → koniec programu						

Nutné vlastnosti algoritmu:

- **Determinovanosť –** táto vlastnosť algoritmu je splnená. Po načítaní prvkov vstupných matíc sa algoritmus realizuje predpísaným spôsobom, pričom v každom kroku je zrejmé, čo sa má realizovať a kade pokračuje tok riadenia.
- **Konečnosť –** každý algoritmus musí skončiť po vykonaní konečného počtu krokov. V tomto prípade je konečnosť zabezpečená s pomocou cyklov s pevným počtom opakovaní tak pri načítaní, spracovaní aj výpise prvkov matíc. Pri načítavaní správnych rozmerov matíc je konečnosť zabezepečená s pomocou cyklov s podmienkou na konci a správne navrhnutými podmienkami cyklu.

Rezultatívnosť – aj táto vlastnosť je splnená. Vstupná podmienka *r1*, *s1*, *r2*, *s2* $\in Z^+$ v prípade tvorby programu bude zabezpečená deklaráciou premenných požadovaného dátového typu: *int* a ošetrením pri načítavaní. Vstupné podmienky *A[i][j]* $\in R \wedge$ *B[i][j]* $\in R$, ktoré hovoria o tom,

že prvky matice sú definované na množine *R*, budú zabezpečené deklarovaním polí požadovaného dátového typu (*float*/*double*). Vstupná podmienka *r1 = s2* bude zabezpečená ošetrením hneď pri načítavaní rozmerov druhej matice. Výstupná podmienka *C[i][j]* ∈ R bude zabezpečená deklarovaním premennej typu pole požadovaného dátového typu (*float*/*double*) a druhá časť podmienky *C[i][j] = A[i, 1]B[1, j] + A[i, 2]B[2, j] + … + A[i, n]B[m, j]* bude zabezpečená správne navrhnutým a zrealizovaným algoritmom.

Odporúčané/očakávané vlastnosti algoritmu:

- **Hromadnosť** – môžeme konštatovať, že dané riešenie spĺňa túto vlastnosť vzhľadom na zadanie problému. To znamená, že algoritmus je platný nielen pre konkrétne hodnoty, ale pre celú prípustnú množinu hodnôt, čo je zadefinované vstupnou podmienkou (rozmery matíc môžu byť ľubovoľné celé kladné čísla, avšak musí platiť, že rozmer riadkov ľavej matice sa musí rovnať rozmeru stĺpcov pravej matice a prvky matice sú definované na celej množine reálnych čísiel).
- **Elementárnosť** – vlastnosť je splnená, algoritmus je zložený z činností, ktoré sú pre realizátora elementárne, zrozumiteľné. Realizácia algoritmu je nezávislá od riešiteľa a od prostredia, v ktorom sa algoritmus bude realizovať (implementovať).
- **Efektívnosť** – môžeme konštatovať, že aj táto vlastnosť je splnená.

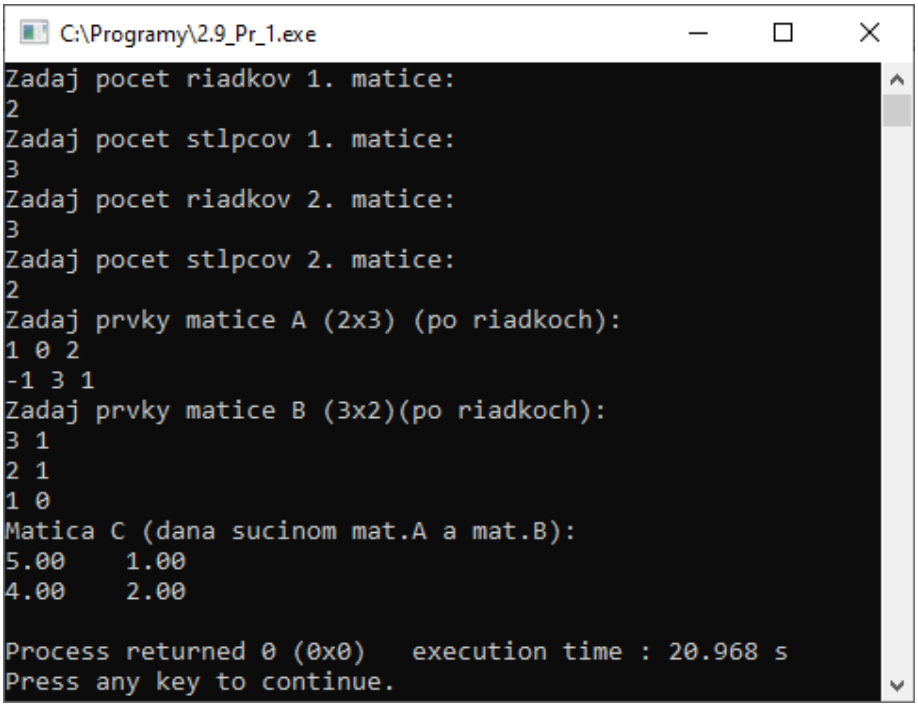
Priestorová zložitosť	main()	*S*(1) = konštantná Používa sa sedem premenných: *r1*, *s1*, *r2*, *s2*, *A[r1][s1]*, *B[r2][s2]* a *C[r1][s2]* Rozmery matíc môžu byť rôzne – sú závislé od používateľa, avšak počas behu algoritmu sa prietorové nároky nemenia, preto stále hovorím o konštatnej priesto-rovej zložitosti.
	nacitanie_matice()	*S*(1) = konštantná Funkcia má tri parametre: premennú, ktorá reprezentuje maticu *A* a jej rozmery *r* a *s*. Disponuje dvomi lokálnymi premennými *i* a *j*.
	vypis_matice()	*S*(1) = konštantná

		Funkcia má tri parametre: premennú, ktorá reprezentuje maticu *A* a jej rozmery *r* a *s*. Disponuje dvomi lokálnymi premennými *i* a *j*.
	vycistenie_matice()	*S*(1) = konštantná Funkcia má tri parametre: premennú, ktorá reprezentuje maticu *A* a jej rozmery *r* a *s*. Disponuje dvomi lokálnymi premennými *i* a *j*.
	sucin_matic()	*S*(1) = konštantná Funkcia má šesť parametrov: premennú, ktorá reprezentuje vstupnú maticu *A*, vstupnú maticu *B* a výstupnú maticu C, ako aj prislúchajúce rozmery matíc potrebné k výpočtu *r1, s1* a *s2*. Disponuje dvomi lokálnymi premennými *i* a *j*.
Časová zložitosť	main()	$O(1)$ = konštantná
	nacitanie_matice()	$O(n^2)$ = kvadratická
	vypis_matice()	$O(n^2)$ = kvadratická
	vycistenie_matice()	$O(n^2)$ = kvadratická
	sucin_matic()	$O(n^3)$ = kubická

Jedna z možných implementácií bez funkcií 2.9_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    const int MAX = 10;
    float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    int r1, s1, r2, s2, i, j, k;
    // nacitanie rozmerov matic
    do
    {
        printf("Zadaj pocet riadkov 1. matice: \n");
        scanf("%d", &r1);
    }
    while (r1 <= 0 || r1 > MAX);

    do
    {
        printf("Zadaj pocet stlpcov 1. matice: \n");
        scanf("%d", &s1);
    }
    while (s1 <= 0 || s1 > MAX);

    do
    {
        printf("Zadaj pocet riadkov 2. matice: \n");
        scanf("%d", &r2);
    }
    while (r2 <= 0 || r2 > MAX);

    do
    {
        printf("Zadaj pocet stlpcov 2. matice: \n");
        scanf("%d", &s2);
    }
    while (s2 <= 0 || s2 > MAX || r1 != s2);

    // nacitanie prvkov matic
    printf("Zadaj prvky matice A (%d x %d) po riadkoch: \n", r1, s1);
    for (i = 0; i < r1; i++)
    {
        for (j = 0; j < s1; j++)
        {
            scanf("%f", &A[i][j]);
        }
    }
    printf("Zadaj prvky matice B (%d x %d) po riadkoch: \n", r2, s2);
    for (i = 0; i < r2; i++)
    {
        for (j = 0; j < s2; j++)
        {
            scanf("%f", &B[i][j]);
        }
    }

    // sucin matic
    printf("Matica C (dana sucinom mat. A a mat. B):\n");
```

```
57      for (i = 0; i < r1; i++)
58      {
59          for (j = 0; j < s2; j++)
60          {
61              int c = 0;
62              for (k = 0; k < s1; k++)
63              {
64                  c += A[i][k] * B[k][j];
65              }
66              C[i][j] = c;
67          }
68      }
69
70      // vypis vystupnej matice
71      for (i = 0; i < r1; i++)
72      {
73          for (j = 0; j < s2; j++)
74          {
75              printf("%.2f\t", C[i][j]);
76          }
77          printf("\n");
78      }
79
80      return 0;
81  }
```



Obr. 37 Konzolový výstup programu 2.9_Pr_1.c.

Pozn. autora: Môžete si všimnúť, že pri realizácii súčinu matíc (riadky 55 – 68) som použila pomocnú premennú *c*, do ktorej sa ukladajú medzivýsledky a následne až potom sa konečná hodnota zapíše na správne miesto v matici *C*. Je to z dôvodu, že takýto prístup by mal minimalizovať počet operácií na pozadí, ktoré sú spojené s prístupom k jednotlivým adresám

prvkov výslednej matice *C*. Vďaka tejto modifikácii, nie je ani nevyhnutné čistiť výstupnú maticu *C*, pretože korektnosť výpočtov je zabezpečená nulovaním pomocnej premennej *c* – riadok 61.

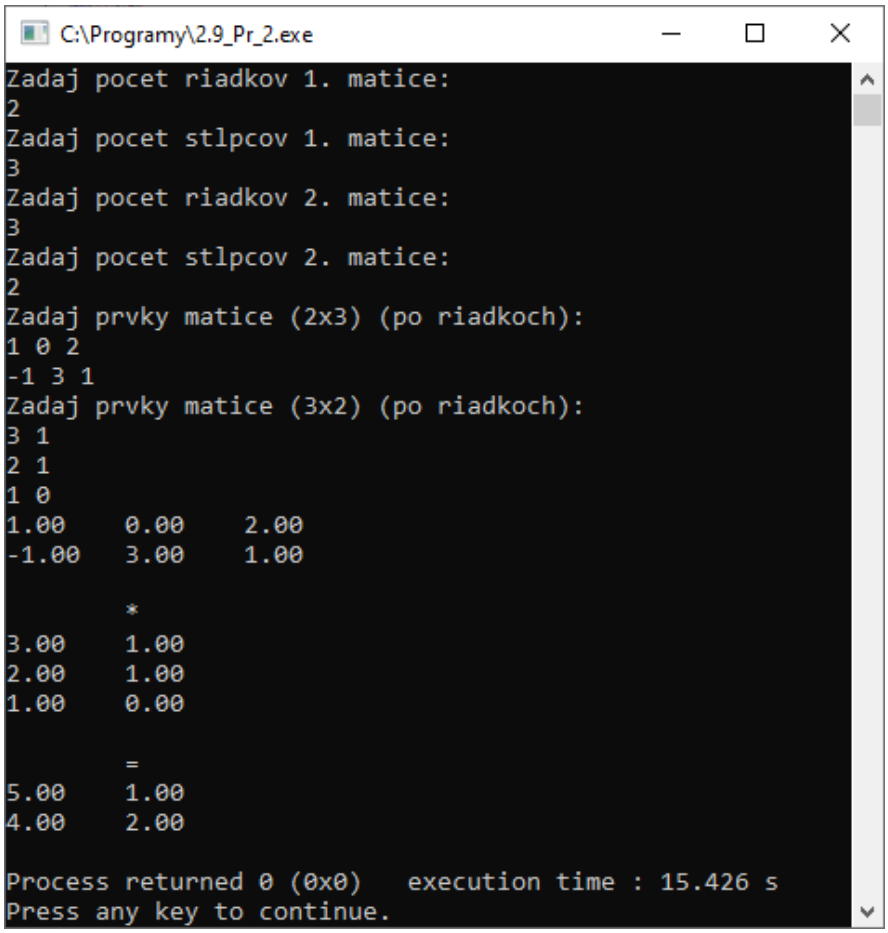
Ak by sme chceli vyčísliť priestorové nároky riešenia, na základe priradených dátových typov premenných, tie predstavujú momentálne 1 236 B = *MAX* (*int* = 4 B) + *A[10][10]* (*float* = 4 B . 100) + *B[10][10]* (*float* = 4 B . 100) + *C[10][10]* (*float* = 4 B . 100) + *r1* (*int* = 4 B) + *s1* (*int* = 4 B) + *r2* (*int* = 4 B) + *s2* (*int* = 4 B) +*i* (*int* = 4 B) + *j* (*int* = 4 B) + *k* (*int* = 4 B) + c (*int* = 4 B).

Jedna z možných implementácií s funkciami 2.9_Pr_2.c:

```
1  #include <stdio.h>
2
3  const int MAX = 10;
4
5  // deklaracie funkcii
6  void nacitanie_matice(float A[][MAX], int r, int s);
7  void sucin_matic(float A[][MAX], float B[][MAX], float C[][MAX], int r1, int s1, int s2);
8  void vypis_matice(float A[][MAX], int r, int s);
9
10 int main(void)
11 {
12     float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
13     int r1, s1, r2, s2;
14
15     // nacitanie rozmerov matic
16     do
17     {
18         printf("Zadaj pocet riadkov 1. matice: \n");
19         scanf("%d", &r1);
20     }
21     while (r1 <= 0 || r1 > MAX);
22
23     do
24     {
25         printf("Zadaj pocet stlpcov 1. matice: \n");
26         scanf("%d", &s1);
27     }
28     while (s1 <= 0 || s1 > MAX);
29
30     do
31     {
32         printf("Zadaj pocet riadkov 2. matice: \n");
33         scanf("%d", &r2);
34     }
35     while (r2 <= 0 || r2 > MAX);
36
37     do
38     {
39         printf("Zadaj pocet stlpcov 2. matice: \n");
40         scanf("%d", &s2);
41     }
42     while (s2 <= 0 || s2 > MAX || r1 != s2);
```

```
43
44      // nacitanie prvkov matic
45      nacitanie_matice(A, r1, s1);
46      nacitanie_matice(B, r2, s2);
47
48      // sucin matic
49      sucin_matic(A, B, C, r1, s1, s2);
50
51      // vypis matic
52      vypis_matice(A, r1, s1);
53      printf("\n\t*\t\n");
54      vypis_matice(B, r2, s2);
55      printf("\n\t=\t\n");
56      vypis_matice(C, r1, s2);
57
58      return 0;
59  }
60
61  // definicie funkcii
62  void nacitanie_matice(float A[][MAX], int r, int s)
63  {
64      int i, j;
65      printf("Zadaj prvky matice (%d x %d) po riadkoch: \n", r, s);
66      for (i = 0; i < r; i++)
67      {
68          for (j = 0; j < s; j++)
69          {
70              scanf("%f", &A[i][j]);
71          }
72      }
73  }
74
75  void sucin_matic(float A[][MAX], float B[][MAX], float C[][MAX], int r1, int s1, int s2)
76  {
77      int i, j, k, c;
78      for (i = 0; i < r1; i++)
79      {
80          for (j = 0; j < s2; j++)
81          {
82              c = 0;
83              for (k = 0; k < s1; k++)
84              {
85                  c += A[i][k] * B[k][j];
86              }
87              C[i][j] = c;
88          }
89      }
90  }
91
92  void vypis_matice(float A[][MAX], int r, int s)
93  {
94      int i, j;
95      for (i = 0; i < r; i++)
96      {
97          for (j = 0; j < s; j++)
98          {
99              printf("%.2f\t", A[i][j]);
100         }
```

```
101             printf("\n");
102         }
103     }
```



Obr. 38 Konzolový výstup programu 2.9_Pr_2.c.

Vyjadrenie priestorových nárokov riešenia s funkciami:

Priestorové nároky	main()	*MAX* (*int* = 4 B) + *r1* (*int* = 4 B) + *s1* (*int* = 4 B) + *r2* (*int* = 4 B) + *s2* (*int* = 4 B) + *A[10][10]* (*float/double* = 4/8 B . 10 . 10) + *B[10][10]* (*float/double* = 4/8 B . 10 . 10) + *C[10][10]* (*float/double* = 4/8 B . 10 . 10) = 20 B + 3 . 400/800 B **= 1 220 B/2 420 B**
	nacitanie_matice()	*A* (*float* ** = 4/8 B) + *r* (*int* = 4 B) + *s* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B) = **20/24 B**
	vypis_matice()	*A* (*float* ** = 4/8 B) + *r* (*int* = 4 B) + *s* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B) = **20/24 B**
	sucin_matic()	*A* (*float* ** = 4/8 B) + *B* (*float* ** = 4/8 B) + *C* (*float* ** = 4/8 B) + *r1* (*int* = 4 B) + *s1* (*int* = 4 B) + *s2* (*int* = 4 B) + *i* (*int* = 4 B) + *j* (*int* = 4 B) + *k* (*int* = 4 B) + *c* (*int* = 4 B) = **40/52 B**

Netreba zabudnúť, že priestorové nároky funkcií vznikajú, keď dôjde k ich vykonávaniu a zanikajú v momente, keď dôjde k ich ukončeniu. Priestorové nároky funkcie *main()* sú trvalé od začiatku až po koniec programu.

2.10 Úlohy na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problémy z kapitoly 1.5 úlohy 1 – 5, ale pri riešení použite funkcie.
2. Algoritmicky riešte problém, ktorý umožní používateľovi zadať tri čísla (*a, b, c*) a následne ich navzájom po dvojiciach porovná (*a* z *b*, potom *b* z *c* a nakoniec *a* z *c*) a označí vzťah medzi nimi (<, >, =). Na porovnanie dvojice čísiel a vypísanie vzťahu medzi nimi použite funkciu.
3. Algoritmicky riešte problém, ktorý pre čísla od 1 – 20 vypíše ich *n*-té mocniny. Umožnite *n* zadať používateľovi a na výpočet mocniny použite vlastnú funkciu.
4. Algoritmicky riešte problém, ktorý pre čísla od 1 – 20 vypíše ich 0 – 5 mocninu. Znovu použite funkciu z úlohy 3.
5. Algoritmicky riešte problém, ktorý umožní číslo zadané v desiatkovej sústave previesť do sústavy dvojkovej, osmičkovej a šestnástkovej, s pomocou vlastných funkcií.

3 Rekurzia

Podľa Wróblewkeho (2004) pod rekurziou rozumieme techniku, ktorá zahŕňa opakované použitie programovej konštrukcie pri riešení tej istej úlohy, pričom použitie tej istej konštrukcie je zahrnuté vo vnútri samotnej konštrukcie. Používa sa všade tam, kde je efektívne pôvodnú úlohu rozdeliť na menšie podúlohy a potom použiť ten istý postup riešenia pre každú podúlohu, ktorá je už ale v určitom zmysle jednoduchšia. Sila rekurzie spočíva aj v možnosti definovať nekonečnú množinu objektov konečným príkazom (Gunišová a Guniš, 2019). Neodporúča sa používať rekurziu, ak sa môže vykonať príliš veľa iterácií (opakovaní volania funkcie), pretože rekurzia využíva veľa pamäte (každé volanie funkcie obsadzuje miesto v zásobníku – na odovzdanie parametrov funkcií a uchovanie návratovej adresy + miesto v pamäti pre lokálne premenné definované vo vnútri funkcie). Preto rekurziu odporúčame používať iba pri funkciách s plytkou rekurziou, t. j. s malým počtom vnáraní a s malým počtom lokálnych premenných.

3.1 Rekurzívna funkcia

V programe sa rekurzia reprezentuje funkciou, ktorá vo vnútri tela obsahuje volanie tej istej funkcie. Hovoríme, že funkcia „volá samu seba“. Je veľmi dôležité, aby sme pri každom rekurzívnom volaní funkcie správne ošetrili ukončovaciu podmienku, inak sa môže stať rekurzia nekonečnou, čo väčšinou vedie k havarovaniu programu kvôli pretečeniu zásobníka (*stack overflow*).

Rekurzívny algoritmus *A* možno vyjadriť ako kompozíciu *K*, ktorá sa skladá zo základných príkazov *Pi* a samotného *A*.

$$A = K\,[Pi, A]$$

Rozlišujeme dva typy rekurzívnych konštrukcií:

- **Nepriama rekurzia** nastáva vtedy, keď jedna funkcia volá inú funkciu, ktorá potom opäť volá prvú funkciu, t. j. ak funkcia *A* obsahuje odkaz na inú funkciu *B*, ktorá obsahuje odkaz na funkciu *A*.
- **Priama rekurzia** znamená, že v tele funkcie vykonávame volanie tej istej funkcie, t. j. ak funkcia *A* obsahuje priamy odkaz na seba.

Aj objekt môže byť rekurzívny, ak sa čiastočne skladá, alebo je definovaný s pomocou seba samého.

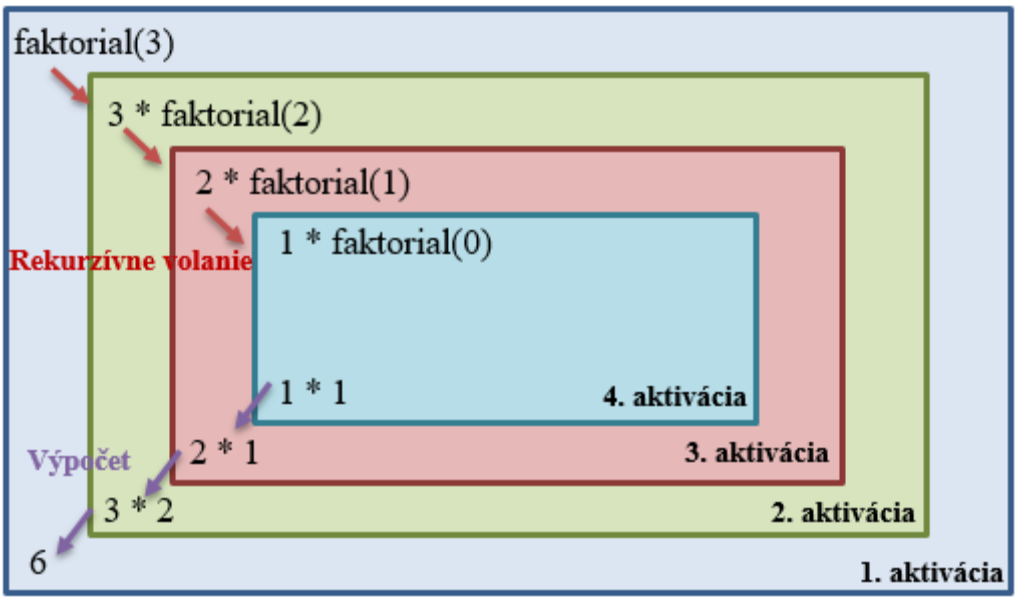
Napríklad na základe definície faktoriálu: „Faktoriálom kladného celého čísla *n* označujeme súčin všetkých kladných celých čísel menších alebo rovných *n*.“, čo sa zapisuje $n \rightarrow n!$. Pritom platí:

- 0! = 1
- ak $n > 0$, tak $n! = n \,.\, (n-1)!$

Je zrejmé, že faktoriál je sám o sebe definovaný rekurzívne a preto sa na prvý pohľad zdá byť vhodné definovať na jeho výpočet rekurzívnu funkciu:

```
int faktorial(int n)
{
    if (n < 1)
        return 1;
    else
        return n * faktorial(n - 1);
}
```

Treba si však uvedomiť, že vždy keď je rekurzívne volaná funkcia, tak sa alokuje nová množina lokálnych premenných a parametrov volaných hodnotou a iba táto nová množina môže byť odkazovaná vo vnútri volania. Ako teda funguje rekurzia z uvedeného príkladu?



Obr. 39 Ilustrácia vykonávania funkcie *faktorial(3)*.

- Predpokladajme, že funkcia *faktorial()* je volaná v programe napr. s parametrom 3, t. j. *faktorial(3);*
- Po zavolaní *faktorial(3)* je hodnota lokálnej premennej *n* = 3.
- *n* nie je menšie ako 1, tak sa znova volá funkcia, ale s argumentom 2, t. j. *faktorial(2);*

- Vstup do *faktorial()* spôsobí novú alokáciu lokálnej premennej *n*, pričom jej predchádzajúca alokácia sa zachová, ale nie je prístupná.
- … proces pokračuje obdobne ďalej, tak ako ilustruje obrázok 39.
- Návrat z *faktorial()* pri splnení podmienky $n < 1$ do bodu predchádzajúceho volania uvoľňuje najnovšiu alokáciu premenných a aktivuje predchádzajúcu množinu týchto premenných. Je to realizované s pomocou zásobníka.
- Po každom vstupe do rekurzívnej funkcie sa nová alokácia jej premenných umiestni do zásobníka a každý odkaz na lokálne premenné je cez aktuálny vrchol zásobníka.
- Každý návrat z rekurzívnej funkcie znamená odber zo zásobníka a na jeho vrchol sa umiestni predchádzajúca alokácia.
- Ak uvažujeme, že premenná dátového typu *int* zaberá v pamäti 4 B, tak priestorové nároky pre funkciu *faktorial(3)* by sme vyčíslili na hodnotu 16 B.

Pozn. autora: Ak by sme upravili podmienku ukončenia na $n \leq 1$, ušetrili by sme jedno rekurzívne vnorenie a priestorové nároky toho istého volania by boli 12 B.

```
int faktorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * faktorial(n - 1);
}
```

- V prípade volania *faktorial(3)*:
- *n* = 3 a teda podmienka ($n \leq 1$) neplatí a vyvolá sa *return* 3 . *faktorial(2)*
    - *faktorial(2)*
    - *n* = 2 a teda ($n \leq 1$) neplatí a vyvolá sa *return* 2 . *faktorial(1)*
        - *faktorial(1)*
        - *n* = 1 a teda podmienka ($n \leq 1$) platí a vyvolá sa *return 1*
        - funkcia vráti 1
    - funkcia vráti 2 . 1 = 2
- funkcia vráti 3 . 2 = 6

Jednoduchá rekurzívna funkcia teda môže byť z hľadiska priestorovej náročnosti nevýhodná. Rekurzia sa preto často nahrádza iteráciou. Dôvodom je často prehľadnejší aj rýchlejší a úspornejší kód. Príklad nerekurzívneho variantu výpočtu faktoriálu:

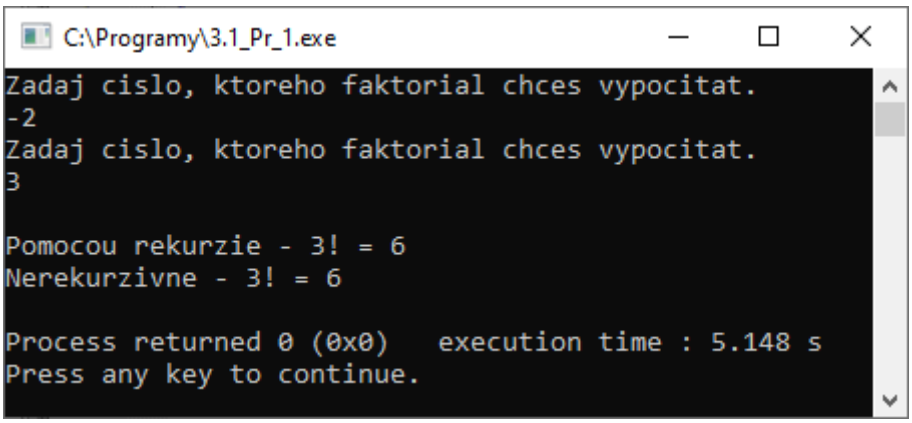
```c
int faktorial_nerekurzivne(int n)
{
    int i, fakt = 1;
    for (i = 2; i <= n; i++)
    {
        fakt = i * fakt;
    }
    return fakt;
}
```

- Pri vykonávaní funkcie *faktorial_nerekurzivne()* sa v pamäti alokuje pamäťať na tri premenné (parameter funkcie *n* a lokálne premenné *i* a *fakt*). Keďže to nie je rekurzívna funkcia, tak počas celého vykonávania funkcie máme len tieto tri premenné, bez ohľadu na hodnotu vstupného parametra *n*. To znamená, že pamäťové nároky by sme mohli vyčísliť na 12 B. Na prvý pohľad sa môže zdať, že nevzniká žiadna úspora, ale ako sme uviedli, tieto pamäťové nároky sú nemenné v závislosti od hodnoty skutočného parametra funkcie.
- Vo funkcii *faktorial()* máme síce len jednu premennú, a to parameter funkcie *n*, ale pri niekoľkonásobnom rekurzívnom zavolaní si musíme pamätať hodnotu premennej *n* pri každom volaní, čo znamená, že táto je závislá od hodnoty vstupného parametra funkcie.

Jedna z možných implementácií 3.1_Pr_1.c:

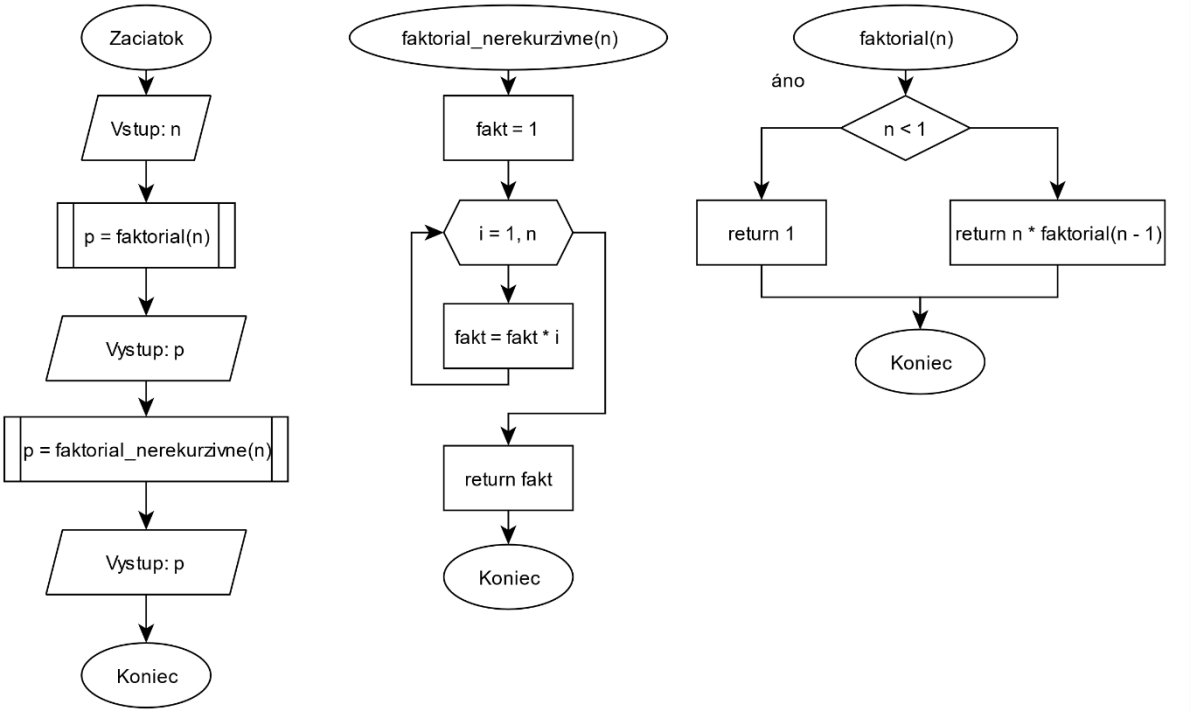
```c
#include <stdio.h>

int faktorial(int n);  // deklaracia rekurzivnej funkcie
int faktorial_nerekurzivne(int n);  // deklaracia nerekurzivnej funkcie

int main(void)
{
    int n;
    do
    {
        printf("Zadaj cislo, ktoreho faktorial chces vypocitat.\n");
        scanf("%d", &n);
    }
    while (n < 0);

    printf("\nS pomocou rekurzie - %d! = %d \n", n, faktorial(n));
    printf("Nerekurzivne - %d! = %d \n", n, faktorial_nerekurzivne(n));

    return 0;
}

// definicia rekurzivnej funkcie
int faktorial(int n)
{
    if (n < 1)
        return 1;
    else
```

```
28          return n * faktorial(n - 1);
29  }
30
31  // definicia nerekurzivnej funkcie
32  int faktorial_nerekurzivne(int n)
33  {
34      int i, fakt = 1;
35      for (i = 1; i <= n; i++)
36      {
37          fakt = i * fakt;
38      }
39      return fakt;
40  }
```



Obr. 40 Konzolový výstup programu 3.1_Pr_1.c.

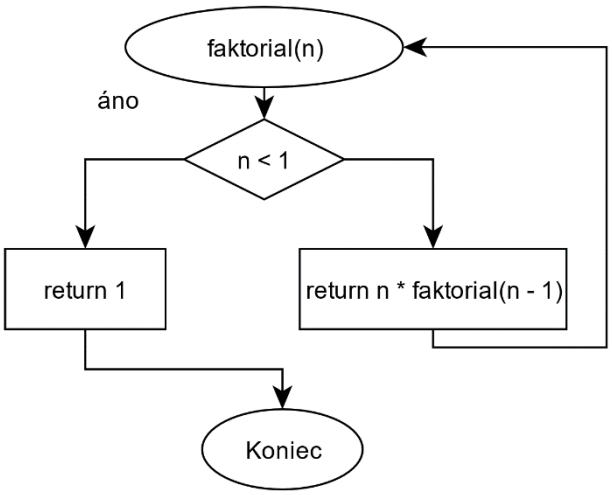
Reprezentácia algoritmu s pomocou vývojového diagramu:



Obr. 41 Vývojový diagram na riešenie výpočtu faktoriálu.

Pozn. autora: Reprezentácia rekurzie v grafickej podobe je značne problematická. Z vývojového diagramu nie je na prvý pohľad zrejmé, že funkcia *faktorial()* je rekurzívna. Vyplýva to až z kontextu a vedomostí, že rekurzívna funkcia musí obsahovať volanie samej seba. Opäť teda narážame na problém, či takáto reprezentácia algoritmu spĺňa vlastnosť elementárnosti. Odpoveď je jednoduchá: možno nie pre všetkých čitateľov, avšak pre programátorov áno.

Študenti často v prípade použitia vývojového diagramu majú tendenciu zakresliť rekurzívnu funkciu takto:

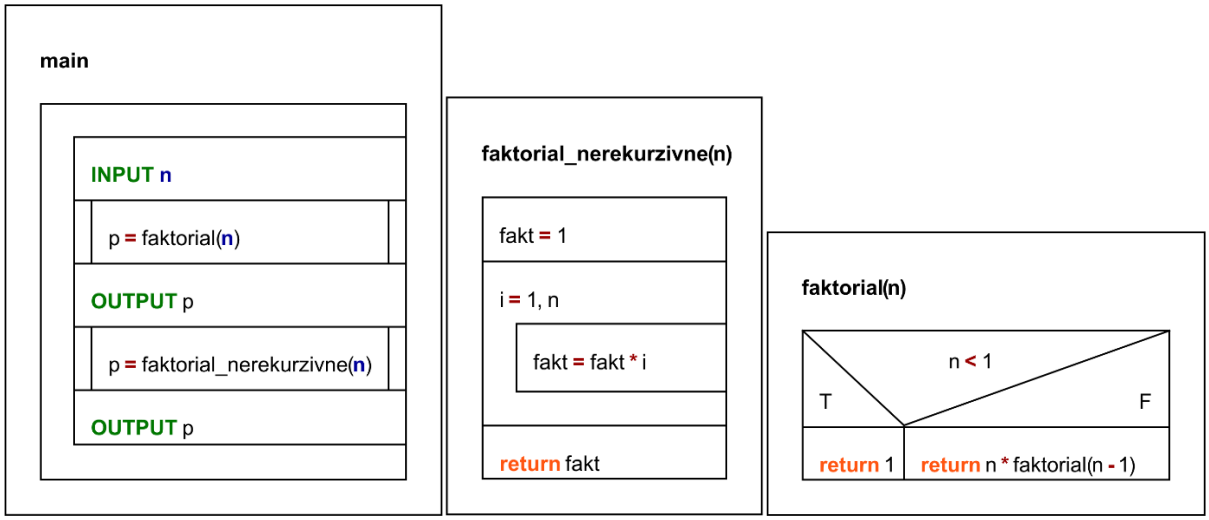


Obr. 42 Nesprávny vývojový diagram rekurzívnej funkcie na výpočet faktoriálu.

Takúto reprezentáciu nepovažujeme za vhodnú z dôvodu, že porušuje základné princípy reprezentáce algoritmu s pomocou vývojového diagramu, ale ani jednoznačne nereprezentuje opisovanú skutočnosť. Odkrokujme si ju pre volanie funkcie s parametrom 3:

- V prípade volania *faktorial(3)*:
- *n* = 3 a teda podmienka (*n* < 1) neplatí a vyvolá sa *return* 3 . *faktorial(2)*
    - *faktorial(2)*
    - *n* = 2 a teda (*n* < 1) neplatí a vyvolá sa *return* 2 . *faktorial(1)*
        - *faktorial(1)*
        - *n* = 1 a teda podmienka (*n* < 1) neplatí a vyvolá sa *return* 1 . *faktorial(0)*
            - *faktorial(0)*
            - *n* = 0 a teda podmienka (*n* < 1) platí a funkcia vráti 1
            - Podľa zakreslenia toku riadenia by malo dôjsť k ukončeniu funkcie, to sa však, ako vieme, nedeje. Toto je hlavný dôvod, prečo takéto zakreslenie neodporúčam.
        - vyprázdňuje sa zásobník, funkcia vráti 1 . 1 = 1
    - vyprázdňuje sa zásobník, funkcia vráti 2 . 1 = 2
- vyprázdňuje sa zásobník, funkcia vráti 3 . 2 = 6

Na druhej strane, pri grafickej reprezentácii s pomocou NS diagramu nemáme možnosť meniť tok riadenia ako v prípade vývojových diagramov, takže takýto variant by ani neprichádzal do úvahy.



Obr. 43 NS diagram na riešenie výpočtu faktoriálu.

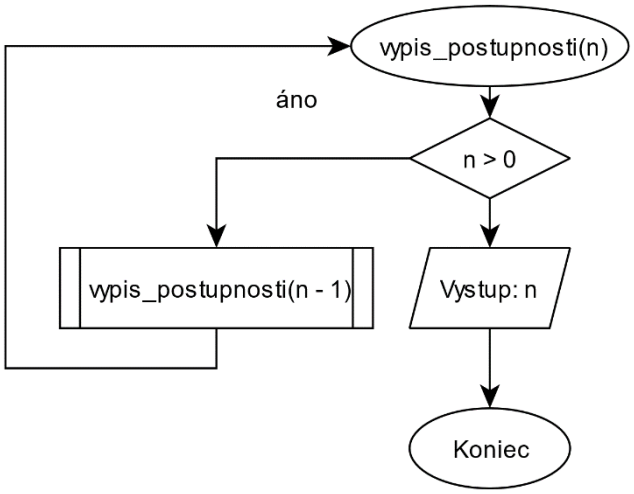
3.1.1 Príklad – výpis postupnosti čísiel

Uveďme si ešte jeden príklad. Predstavte si, že potrebujete navrhnúť rekurzívnu funkciu, ktorá vypíše postupnosť celých čísiel od zadaného čísla *n* až po 0 (vrátane), teda že pri zadaní čísla 5 sa vypíše: 5 4 3 2 1 0

Definícia takejto funkcie by mohla vyzerať:

```
void vypis_postupnosti(int n)
{
    if (n > 0)
        vypis_postupnosti(n - 1);
    printf("%d ", n);
}
```

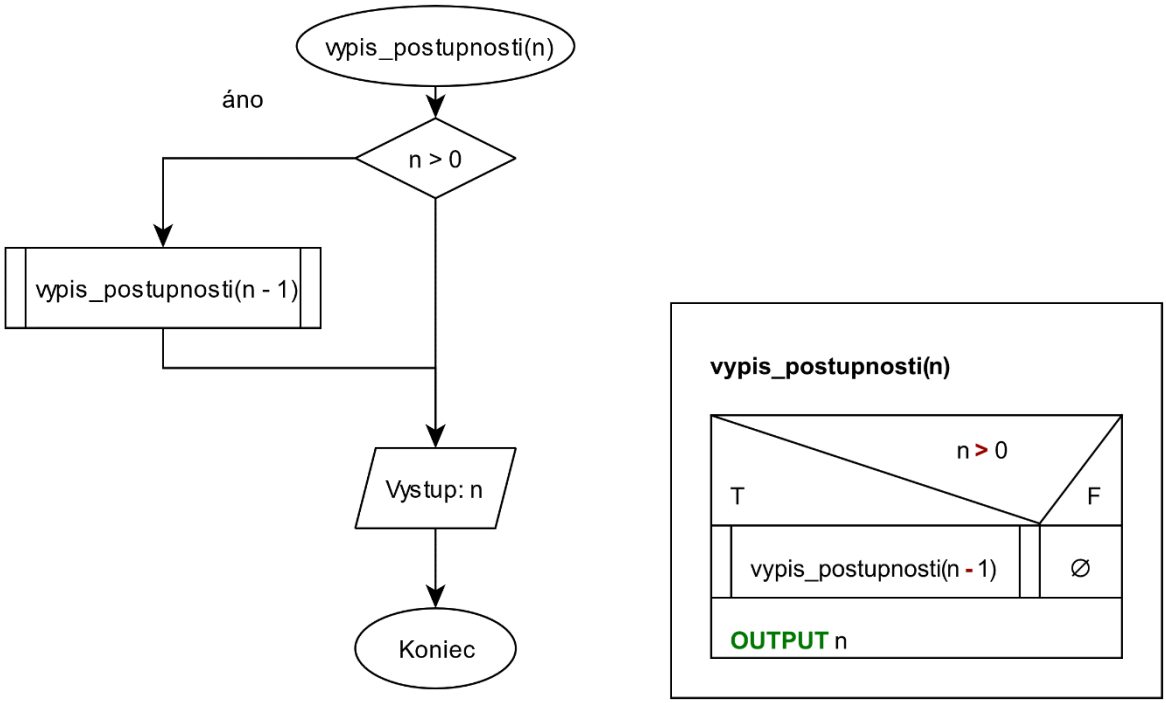
Pri jej grafickej reprezentácii sa dostaneme k obdobným problémom ako sme uviedli vyššie.



Obr. 44 Vývojový diagram rekurzívnej funkcie na výpis postupnosti – nesprávny.

Odkrokujme si ju pre volanie funkcie s parametrom 3:

- V prípade volania *vypis_postupnosti(3)*:
- $n = 3$ a teda podmienka ($n > 0$) platí a nastáva opätovné volanie funkcie *vypis_postupnosti(2)*
    - *vypis_postupnosti(2)*
    - $n = 2$ a teda ($n > 0$) platí a nastáva opätovné volanie funkcie *vypis_postupnosti(1)*
        - *vypis_postupnosti(1)*
        - $n = 1$ a teda podmienka ($n > 0$) platí a vzniká opätovné volanie funkcie *vypis_postupnosti(0)*
            - *vypis_postupnosti(0)*
            - $n = 0$ a teda podmienka ($n > 0$) neplatí a dôjde k vypísaniu hodnoty parametra *n*
            - Podľa zakreslenia toku riadenia by malo dôjsť k ukončeniu funkcie, to sa však, ako vieme, nedeje. K ukončeniu rekurzívnej funkcie nepríde skôr, než je vyprázdnený zásobník s jednotlivými volaniami.
        - funkcia vypíše hodnotu parametra *n*, t. j. 1
    - funkcia vypíše hodnotu parametra *n*, t. j. 2
- funkcia vypíše hodnotu parametra *n*, t. j. 3
- koniec funkcie.



Obr. 45 Vývojový a NS diagram rekurzívnej funkcie na výpis postupnosti – správny.

Pozn. autora: Na základe vedomostí o rekurzii by ste mali vedieť, že na riešenie tohto problému nie je rekurzia vhodná. Príklad je len ilustratívny.

3.1.2 Príklad – výpočet členov Fibonacciho postupnosti

Odstrašujúcim príkladom použitia rekurzie je napríklad výpočet členov *Fibonacciho postupnosti*. Postupnosť sa začína číslami 0 a 1 a každý ďalší člen je daný súčtom predchádzajúcich dvoch členov. Dostaneme teda:

$F_0 = 0$, postupnosť je (0)

$F_1 = 1$, postupnosť je (0, 1)

$F_2 = F_0 + F_1 = 0 + 1 = 1$, postupnosť je (0, 1, 1)

$F_3 = F_1 + F_2 = 1 + 1 = 2$, postupnosť je (0, 1, 1, 2)

$F_4 = F_2 + F_3 = 1 + 2 = 3$, postupnosť je (0, 1, 1, 2, 3)

$F_5 = F_3 + F_4 = 2 + 3 = 5$, postupnosť je (0, 1, 1, 2, 3, 5)

…

Po zovšeobecnení, pre $n > 1$:

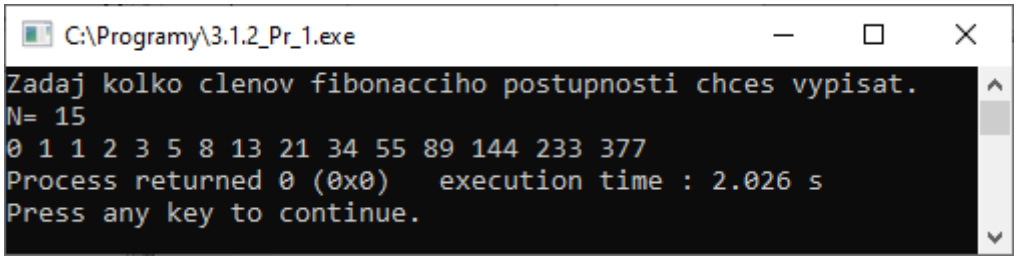
$F_n = F_{n-2} + F_{n-1}$

V prípade, ak použijeme rekurziu, dostaneme veľmi pomalý program, pretože pre každý člen postupnosti je nevyhnutné volať dvakrát celý rekurzívny výpočet.

Jednotlivé členy postupnosti sa nazývajú *Fibonacciho čísla*: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, … (Wikipédia, 2021).

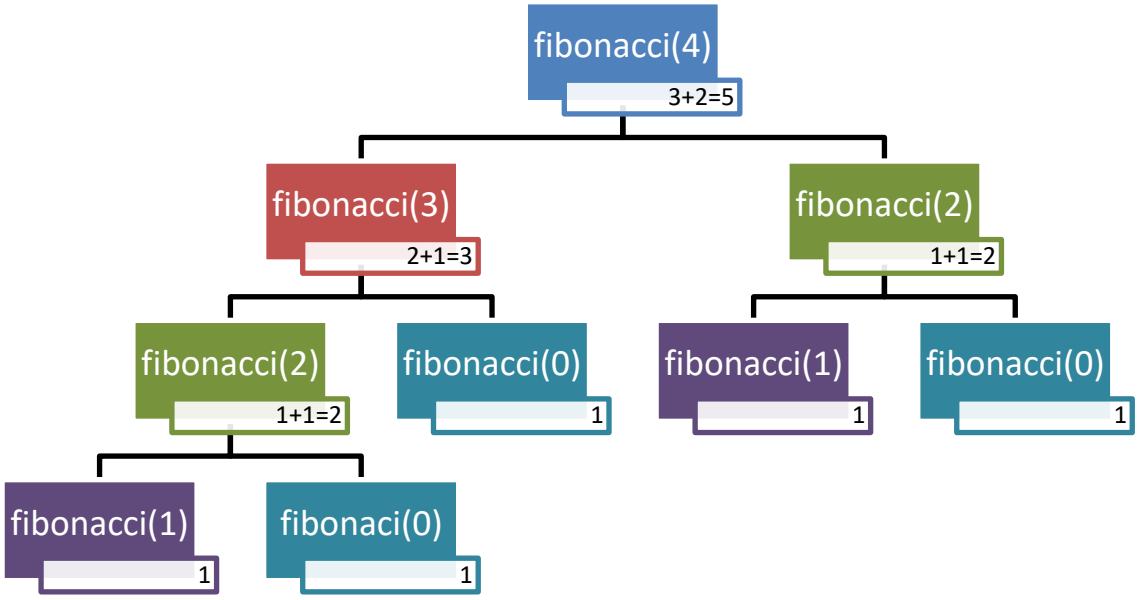
Príklad uvádza výpočet *n*-členov *Fibonacciho postupnosti* 3.1.2_Pr_1.c:

```c
#include <stdio.h>

int fibonacci(int n);   // deklaracia rekurzivnej funkcie

int main(void)
{
    int n, i;
    printf("Zadaj, kolko clenov fibonacciho postupnosti chces vypisat.\n");
    printf("N = ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        printf("%d ", fibonacci(i));
    return 0;
}

// definicia rekurzivnej funkcie
int fibonacci(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```



Obr. 46 Konzolový výstup programu 3.1.2_Pr_1.c.

Ak by sme chceli graficky reprezentovať vykonávanie výpočtu a zobraziť napríklad piateho člena *Fibonacciho postupnosti*, ktorého hodnota je 5, dopracovali by sme sa zrejme k takejto reprezentácii:



Obr. 47 Grafická reprezentácia výpočtu 5-teho člena *Fibonacciho postupnosti*.

V podstate ide o stromovú štruktúru, konkrétne binárny strom, keďže každý uzol má vždy len dvoch potomkov. Jednotlivé obdĺžniky s textom *fibonacci(X)* predstavujú samotné volania funkcie *fibonacci()*, hodnoty uvedené v spodnej časti obdĺžnikov predstavujú návratové hodnoty daných volaní. Postup volania a vyhodnocovania je smerom od koreňa stromu doľava, t. j. *fibonacci(4)* → *fibonacci(3)* → *fibonacci(2)* → *fibonacci(1)* → vyhodnotenie na 1, návrat a postup doprava *fibonacci(0)* → vyhodnotenie na 1, návrat a vyhodnotenie 1 + 1 = 2 → návrat a postup doprava *fibonacci(0)* → vyhodnotenie na 1, návrat a vyhodnotenie 2 + 1 = 3 → návrat a postup doprava *fibonacci(2)* → *fibonacci(1)* → vyhodnotenie na 1, návrat a postup doprava *fibonacci(0)* → vyhodnotenie na 1, návrat a vyhodnotenie 1 + 1 = 2 → návrat a vyhodnotenie 3 + 2 = 5. Koniec.

Časová zložitosť takejto funkcie je exponenciálna $O(2^n)$, ktorú z hľadiska programovania nepovažujeme za vhodnú.

Pozn. autora: Už pri výpočte piateho člena *Fibonacciho postupnosti* si môžeme všimnúť, že výpočet *fibonacci(0) je vykonaný trikrát*, výpočet *fibonacci(1)* dvakrát a *fibonacci(2)* tiež dvakrát. Z uvedeného je vidieť veľká nehospodárnosť pri výpočte, nehovoriac o tom, že ak chceme vypísať postupnosť *Fibonacciho čísiel*, voláme funkciu na určenie konkrétneho člena

a k čiastkovým volaniam funkcie na jeho výpočet násobne viac ráz. Riešením je použiť iteratívny výpočet členov *Fibonacciho postupnosti*, t. j. s pomocou jednoduchého dynamického programovania. Toto vychádza z myšlienky, ukladať si už vypočítané hodnoty, ktoré sú nevyhnutné na ďalšie výpočty. V tomto prípade je nevyhnutné mať dve pomocné premenné, do ktorých budeme ukladať predposledné a posledné vypočítané číslo.

Jedna z možných implementácií 3.1.2_Pr_2.c:

```c
#include <stdio.h>

int fibonacci_dp(int n);    // deklaracia funkcie

int main(void)
{
    int n, i;
    printf("Zadaj, kolko clenov fibonacciho postupnosti chces vypisat.\n");
    printf("N = ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        printf("%d ", fibonacci_dp(i));
    return 0;
}

// definicia funkcie s pomocou dynamickeho programovania
int fibonacci_dp(int n)
{
    int i;
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    int predposledne = 0, posledne = 1, vysledok = 0;
    for (i = 1; i < n; i++)
    {
        vysledok = predposledne + posledne;
        predposledne = posledne;
        posledne = vysledok;
    }
    return vysledok;
}
```

```
C:\Programy\3.1.2_Pr_2.exe
Zadaj kolko clenov fibonacciho postupnosti chces vypisat.
N= 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
Process returned 0 (0x0)   execution time : 2.382 s
Press any key to continue.
```

Obr. 48 Konzolový výstup programu 3.1.2_Pr_2.c.

3.2 Používať alebo nepoužívať rekurziu

Ako uvádzajú Guniš a Gunišová (2019) rekurzívne algoritmy sú výhodné vtedy, ak problém, ktorý sa rieši, alebo údaje, s ktorými sa má manipulovať sú definované rekurzívne. To ale nezaručuje, že použitie rekurzívneho algoritmu bude aj najlepším riešením. Výhodou rekurzie je to, že nám umožňuje rozdeliť problém na podproblémy, kde lepšie dokážeme pochopiť riešenie. Jej nevýhodou je to, že je neefektívna – pamäťovo náročná.

Výhodné je využívanie rekurzívnych funkcií na naprogramovanie správania sa niektorých matematických funkcií alebo javov v prírode, ktoré bežia „nekonečne dlho“ aj na kreslenie zaujímavých obrázkov (napr. fraktálov), ktoré by sme bez nej kreslili zbytočne dlho a zložito. Avšak, ak je možné daný problém vyriešiť s pomocou iterácie, vyhýbajme sa rekurzii. Nezabúdajte na fakt, že existuje veľa problémov, kde sa rekurzia s výhodou používa, preto ju netreba hneď zatracovať.

3.3 Úlohy na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problém, ktorý do poľa načíta postupnosť čísiel a nájde maximálny (minimálny) prvok v poli s pomocou rekurzívnej funkcie.
2. Algoritmicky riešte problém, ktorý umožní v poli vyhľadať hľadané číslo s pomocou Binárneho vyhľadávania.
3. Algoritmicky riešte problém, ktorý zrealizuje súčet čísiel od 1 po n s pomocou rekurzívnej funkcie. Používateľovi umožnite zadať n.
4. Algoritmicky riešte problém, ktorý pre čísla od 1 – 20 vypíše ich n-té mocniny. Umožnite n zadať používateľovi a na výpočet mocniny použite vlastnú rekurzívnu funkciu.
5. Algoritmicky riešte problém, ktorý vypočíta i-te *Fibonacciho číslo*. Umožnite používateľovi zadať i a problém riešte s pomocou rekurzie.
6. Algoritmicky riešte problém, ktorý nájde najväčší spoločný deliteľ dvoch zadaných čísiel *nsd(a, b)* s pomocou rekurzívnej funkcie.
7. Algoritmicky riešte problém, ktorý umožní riešiť problém Hanojských veží.

4 Dynamické údajové štruktúry

Vzhľadom na to, že ukazovatele sú nevyhnutným prostriedkom aj na implementáciu dynamických údajových štruktúr, venujeme im v tejto kapitole náležitú pozornosť. Pri spracovaní sa opieram najmä o zdroje ako Belan (2011), Herout (2010), Palmarová (2003) a Horovčák a Podlubný (1997). Správnym a vhodným používaním ukazovateľov môžeme riešenie mnohých, niekedy na prvý pohľad komplikovaných problémov zjednodušiť a zefektívniť. V niektorých situáciách sa použitie ukazovateľa dokonca vyžaduje a nemožno ho obísť. Bez toho, aby sme dokonale pochopili, čo je to ukazovateľ a vedeli ho použiť, nemôžeme povedať, že sme skúseným programátorom. Všetky používané premenné môžeme podľa doby ich trvania kategorizovať do dvoch skupín:

- **Statické premenné**

  Vznikajú vyhradením miesta kompilátorom počas prekladu a trvajú po celý čas vykonávania programu. Sú to premenné, ktoré majú presne definovanú štruktúru, ktorá je nemenná počas celého výpočtu. To znamená, že majú presne stanovený rozsah hodnôt, ktoré táto premenná môže nadobúdať a tiež majú stanovený pevný počet pamäťových miest potrebných na ich reprezentáciu.

- **Dynamické premenné**

  Vznikajú a zanikajú podľa potreby počas vykonávania programu. Sú to premenné, ktorých štruktúra sa počas výpočtu mení. Avšak, elementárne zložky týchto štruktúr sú na určitom stupni detailnosti statické. Nedefinujú sa deklaráciou, ale vzniknú aj zaniknú počas vykonávania sa programu s pomocou špeciálnych príkazov (v jazyku *C* zvyčajne s pomocou štandartných knižničných funkcií *malloc()* a *free()*). Dynamické premenné teda nemožno zaviesť v úseku deklarácií/definícií a zabezpečiť prístup k ich hodnotám s pomocou ich identifikátorov. Aby bolo možné sa k nim dostať, zavádza sa dátový typ ukazovateľ (smerník, pointer), ktorý ukazuje na premennú, ktorá je uložená v pamäti.

4.1 Smerníky alebo referencie (pointer = ukazovateľ = smerník)

Charakteristickou vlastnosťou rekurzívnych dátových štruktúr je ich premenlivá veľkosť, čím sa odlišujú od základných štruktúr. Preto nie je možné priradiť rekurzívne definovanej štruktúre pevný počet pamäťových miest, z čoho vyplýva, že kompilátor v takomto prípade nie je schopný adresovať jednotlivé zložky premenných. Hodnoty rekurzívneho typu údajov obsahujú jednu, alebo viacero zložiek toho istého typu, akého sú sami. Táto vlastnosť predstavuje analógiu s rekurzívnou funkciou, ktorá obsahuje jedno alebo viaceré volania seba samej.

Podobne ako funkcie aj definície rekurzívnych typov údajov môžu byť priamo, alebo nepriamo rekurzívne. Príkladom rekurzívnej dátovej štruktúry môže byť rodokmeň. Rodokmeň definujme ako štruktúru pozostávajúcu z mena osoby a rodokmeňa jej dvoch rodičov. Definícia vedie k nekonečnej štruktúre, avšak v skutočnosti sú rodokmene ohraničené, lebo od istej úrovne pra-pra…prarodičov sú ďalšie údaje neznáme.

Technika, ktorá rieši uvedený problém, vychádza z princípov dynamického prideľovania pamäte. Pamäť sa jednotlivým zložkám premennej prideľuje nie v čase prekladu programu, ale v tom okamihu, keď začínajú existovať – v priebehu realizácie programu. Kompilátor namiesto pridelenia pamäti potrebnej na reprezentáciu jednotlivých zložiek premennej vyhradí pevnú časť pamäte potrebnú na reprezentáciu adresy dynamickej premennej.

Pointer (smerník = ukazovateľ) je premenná, v ktorej je uložená adresa pamäte a na tejto adrese sa až nachádza prislúchajúci objekt (hodnota premennej, prvok poľa, inštancia štruktúry a pod.; Herout, 2010). Smerník je ako šípka, ktorá ukazuje niekam do pamäte. Okrem toho, kam smerník ukazuje, treba mu väčšinou určiť aj to, aký typ premennej tam môže očakávať. Takže sú smerníky, ktoré sú typu *int* pretože ukazujú na premenné typu *int*, sú smerníky typu *float*, typu *char* a pod. Ďalšou dôležitou skutočnosťou je, že pri pointeroch musíme pracovať len s pamäťou, ktorá je naša. V jazyku *C* nič túto skutočnosť nekontroluje.

Deklarácia ukazovateľa

Realizuje sa podobne, ako deklarácia inej premennej, ale použije sa znak hviezdičky '*'. Je nevyhnutné určiť typ, na ktorý ukazovateľ môže ukazovať. Často sa používa pred identifikátorom takejto premennej prefix *p_*, zdôrazňujúci skutočnosť, že ide o premennú typu pointer, napr.:

```
double *p_d;          // ukazovatel na double
int *p_i;             // ukazovatel na int
char *p_c;            // ukazovatel na char
```

Deklarácia spolu s viacerými premennými

```
int *p_i, p_j;        // casta chyba, len p_i je pointer, premenna p_j je typu int
```

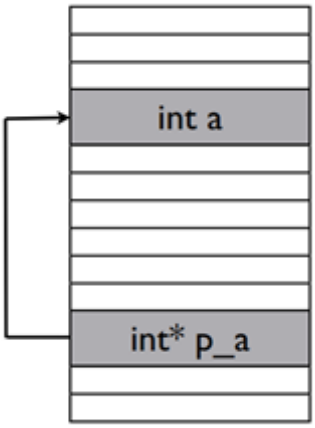
Inicializácia ukazovateľa

Inicializácia ukazovateľa *p_i* adresou premennej *i* sa realizuje s pomocou referenčného operátora '&'. V prípade, ak by sme pointer neinicializovali môžeme hovoriť o tzv. zablúdenom pointeri (*wild pointer*).

```
int a, *p_a = &a;      // pri deklaracii
```

alebo

```
int a, *p_a;
p_a = &a;            // v programe
```



Obr. 49 Grafická reprezentácia pointera a jeho inicializácie.

Práca s ukazovateľmi

Ak pointer *p_a* je inicializovaný premennou *a*, tak hodnotu premennej *a* s pomocou pointera môžeme vypísať:

```
int a, *p_a = &a;
a = 3;
printf("Ukazovatel p_n ukazuje na hodnotu %d.", *p_a);
```

Z uvedeného príkladu je zrejmé, že operátor referencie je '&' a dereferencovania je '*'.

Ak pointer *p_a* nie je pri deklarácii aj inicializovaný, ukazuje na náhodné miesto v pamäti.

```
int *p_a;
printf("%d", *p_a);  // chyba - nevieme kam p_a ukazuje
```

Ak do *p_a* zapíšeme literál, určili sme adresu kam *p_a* ukazuje. Takéto priradenie nie je samozrejme správne. Pretože vieme, že adresa v pamäti je reprezentovaná v 16-kovej (hexadecimálnej) číselnej sústave a tiež, že priraďovať napevno pamäťové úseky nie je vhodné.

```
int *p_a;
p_a = 10;        // chyba - p_a ukazuje na miesto v pamati s adresou 10
```

Príklad nižšie ilustruje rozdielne priradenia pri práci s pointermi:

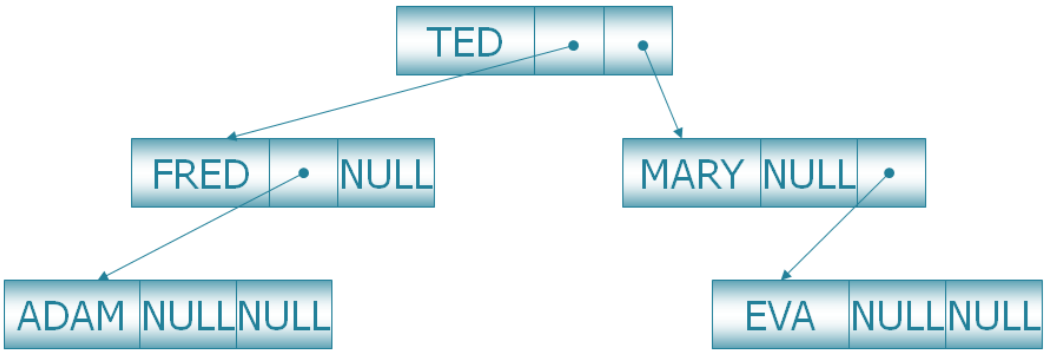
```
int m = 2, n = 4, *p1 = &m, *p2 = &n;
p1 = p2;          // kopirovanie adresy z p2 do p1
*p1 = *p2;        // kopirovanie hodnoty z pamate kam ukazuje p2 do pamate, kam ukazuje p1
```

Príklad nižšie ilustruje konverziu pointerov:

```
char znak = 'A', *p_znak = &znak;
int i, *p_i = &i;
p_znak = p_i;             // nevhodne pouzitie
p_znak = (char *) p_i;    // odporucane pouzitie
```

Ako sme uviedli vyššie, napríklad štruktúra *osoba* môže byť použitá na reprezentáciu rodokmeňovej štruktúry. Táto bude reprezentovaná prostredníctvom jednotlivých, pravdepodobne nespojitých záznamov. Pre každú osobu bude existovať práve jeden záznam. Tieto osoby sú v rámci štruktúry pospájané s pomocou adries priradených položkám *otec* a *matka*. Túto situáciu možno najlepšie graficky zobraziť s pomocou šípok, t. j. smerníkov. Pre smerník, ktorý nikam neukazuje sa používa hodnota *NULL* – prázdny smerník. V tomto prípade to reprezentuje koniec dátovej štruktúry.

```
typedef struct osoba
{
    char meno[20];
    struct osoba *otec;
    struct osoba *matka;
}OSOBA;
```



Obr. 50 Grafická reprezentácia rodokmeňovej štruktúry.

4.1.1 Konštantný smerník

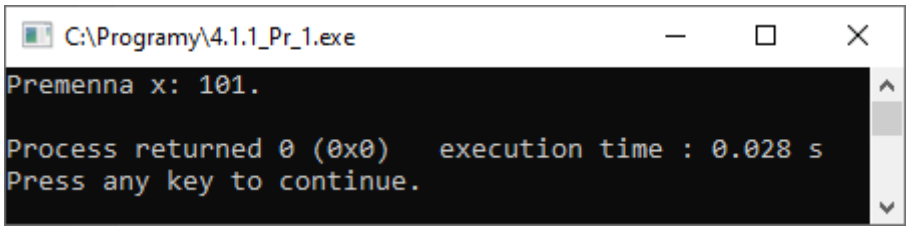
Konštantný smerník je smerník, ktorý po inicializácii nemôže meniť svoju hodnotu. Môže byť inicializovaný len raz. Smie meniť hodnotu objektu, na ktorý ukazuje.

Smerníkovú premennú do ktorej bude konštantný smerník uložený, deklarujeme a inicializujeme takto:

```
DatovyTyp * const  NazovSmernikovejPremennej = &Objekt;
```

Príklad 4.1.1_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    int x = 100, y = 200;
    int * const p_x = &x;
    (*p_x)++;   // moze menit hodnotu objektu, na ktory ukazuje
    printf("Premenna x: %d.\n", x);
    // p_x=&y;   // nemoze byt znovu inicializovany

    return 0;
}
```



Obr. 51 Konzolový výstup programu 4.1.1_Pr_1.c.

4.1.2 Smerník na konštantu

Smerník na konštantu je smerník, ktorý nemôže meniť hodnotu objektu, na ktorý ukazuje. Z pohľadu smerníka ide o konštantný objekt. Počet inicializácií smerníka na konštantu nie je obmedzený.

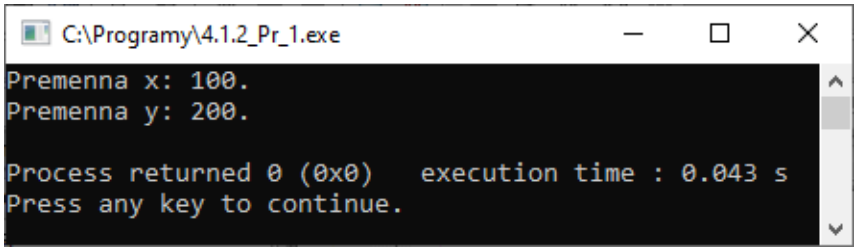
Smerníkovú premennú, do ktorej bude smerník na konštantu uložený, deklarujeme a inicializujeme takto:

```
DatovyTyp  const*  NazovSmernikovejPremennej = &Objekt;
```

Príklad 4.1.2_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    int x = 100, y = 200;
    int const * p_x = &x;
    // *p_x = 150;   // nemoze menit hodnotu objektu, na ktory ukazuje
    printf("Premenna x: %d.\n", *p_x);
```

```
9       p_x = &y;    // moze byt znovu inicializovany
10      printf("Premenna y: %d.\n", *p_x);
11
12      return 0;
13  }
```

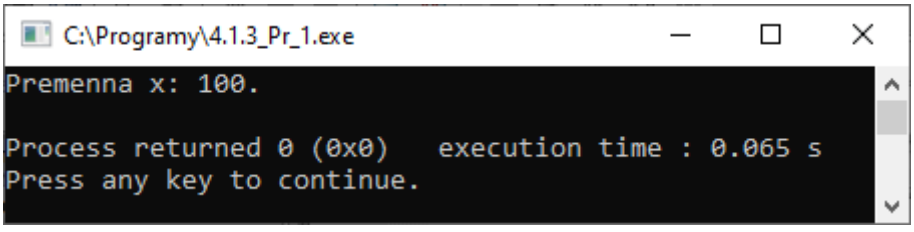


Obr. 52 Konzolový výstup programu 4.1.2_Pr_1.c.

4.1.3 Nulový a generický ukazovateľ

Nulový ukazovateľ (*NULL*) je smerník, ktorý nie je aktuálne nasmerovaný na žiaden cieľový objekt. Je definovaný ako 0, s pomocou symbolickej konštanty (makra) *NULL*. Je možne priradiť ho bez pretypovania k všetkým typom ukazovateľov. Príklad 4.1.3_Pr_1.c:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int x = 100;
6       int * p1;
7       void * p2;
8       int * p3 = NULL;     // nulovy pointer
9       p1 = &x;
10      p2 = p1;             // do generickeho pointera mozem ulozit iny pointer bez pretypovania
11      p3 = (int *) p2;     // tu je pretypovanie nevyhnutne
12      printf("Premenna x: %d.\n", *p3);
13
14      return 0;
15  }
```



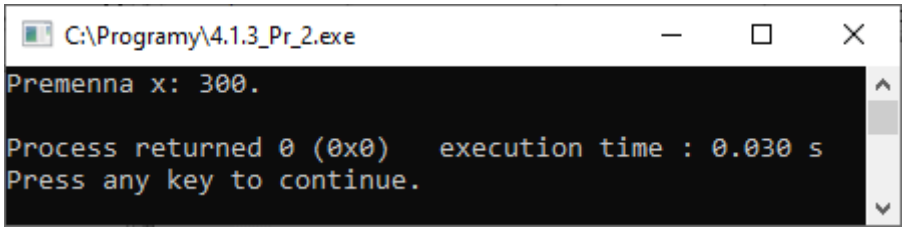
Obr. 53 Konzolový výstup programu 4.1.3_Pr_1.c.

Ukazovateľ na typ *void* neukazuje na žiadny konkrétny typ. Môže ukazovať na objekt ľubovoľného typu, tzv. všeobecný alebo generický ukazovateľ. Takýto smerník však nie je možné priamo dereferencovať '*'.

Príklad 4.1.3_Pr_2.c:

```
1   #include <stdio.h>
2
3   int main(void)
```

```c
{
    int x = 100;
    int * p1;
    void * p2;
    int * p3 = NULL;     // nulovy pointer
    p1 = &x;
    p2 = p1;             // do generickeho pointera mozem ulozit iny pointer bez pretypovania
    p3 = (int *) p2;     // tu je pretypovanie vhodne
    // *p2 = 200;        // nie je mozne priamo dereferencovat
    *(int*) p2 = 300;    // musim najpr realizovat pretypovanie

    printf("Premenna x: %d.\n", *p3);

    return 0;
}
```

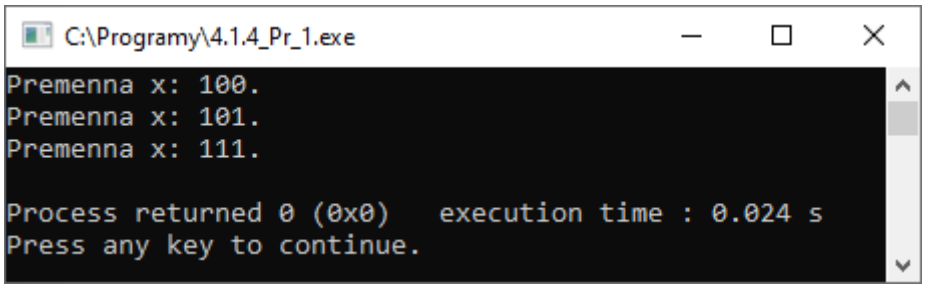


Obr. 54 Konzolový výstup programu 4.1.3_Pr_2.c.

4.1.4 Viacnásobná dereferencia

Je prípustné, aby sa smerník odkazoval na iný smerník. Tomuto mechanizmu sa hovorí viacnásobná dereferencia (resp. viacnásobná indirekcia). Príklad 4.1.4_Pr_1.c:

```c
#include <stdio.h>

int main(void)
{
    int x;
    int * p1;      // jednoduchy pointer
    int ** p2;     // dvojity pointer
    x = 100;
    printf("Premenna x: %d.\n", x);
    p1 = &x;       // inicializacia pointera p1 premennou x
    (*p1)++;       // zmena hodnoty premennej x s pomocou pointera p1
    printf("Premenna x: %d.\n", x);
    p2 = &p1;      // inicializacia pointera p2 pointerom p1
    **p2 += 10;
    printf("Premenna x: %d.\n", x);

    return 0;
}
```



Obr. 55 Konzolový výstup programu 4.1.4_Pr_1.c.

4.2 Pointery a funkcie

Pri odovzdávaní skutočných parametrov funkcie rozlišujeme:

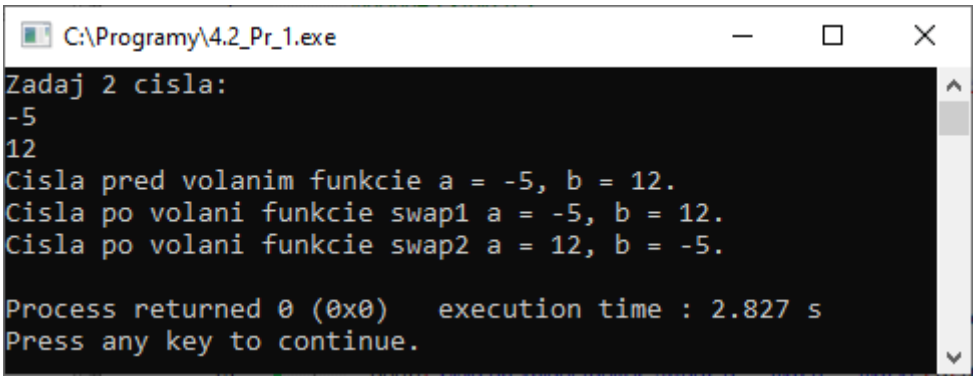
1. **Odovzdávanie parametrov hodnotou** = volanie hodnotou
   - Pri volaní funkcie sa odovzdáva hodnota parametra.
   - Funkcia nemôže trvalo meniť hodnotu skutočného parametra. Zmeny v parametri sú viditeľné iba v rámci funkcie.
2. **Odovzdávanie parametrov odkazom (referenciou)** = volanie odkazom
   - Pri volaní funkcie sa odovzdáva adresa danej premennej, nie jej hodnota.
   - Funkcia môže trvalo meniť hodnotu skutočného parametra.
   - Pri odovzdávaní parametrov odkazom je nevyhnutné formálny parameter definovať ako pointer a skutočný parameter uvádzať s operátorom '&'.
   - Ak sa má zmeniť hodnota skutočného parametra, ktorý je pointer, je nevyhnutné použiť formálny parameter ako pointer na pointer.

Príklad: Navrhnite funkciu, ktorá zrealizuje výmenu obsahu dvoch premenných: *swap1* – volanie hodnotou a *swap2* – volanie odkazom.

Jedna z možných implementácií 4.2_Pr_1.c:

```
1  #include <stdio.h>
2
3  void swap1(int a, int b);          // deklaracia funkcie - volanie hodnotou
4  void swap2(int *p_a, int *p_b);    // deklaracia funkcie - volanie odkazom
5
6  int main(void)
7  {
8      int a, b;
9      printf("Zadaj dve cisla:\n");
10     scanf("%d %d", &a, &b);
11     printf("Cisla pred volanim funkcie a = %d, b = %d.\n", a, b);
12     swap1(a, b);    // volanie hodnotou
13     printf("Cisla po volani funkcie swap1 a = %d, b = %d.\n", a, b);
```

```c
    swap2(&a, &b);  // volanie odkazom
    printf("Cisla po volani funkcie swap2 a = %d, b = %d.\n", a, b);

    return 0;
}

// definicia funkcie - volanie hodnotou
void swap1(int a, int b)
{
    int c;
    c = a;
    a = b;
    b = c;
}

// definicia funkcie - volanie odkazom
void swap2(int *p_a, int *p_b)
{
    int c;
    c = *p_a;
    *p_a = *p_b;
    *p_b = c;
}
```



Obr. 56 Konzolový výstup programu 4.2_Pr_1.c.

Pozn. autora: Z príkladu je zrejmé, že trvalá zmena skutočných parametrov sa prejaví len pri volaní odkazom. V jazyku *C* je však aj toto volanie hodnotou, pretože v zásobníku sa vytvorí lokálna kópia na uloženie parametra, t. j. adresy skutočného parametra. Táto lokálna premenná síce zaniká s ukončením prislúchajúcej funkcie, ale má tú vlastnosť, že v tomto pointeri je uložená adresa skutočného parametra, s pomocou ktorého sa nepriamo zmenia údaje počas vykonávania tela funkcie, ktoré nemajú s touto funkciou nič spoločné – boli definované mimo tejto funkcie a nezanikajú s jej koncom. Pri volaní hodnotou – *swap1()* sú zmeny len lokálne, ako sme očakávali.

Pozor na častú chybu študentov – je chybou vrátiť vo funkcii adresu lokálnej premennej, napr.:

```
int* funkcia(void)
{
    int i;
    return &i;
}
```

Táto skutočnosť je chybou, pretože premenná *i* je lokálnou premennou a po vykonaní funkcie *funkcia()* táto premenná zanikne.

4.3 Aritmetické operácie s ukazovateľmi = pointerová aritmetika

Medzi platné operácie s pointermi zaraďujeme:

- Súčet pointera a celého čísla.
- Rozdiel pointera a celého čísla.
- Rozdiel dvoch pointerov rovnakého typu.
- Porovnanie pointerov rovnakého typu.

Nesmieme zabudnúť aj na unárne operácie inkrementácie a dekrementácie. Pointerovú aritmetiku využívame zvyčajne pri práci s poľami, t. j. pamäťovými blokmi, ktoré sú zložené z premenných rovnakého typu ležiacich v pamäti za sebou. Často sa tu využíva fakt, že identifikátor premennej typu pole je pointer, ktorý obsahuje adresu prvého prvku poľa. Použitie pointerovej aritmetiky pri práci s poľom vraj urýchľuje program voči použitiu napr. štandardného prístupu k prvkom poľa s pomocou indexov.

Krátke vysvetlenie týchto operácií uvediem na príklade 4.3_Pr_1.c:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      // deklaracia pola s inicializaciou, pointerov (p, q, r) a premennych typu int (i, d)
6      int a[10] = {1,2,3,4,5,6,7,8,9,10}, *p, *q, *r, i, d;
7      printf("Prvky pola: ");
8      for (i = 0; i < 10; i++)
9      {
10         printf("%d ", a[i]);
11     }
12     printf("\n");
13
14     p = a;          // pointer p reprezentuje zaciatok pola
15     *p = 10;        // cez pointer p prepisem hodnotu prveho prvku pola
16     q = &a[4];      // do q sa ulozi adresa piateho prvku pola
17     printf("Piaty prvok pola %d.\n", *q);
18     r = &a[6];      // do r sa ulozi adresa siedmeho prvku pola
19     printf("Siedmy prvok pola %d.\n", *r);
20
```

```
21      // pricitanie celeho cisla k ukazovatelu
22      r += 2;         // r ukazuje na prvok a[8]
23      printf("Deviaty prvok pola %d.\n", *r);
24
25      // odcitanie celeho cisla od ukazovatela
26      q -= 1;         // q ukazuje na prvok a[3]
27      printf("Stvrty prvok pola %d.\n", *q);
28
29      // odcitanie dvoch ukazovatelov
30      d = r - q;      // d je 5
31      // d = q - r; // pozor d je -5
32      printf("Pocet prvok medzi a[8] a a[3] je %d.\n", d-1);
33
34      // porovnanie dvoch ukazovatelov + unarna operacia inkrementacie
35      p = a;          // zaciatok pola
36      q = &a[9];      // koniec pola
37      printf("Prvky pola: ");
38      for (p; p <= q; p++)
39      {
40          printf("%d ", *p);
41      }
42
43      return 0;
44  }
```

```
C:\Programy\4.3_Pr_1.exe
Prvky pola: 1 2 3 4 5 6 7 8 9 10
5-ty prvok pola 5.
7-mi prvok pola 7.
9-ty prvok pola 9.
4-ty prvok pola 4.
Pocet prvok medzi a[8] a a[3] je 4.
Prvky pola: 10 2 3 4 5 6 7 8 9 10
Process returned 0 (0x0)   execution time : 0.077 s
Press any key to continue.
```

Obr. 57 Konzolový výstup programu 4.3_Pr_1.c.

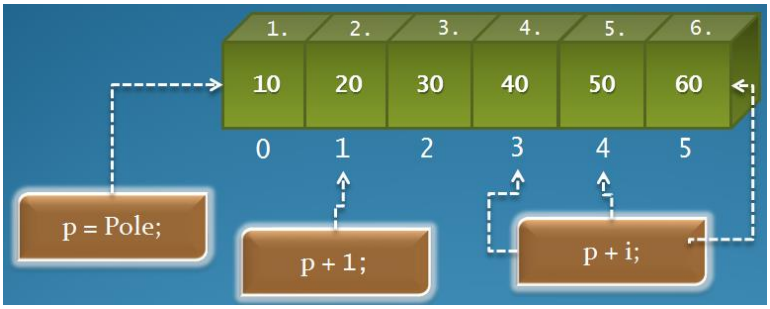
Pozn. autora: S pomocou rozdielu pointerov, vieme zistiť počet prvkov poľa ležiacich medzi určenými prvkami, musíme však odpočítať ešte 1, aby hodnota korešpondovala s tým, že chceme len prvky ležiace medzi určenými prvkami poľa.

Ako je zrejmé z príkladu, práca s poľom a práca s ukazovateľom je skoro rovnaká:

- k prvkom poľa môžeme pristupovať s pomocou indexov: *Pole[index]*
- výrazy *&Pole[i]* a *p + i* sú ekvivalentné (prístup k *i*-temu prvku poľa), *Pole* môže byť deklarované napr. ako `int Pole[10];` *p* ako `int *p;`
- `int Pole[10];  // deklaracia statickeho pola`
  premenná *Pole* ukazuje na alokované pamäťové miesto a je konštantná – jej hodnotu (adresu) nie je možné meniť (nie je to *l-value*)

- `int *p = Pole;` `int *p = &Pole[0];` `// definicia ukazovatela na int`

  premenná *p* pri deklarácii nemá určenú počiatočnú hodnotu, pamäť, kam ukazuje nie je alokovaná, ak *p* inicializujeme ihneď pri deklarácii je možné do *p* zapísať adresu (je to *l-value*)



Obr. 58 Grafická ilustrácia pointerovej aritmetiky.

Všimnite si rozdiely medzi naplnením jednorozmerného statického poľa s pomocou indexovej notácie a pointerovej aritmetiky:

Indexová notácia:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int Pole[5], i;
6
7      for (i = 0; i <= 4; i++)
8            Pole[i] = i;
9
10     return 0;
11 }
```

Pointerová aritmetika:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int Pole[5], i;
6
7      int* p_pole = Pole;
8      for (i = 0; i <= 4; ++i)
9          *(p_pole + i) = i;
10
11     return 0;
12 }
```

4.3.1 Príklad – pointerová aritmetika – pole a funkcie

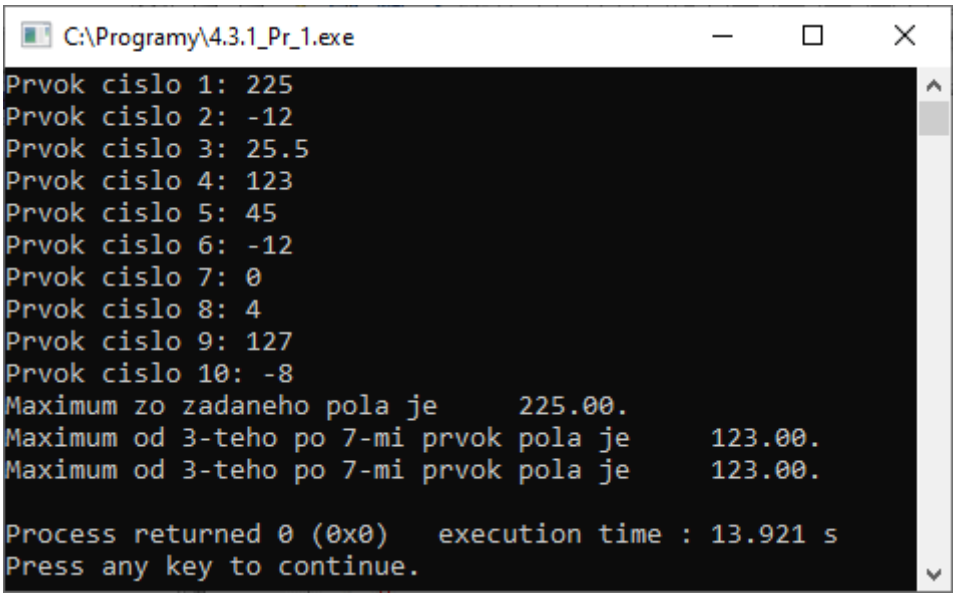
Zadanie problému: Vytvorte program, ktorý s pomocou funkcie nájde maximálny prvok zadaného poľa. Funkciu navrhnite tak, aby bolo možné zisťovať maximálny prvok aj z častí daného poľa.

Pozn. autora: Pole môže pôsobiť ako argument, ktorý bude odovzdaný funkcii pri jej volaní. V tomto prípade je pole odovzdávané vždy odkazom, keďže v skutočnosti funkcii nieje poskytnuté pole, ale smerník ukazujúci na prvý prvok tohto poľa. Aby sme splnili druhú časť zadania (možnosť hľadať maximum aj z častí poľa), budeme musieť vo funkcii využiť na posun po poli pointerovú aritmetiku.

Jedna z možných implementácií 4.3.1_Pr_1.c:

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10

double max(double *pole, int n);
double max_2(double *zaciatok_pola, double *koniec_pola);

int main(void)
{
    double a[N];
    int i;

    for (i = 0; i < N; i++)
    {
        printf("Prvok cislo %d: ",i + 1);
        scanf("%lf",&a[i]);
    }

    printf("Maximum zo zadaneho pola je %10.2lf. \n", max(a, N));
    printf("Maximum od tretieho po siedmy prvok pola je %10.2lf. \n", max(a + 2, 5));
    printf("Maximum od tretieho po siedmy prvok pola je %10.2lf. \n", max_2(a + 2, &a[7]));

    return 0;
}
// definicia funkcie na najdenie maxima
double max(double *pole, int n)
{
    double max = *pole, *p_pom;    // inicializacia max na prvy prvok pola

    // prehladavame uz len zvysok pola
    for (p_pom = pole + 1; p_pom < pole + n; p_pom++)
    {
        if (*p_pom > max)
            max = *p_pom;
    }
    return max;
}

// definicia funkcie na najdenie maxima
double max_2(double *zaciatok_pola, double *koniec_pola)
{
    double max = *zaciatok_pola, *p_pom;   // inicializacia max na prvy prvok pola

    // prehladavame uz len zvysok pola
    for (p_pom = zaciatok_pola + 1; p_pom < koniec_pola; p_pom++)
    {
        if (*p_pom > max)
            max = *p_pom;
    }
    return max;
}

/*
// definicia funkcie na najdenie maxima - ina alternativa
double max(double pole[], int n)
//double max(double *pole, int n)
```

```
57  {
58      double *p_max = pole, *p_pom;  // inicializacia pointera, ktory reprezentuje maximum
59
60      for (p_pom = pole + 1; p_pom < pole + n; p_pom++){
61          if (*p_pom > *p_max)
62            p_max = p_pom;
63      }
64      return (*p_max);
65  }
66  */
```



Obr. 59 Konzolový výstup programu 4.3.1_Pr_1.c.

Zo skutočnosti, že sa funkcii odovzdáva s pomocou skutočného parametru iba adresa začiatku poľa, vyplýva, že je jednoduché funkciu použiť i na prácu s úsekmi tohto poľa. Druhé volanie funkcie *max(),* riadok 20 ilustruje, ako je možné meniť začiatok poľa v ktorom sa bude hľadať maximum.

Zložitejšie je už naplnenie požiadavky, aby sme vedeli modifikovať aj koniec poľa, t. j. zadefinovať z ktorého úseku poľa (ohraničeného zdola, resp. zhora) má byť maximum hľadané. Podľa stanového princípu algoritmu a posuvu po prvkoch poľa s pomocou pointerovej aritmetiky, vieme správnym volaním funkcie už teraz realizovať túto skutočnosť. To ilustruje volanie funkcie na riadku 20. Pre niekoho by sa takéto volanie mohlo zdať zmätočné, druhý parameter vlastne určuje, z koľkých prvkov má byť maximum od definovaného začiatku úseku s pomocou prvého parametra funkcie hľadané.

Ak sa Vám to stále zdá zmätočné, navrhla som aj inú alternatívu tejto funkcie *max_2(),* pri ktorej sa algoritmus vôbec nezmenil, nastala len zmena parametrov funkcie. Prvý parameter reprezentuje začiatok úseku poľa, tak ako aj vo funkcii *max()*, druhý parameter na rozdiel od

funkcie *max()* reprezentuje koniec úseku poľa. V takomto prípade stačí pri volaní funkcie odovzdať len začiatok a koniec úseku poľa, ako ilustruje volanie funkcie na riadku 21.

Samozrejme, vhodnou zmenou relačného operátora pri hľadaní maxima vo funkcii vieme pracovať na otvorenom, resp. uzatvorenom intervale.

Pozn. autora: Ako vyplýva zo zadania problému, v tomto prípade sme sa vôbec nesústredili na algoritmické spracovanie úlohy. Toto by totiž bolo vo svojej podstate viac-menej to isté, ako v príklade 1.1.2 – kde hľadáme extrém (minimum) v jednorozmernom poli. Rozdiel by sa prejavil pri volaní funkcie na hľadanie extrému a tiež v nastavení cyklu *for,* s pomocou ktorého je zabezpečené prechádzanie prvkov poľa. Algoritmus (podstata riešenia) však vo svojej podstate ostáva rovnaký.

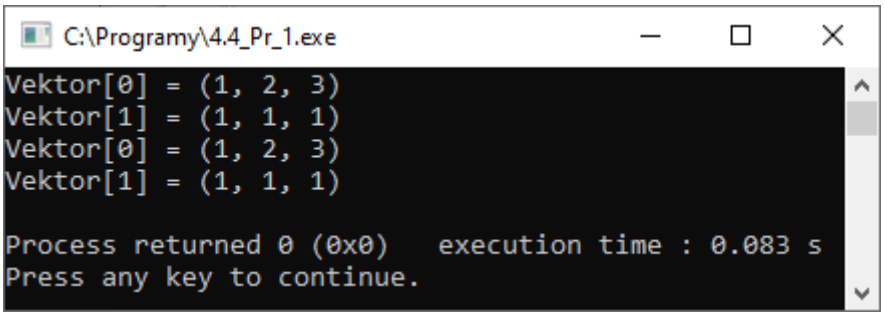
4.4 Pole štruktúr

Ako som už ilustrovala v kapitole 1.3, prvkom poľa môže byť aj inštancia štruktúry, podobne ako inštancia akéhokoľvek iného používateľsky definovaného dátového typu. K dátovým členom inštancií umiestnených v poliach možno pristupovať s pomocou smerníka a špeciálneho operátora šípky '−>'. Ako je však z príkladu zrejmé, stále je možné pristupovať k jednotlivým položkám štruktúry aj cez bodkovú notáciu. Dané skutočnosti ilustrujem na jednoduchom príklade, kde v poli uchovávame súradnice vektorov, ktoré sú reprezentované súradnicami koncového bodu *x*, *y* a *z*.

Príklad 4.4_Pr_1.c:

```
1  #include <stdio.h>
2
3  // definicia typu struktury
4  typedef struct vektor
5  {
6      int x, y, z;
7  } VEKTOR;
8
9  int main(void)
10 {
11     // deklaracia pola, prvky su instancie struktury
12     VEKTOR vektory[2];
13
14     // definicia dvoch pointerov ako ukazka roznych moznosti pristupu k prvkom pola
15     VEKTOR * p_vektor1 = &vektory[0];
16     VEKTOR * p_vektor2 = &vektory[1];
17
18     // naplnenie suradnic vektorov
19     p_vektor1->x = 1;
20     p_vektor1->y = 2;
21     p_vektor1->z = 3;
```

```
22      p_vektor2->x = p_vektor2->y = p_vektor2->z = 1;
23
24      // pouzitie premennej typu struktury - bodkova notacia, pristup k prvkom pola cez index
25      printf("Vektor[0] = (%d, %d, %d)\n", vektory[0].x, vektory[0].y, vektory[0].z);
26      printf("Vektor[1] = (%d, %d, %d)\n", vektory[1].x, vektory[1].y, vektory[1].z);
27
28     // pouzitie premennej typu struktury - sipkova notacia, pristup k prvkom pola cez pointer
29      printf("Vektor[0] = (%d, %d, %d)\n", p_vektor1->x, p_vektor1->y, p_vektor1->z);
30      printf ("Vektor[1] = (%d, %d, %d)\n", p_vektor2->x, p_vektor2->y, p_vektor2->z);
31
32      return 0;
33  }
```



Obr. 60 Konzolový výstup programu 4.4_Pr_1.c.

4.5 Dynamické prideľovanie a navracanie pamäti

Najpoužívanejšou funkciou na prideľovanie pamäte v jazyku *C* je funkcia, deklarovaná v *stdlib.h*, ktorej prototyp je:

`void *malloc(size_t size)` ktorá:

- Alokuje súvislý blok pamäte požadovanej veľkosti (*size_t* = *unsigned int*) (tento parameter udáva, koľko bajtov chceme alokovať) a vráti generický ukazovateľ na tento blok. Tento pointer je vhodné pretypovať na pointer zodpovedajúceho typu.
- Pamäť sa alokuje na halde (hromada, heap), nie na zásobníku, až počas vykonávania sa programu.
- Ak nie je v pamäti dosť miesta na pridelenie požadovanej časti, vracia funkcia hodnotu *NULL*. Pri každom prideľovaní pamäti, sa odporúča testovať návratovú hodnotu na *NULL* a nespoliehať sa na fakt, že pamäti je dostatok.

Ukážka použitia *malloc()* vrátane reakcie na prípadný neúspech:

```
int *p_i;
if ((p_i = (int *) malloc(1000)) == NULL)
{
    printf("Nedostatok pamate. Program konci.\n");
    exit(1);
}
```

Pozn. autora: V tomto prípade išlo o deklaráciu bloku s veľkosťou 1 000 B. Napriek tomu, že syntakticky je táto alokácia správna, v praxi bežne nepoužívame alokáciu priestoru s pomocou pevne stanovej veľkosti, ale využívame operátor *sizeof()*. Toto použitie si ukážeme neskôr. V prípade, ak by nebolo dostatok pamäte, dôjde k ukončeniu programu s pomocou funkcie *exit()*, predtým je však používateľ náležite informovaný o tom, čo sa deje.

Uvoľnenie (vrátenie pamäti) je opačná akcia než prideľovanie. Platí, že nepotrebnú pamäť je dobré okamžite vrátiť a nečakať až na koniec programu. Na uvoľnenie pamäte sa používa funkcia `void free(void *)`, ktorej parametrom je pointer na typ *void*, ktorý ukazuje na začiatok skôr prideleného bloku pamäte. Dôležité je, že *free()* nemení hodnotu svojho parametru. To znamená, že pointer stále ukazuje na to isté miesto v pamäti, takže je možné s touto pamäťou ďalej pracovať, aj keď už programu nepatrí. To môže spôsobiť veľa problémov, preto po príkaze:

```
free((void *) p_i);
```

je vhodné uviesť aj príkaz:

```
p_i = NULL;
```

čím zabránime možnému prístupu do uvoľnenej pamäte, ako aj zabránime problému, ktorý by mohol nastať pri dvojitom aplikovaní funkcie *free()* na ten istý smerník za sebou bez pridelenia novej pamäte. V tomto prípade by nastalo zrútenie sa programu. Avšak zavolanie funkcie *free()* s pointerom, ktorý je nastavený na hodnotu *NULL* je bezpečné.

Pri dynamickom prideľovaní a uvoľňovaní pamäte často vznikajú nasledujúce chyby:

Majme deklaráciu dvoch pointerov:

```
char *p_c;
int *p_i;
```

Potom príkaz:

```
*p_c= 'a';
```

nie je úplne korektný, pretože *p_c* ukazuje niekam do pamäte, ktorú nemáme pridelenú. Pred týmto príkazom je teda nevyhnutné uviesť príkaz, aby sme alokovali dostatočný priestor pamäte:

```
p_c = malloc(1);
```

Aby sme dodržiavali odporúčania, je vhodné uviesť explicitnú typovú konverziu:

```
p_c = (char *) malloc(1);
```

Aby sme dodržiavali všetky odporúčania, je treba ešte zistiť, či sa požadovanú pamäť podarilo priradiť:

```
if (((char *)p_c = malloc(1)) == NULL)
{
        printf("Nieje volna pamat. \n");
        return;
}
```

Ak chceme následne alokovať pamäť s veľkosťou 20 B prístupného s pomocou rovnakého pointera *p_c*, tak príkaz:

```
p_c = (char*) malloc(20);
```

nie je úplne vhodný pretože takto sme stratili pointer na skôr alokovanú pamäť (v ktorej je znak 'a'). Túto pamäť sa už nikdy nepodarí uvoľniť a do konca programu bude znak 'a' visieť niekde v pamäti. Preto pred každou ďalšou alokáciou je potrebné pamäť uvoľniť:

```
free(p_c);
```

Ak potrebujeme alokovať pamäť na uloženie napr. *int* hodnoty, tak je najvhodnejší nasledujúci príkaz:

```
p_i = (int *) malloc(sizeof(int));
```

t. j. využitie operátora *sizeof()*.

Ak používame pri definícii typov operátor *typedef*, je možné, že pri alokovaní pamäte sa stretneme aj s nasledujúcim spôsobom alokovania:

```
typedef int * P_INT;
P_INT p_i;
p_i = (P_INT) malloc(sizeof(int));
```

alebo

```
p_i = (P_INT) malloc(sizeof(*p_i));
```

Okrem funkcie *malloc()* môžeme pamäť dynamicky alokovať aj prostredníctvom funkcie *calloc()*. Je deklarovaná v *stdlib.h* a jej prototyp vyzerá takto:

`void* calloc(size_t pocet, size_t velkost)`, kde:

- *pocet* predstavuje počet prvkov dynamického poľa,
- *velkost* je nominálna alokačná kapacita jedného prvku dynamického poľa (v bajtoch).

Ak je volanie funkcie *calloc()* úspešné, funkcia alokuje pamäťový blok pre dynamické pole, pričom jednotlivé prvky poľa inicializuje na 0. Ak nie je úspešné, funkcia vráti nulový smerník (*NULL*). Návratovou hodnotou funkcie *calloc()* je generický smerník (*void* *) ukazujúci na začiatok dynamicky alokovanej pamäte. Použitie funkcie *calloc()* ilustruje nasledujúci fragment kódu:

```
...
printf("Kolko cisel chces zadat? ");
scanf("%d", &n);
p = (int*) calloc(n, sizeof(int));
```

```
if (p == NULL){
    printf("Nedostatok pamatovych prostriedkov.\n");
    exit(1);
}
...
```

4.5.1 Príklad – dynamické prideľovanie a uvoľňovanie pamäte

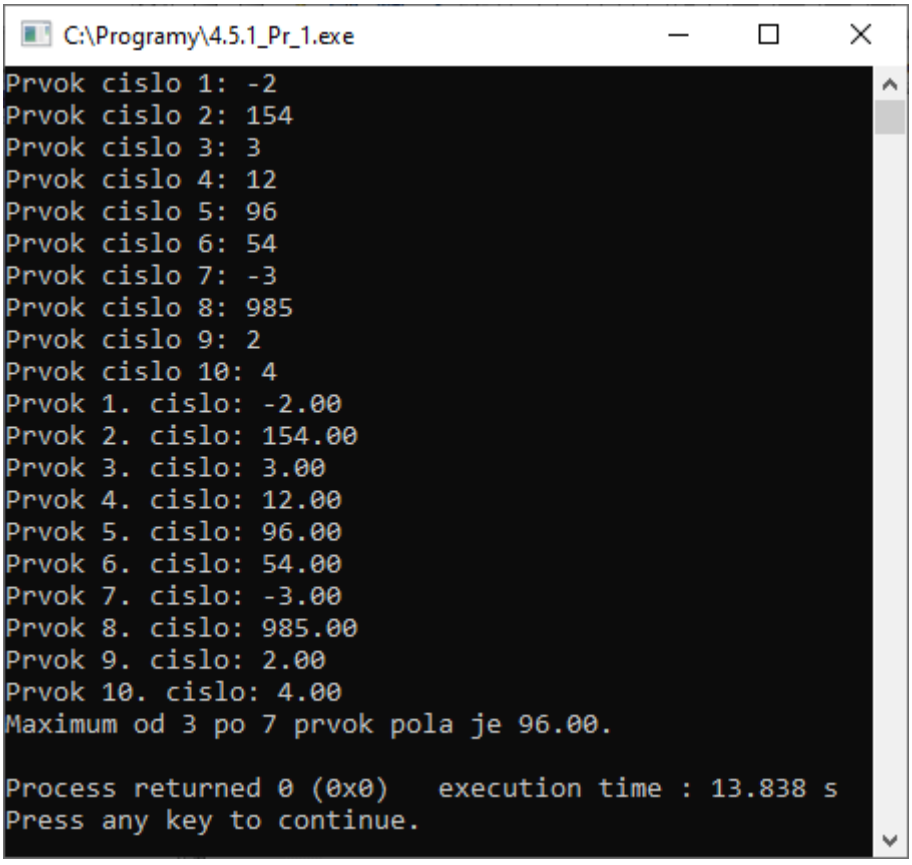
Zadanie problému: Vytvorte program, ktorý s pomocou funkcie nájde maximálny prvok zadaného poľa. Funkciu navrhnite tak, aby bolo možné zisťovať maximálny prvok aj z častí daného poľa.

Pozn. autora: Môžete si všimnúť, že zadanie problému je identické ako v kapitole 4.3.1. Je tak z dôvodu, že ak si uvedomujeme skutočnosť, že by sme mali s pamäťou narábať hospodárne, nikto nám nemusí exaktne hovoriť, že je vhodné pracovať s dynamickým poľom. Je zrejmé aj to, že dynamická alokácia pamäte, resp. dealokácia nemá žiadny vplyv na algoritmické spracovanie úlohy. Z tohto dôvodu pri tomto príklade neuvádzam grafickú reprezentáciu algoritmu. Táto by bola identická ako v príklade v kapitole 1.1.2.

Jedna z možných implementácií 4.5.1_Pr_1.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define N 10
4
5  double max(double *zaciatok_pola, double *koniec_pola);     // deklaracia funkcie
6
7  int main(void)
8  {
9      double *p_pole;    // deklaracia pointera reprezentujuceho pole
10     int i;
11
12     // dynamicka alokacia pola
13     p_pole = (double *) malloc(N*sizeof(double));
14     if (p_pole == NULL)
15     {
16         printf("Pole sa nepodarilo vytvorit. Nedostatok pamate. Program sa konci.\n");
17         return 1;
18     }
19
20     // naplnenie pola
21     for (i = 0; i < N; i++)
22     {
23         printf("Prvok cislo %d: ", i + 1);
24         scanf("%lf",(p_pole + i));
25     }
26
27     // vypis pola
28     for (i = 0; i < N; i++)
29     {
30         printf("Hodnota %d. cislo: %.2lf\n", i + 1, p_pole[i]);
```

```c
    }

    printf("Maximum od 3. po 7. prvok pola je %0.2lf. \n", max(p_pole + 2, &p_pole[7]));

    // uvolnenie alokovanej pamate
    free(p_pole);
    p_pole = NULL;
    return 0;
}

// definicia funkcie na najdenie maxima
double max(double *zaciatok_pola, double *koniec_pola)
{
    double maximum = *zaciatok_pola, *p_pom;    // inicializacia max na prvy prvok pola

    // prehladavame uz len zvysok pola
    for (p_pom = zaciatok_pola + 1; p_pom < koniec_pola; p_pom++)
    {
        if (*p_pom > maximum)
            maximum = *p_pom;
    }
    return maximum;
}
```



Obr. 61 Konzolový výstup programu 4.5.1_Pr_1.c.

4.5.2 Dynamická realokácia

Realokácia znamená zmenu veľkosti (zväčšenie alebo zmenšenie) dynamicky alokovaného pamäťového bloku, ktorý bol pridelený alokačnými funkciami *malloc()* resp. *calloc()*. Realokáciu dynamicky alokovanej pamäte uskutočňuje funkcia *realloc()*, ktorá je deklarovaná v *stdlib.h*. Prototyp funkcie realloc vyzerá takto:

`void * realloc(void * PamatovyBlok, size_t velkost)`, kde:

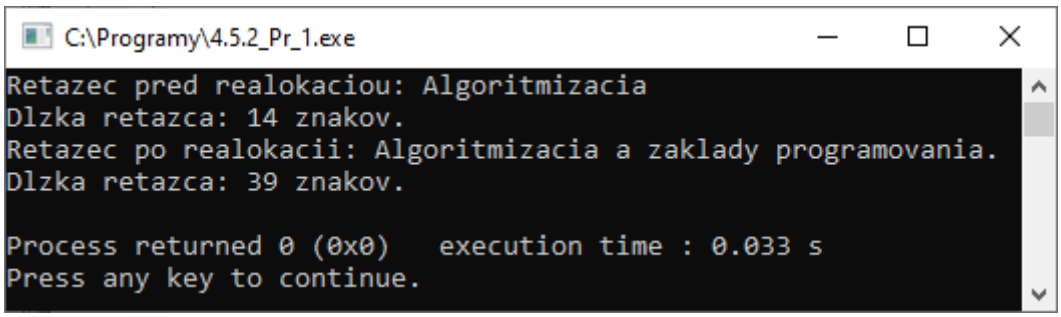
- *PamatovyBlok* je generický smerník na už dynamicky alokovanú pamäťovú oblasť.
- *velkost* predstavuje novú alokačnú kapacitu pamäťového bloku v bajtoch.

Ak je dynamická realokácia úspešná, návratovou hodnotou funkcie *realloc()* je generický smerník na realokovaný pamäťový blok. Ak nie je dostatok dodatočného pamäťového priestoru, funkcia *realloc()* vracia nulový smerník (*NULL*).

Ukážka použitia funkcie *realloc()* na príklade 4.5.2_Pr_1.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // deklaracia pointera na char
    char * p;

    // alokacia priestoru pre 20 znakov
    if ((p = (char *) malloc(20*sizeof(char))) == NULL)
    {
        printf("Nedostatok pamate. Program sa konci.\n");
        return 1;
    }

    // naplenie priestoru retazcom
    strcpy(p, "Algoritmizacia");
    printf("Retazec pred realokaciou: %s\n", p);
    printf("Dlzka retazca: %d znakov.\n", strlen(p));

    // realokacia priestoru pre 40 znakov
    if ((p = (char *) realloc(p, 40)) == NULL)
    {
        printf("Nedostatok pamate. Program sa konci.\n");
        free(p);     // uvolnenie predtym alokovanej pamate
        p = NULL;
        return 1;
    }

    // doplnenie existujuceho retazca o dalsi
    strcat(p, " a zaklady programovania.");
    printf("Retazec po realokacii: %s\n", p);
    printf("Dlzka retazca: %d znakov.\n", strlen(p));
```

```
35
36      free(p);
37      p = NULL;
38
39      return 0;
40  }
```



Obr. 62 Konzolový výstup programu 4.5.2_Pr_1.c.

4.6 Jednorozmerné dynamické pole

Chápeme ním dynamickú údajovú štruktúru – kontajner (úložisko dát), ktorá automaticky prispôsobuje veľkosť poľa v prípade pokusu o vloženie nového prvku pri zaplnenej kapacite. Zväčšenie kapacity dynamického poľa môže byť vykonané pre viac položiek, alebo iba jednu položku. Zväčšovanie kapacity pre jednu položku je málo efektívne, keďže je nevyhnutné vykonávať túto operáciu pri každom pridaní novej položky – pričom na pozadí je prekopírovaný celý obsah poľa. Z dôvodu šetrenia pamäťou, pri odstraňovaní položiek môže byť zabezpečené uvoľnenie prebytočnej pamäte. Záleží však od kontextu ako sa pole využíva, pretože ak sa využíva ako kontajner s pevnou veľkosťou, tak sa jeho kapacita zmenšuje len málokedy.

Jedným z variantov šetrenia pamäte je ten, že pri vkladaní nových položiek sa cyklicky prechádza pole a hľadá sa voľné – prázdne miesto. Ak také neexistuje, kapacita poľa sa zväčšuje. Takáto implementácia je však možná len v prípade, ak vieme pri odstraňovaní prvkov určiť jednoznačný identifikátor, ktorý signalizuje, že dané miesto je prázdne. Tento spôsob nie je príliš využívaný. Pri odstraňovaní prvku sa prvky ležiace za odstraňovaným prvkom posúvajú častejšie.

Z uvedeného vyplýva, že najčastejšími operáciami nad dynamickým poľom sú:

- Vyhraď pamäť pre oblasť nového poľa. Kapacita (veľkosť) poľa môže byť určená používateľom, alebo môže byť inicializovaná priamo programátorom na východiskovú hodnotou, ktorá logicky vyplynula zo špecifikácie.

- Vloženie položky na pozíciu *pozicia* v poli. Najjednoduchšie pridávanie je na začiatok poľa. V takomto prípade pomocnú premennú *pozicia* inicializujeme väčšinou hodnotou 0 a po každom úspešnom vložení prvku ju inkrementujeme, prípadne ju môžeme inicializovať hodnotou −1 a inkrementovať pred vložením prvku.
- Ak je aktuálny počet položiek + 1 > kapacita poľa => zväčši kapacitu poľa. Ako sme naznačili vyššie, funkcionalita zväčšovania kapacity je úplne závislá od špecifikácie problému.
- Odstránenie položky z poľa na pozícii *index,* alebo inej (určenej kľúčovou hodnotou). Operácia odstraňovania v poli je značne komplikovaná, pretože vyžaduje reorganizáciu položiek. V niektorých prípadoch môže viesť aj k zmene kapacity poľa. Jedným z univerzálnych spôsobov riešenia je reorganizácia položiek s pomocou operácie:
    - Urob posun položiek v poli od pozície *index*, prípadne od kľúčovej položky (t. j. tá ktorá sa ide odstraňovať) na pozíciu o 1 menšiu. Ak sme však realizovali vkladanie cez pomocnú premennú *pozicia*, treba túto premennú následne dekrementovať. Iná situácia by samozrejme bola v prípade, ak by sme proces vkladania jednotlivých položiek riešili cyklickým prehľadávaním voľných miest.
- V prípade budovania usporiadaného dynamického poľa sa často stretáme s operáciou:
    - Urob posun položiek v poli od pozície *index* na pozíciu o 1 väčšiu.

Výhodou dynamického poľa oproti statickému je, že kapacita poľa nie je ohraničená pri jeho vytvorení. Nevýhodou statického poľa oproti dynamickému je tiež skutočnosť, že sa preň rezervuje pamäť, ktorá nemusí byť nikdy využitá.

Medzi problémové operácie nad dynamickým poľom zaraďujeme:

- Vkladanie položky na ľubovoľnú pozíciu inú než začiatok a koniec poľa – nevyhnutná reorganizácia položiek poľa.
- Odstránenie položky z ľubovoľnej pozície inej ako je začiatok a koniec poľa – vznik voľných miest v poli, resp. reorganizácia poľa.

4.6.1 Príklad – dynamické neusporiadané pole

Zadanie problému: Vytvorte program, ktorý bude poskytovať možnosť ukladania celých čísiel do dynamického poľa. Tento bude slúžiť ako kontajner, čiže jeho kapacita, ktorá bude na začiatku zadaná používateľom, sa meniť nebude. V prípade potreby vloženia prvku pri nedostatočnej kapacite sa táto automaticky zväčší (zdvojnásobí).

Pozn. autora: Medzi základné operácie, ktoré sa pri použití kontajnerov používajú, patrí vloženie prvku, odstránenie prvku, zobrazenie obsahu, vyhľadanie konkrétnej položky, ale aj možnosť zistenia, či je kontajner plný, prázdny, aká je jeho kapacita a pod. V ilustračnom príklade som implementovala len niektoré funkcionality. Ostatné, resp. vylepšenie existujúcich už extistujúcich ponechávam na čitateľa.

Jedna z možných implementácií 4.6.1_Pr_1.c:

```c
#include <stdio.h>
#include <stdlib.h>

// deklaracie funkcii
void vypisPola(int *p_pole, int pozicia);
int sekvencneVyhladanie(int *p_pole, int pozicia, int prvok);
void vlozPrvok(int prvok, int *pozicia, int *p_pole, int *velkost);
int odstranPrvok(int prvok, int *pozicia, int *p_pole);

int main(void)
{
    // deklaracia pointera reprezentujuceho pole a dalsich premennych
    int pozicia = 0;
    int *p_pole, velkost, volba, odpoved = 0, prvok, i, index;

    // alokacia priestoru (kapacity pola) na zaklade volby pouzivatela
    do
    {
        printf("Ake velke ulozisko si zelas vytvorit?: \n");
        scanf("%d", &velkost);
    }
    while (velkost <= 0);

    if ((p_pole = (int *) malloc(sizeof(int) * velkost)) == NULL)
    {
        printf("Nedostatok pamate. Program sa konci.\n");
        return 1;
    }
    // menu
    do
    {
        printf("1 - vlozenie prvku do pola.\n");
        printf("2 - odstranenie prvku z pola.\n");
        printf("3 - vypisanie obsahu pola.\n");
        printf("4 - hladanie v poli.\n");
        printf("5 - koniec programu.\n");

        scanf("%d", &volba);
        switch(volba)
        {
        case 1:
            printf("Aky prvok chces vlozit do kontajnera?:\n");
            scanf("%d", &prvok);
            vlozPrvok(prvok, &pozicia, p_pole, &velkost);
            break;
        case 2:
            if (pozicia == 0)
```

```
48                  printf("Tvoj kontajner je prazdny, nemas co odstranovat.\n");
49              else
50              {
51                  printf("Aky prvok chces odstranit z kontajnera?:\n");
52                  scanf("%d", &prvok);
53                  if (odstranPrvok(prvok, &pozicia, p_pole) == 0)
54                      printf("Taky prvok sa v kontajneri nenachadza.\n");
55              }
56              break;
57          case 3:
58              if (pozicia == 0)
59                  printf("Tvoj kontajner je prazdny, nemam co vypisat.\n");
60              else
61                  vypisPola(p_pole, pozicia);
62              break;
63          case 4:
64              if (pozicia == 0)
65                  printf("Tvoj kontajner je prazdny, nemam kde vyhladavat.\n");
66              else
67              {
68                  printf("Aky prvok chces vyhladat v kontajneri?:\n");
69                  scanf("%d", &prvok);
70                  if ((index = sekvencneVyhladanie(p_pole, pozicia, prvok)) == -1)
71                      printf("Taky prvok sa v kontajneri nenachadza.\n");
72                  else
73                      printf("Index hladaneho prvku %d je: %d.\n",prvok, index);
74              }
75              break;
76          default:
77              odpoved = 1;
78          }
79      }
80      while (!odpoved);
81
82      // uvolnenie alokovanej oblasti
83      free(p_pole);
84      p_pole = NULL;
85
86      return 0;
87  }
88
89  // definicie funkcii
90  void vypisPola(int *p_pole, int pozicia)
91  {
92      int i;
93      printf("Prvky pola: ");
94      for (i = 0; i < pozicia; i++)
95          printf("%d ", *(p_pole + i));
96      printf("\n");
97  }
98
99  // ak sa prvok nenájde, tak vrati -1, inak vrati jeho index
100 int sekvencneVyhladanie(int *p_pole, int pozicia, int prvok)
101 {
102     int i;
103     for (i = 0; i < pozicia; i++)
104     {
105         if (p_pole[i] == prvok)
```

```
106             return i;
107     }
108     return -1;
109 }
110
111 // vlozenie prvku, ak sa presiahne velkost pola, tak sa tato automaticky zdvojnasobi
112 void vlozPrvok(int prvok, int *pozicia, int *p_pole, int *velkost)
113 {
114     if (*pozicia < *velkost)
115     {
116         p_pole[(*pozicia)++] = prvok;
117     }
118     else
119     {
120         if ((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
121         {
122             printf("Nedostatok pamate. Program sa konci.\n");
123             return;
124         }
125         else
126         {
127             (*velkost) *= 2;
128             p_pole[(*pozicia)++] = prvok;
129             printf("Prvok bol vlozeny po zvacseni kontajnera.\n");
130         }
131     }
132 }
133
134 // funkcia odstranuje prvy vyskyt zadaneho prvku, ak odstranenie nenastane, tak vrati 0
135 int odstranPrvok(int prvok, int *pozicia, int *p_pole)
136 {
137     int i, j;
138     for (i = 0; i < *pozicia; i++)
139     {
140         if (prvok == p_pole[i])
141         {
142             for (j = i; j < *pozicia; j++)
143             {
144                 p_pole[j] = p_pole[j + 1];
145             }
146             (*pozicia)--;
147             printf("Prvy vyskyt prvku %d bol odstraneny.\n", prvok);
148             return 1;
149         }
150     }
151     return 0;
152 }
```

Pozn. autora: Vzhľadom na to, že pracujeme s neusporiadaným poľom, nemôžeme využiť vyhľadávací algoritmus s nižšou časovou zložitosťou, ktorým je binárne vyhľadávanie. Jeho časová zložitosť je logaritmická *O(log n)*, na rozdiel od sekvenčného vyhľadávania, ktoré pracuje s lineárnou časovou zložitosťou *O(n)*.

4.6.2 Príklad – dynamické usporiadané pole

Zadanie problému: Vytvorte program, ktorý bude poskytovať možnosť usporiadaného ukladania celých čísiel do dynamického poľa. Tento bude slúžiť ako kontajner. To znamená, že jeho kapacita, ktorá bude na začiatku zadaná používateľom, sa meniť nebude. V prípade potreby vloženia prvku pri nedostatočnej kapacite sa táto automaticky zväčší (zdvojnásobí).

Pozn. autora: Voči príkladu v kapitole 4.6.1 nastala úprava funkcie na vkladanie prvku a zmenu funkcie na vyhľadanie prvku.

Jedna z možných implementácií 4.6.2_Pr_1.c:

```
#include <stdio.h>
#include <stdlib.h>

// deklaracie funkcii
void vypisPola(int *p_pole, int pozicia);
int binarneVyhladanie(int *p_pole, int sh, int hh, int kluc);
void vlozPrvokUsporiadane(int prvok, int *pozicia, int *p_pole, int *velkost);
int odstranPrvok(int prvok, int *pozicia, int *p_pole);
int main(void)
{
    // deklaracia pointera reprezentujuceho pole a dalsich premennych
    int pozicia = 0;
    int * p_pole, velkost, volba, odpoved = 0, prvok, i, index;

    // alokacia priestoru (kapacity pola) na zaklade volby pouzivatela
    do
    {
        printf("Ake velke ulozisko si zelas vytvorit?: \n");
        scanf("%d", &velkost);
    }
    while (velkost <= 0);

    if ((p_pole = (int *) malloc(sizeof(int) * velkost)) == NULL)
    {
        printf("Nedostatok pamate. Program sa konci.\n");
        return 1;
    }
    // menu
    do
    {
        printf("1 - vlozenie prvku do pola.\n");
        printf("2 - odstranenie prvku z pola.\n");
        printf("3 - vypisanie obsahu pola.\n");
        printf("4 - hladanie v poli.\n");
        printf("5 - koniec programu.\n");

        scanf("%d", &volba);
        switch(volba)
        {
        case 1:
            printf("Aky prvok chces vlozit do kontajnera?:\n");
```

```
42              scanf("%d", &prvok);
43              vlozPrvokUsporiadane(prvok, &pozicia, p_pole, &velkost);
44              break;
45          case 2:
46              if (pozicia == 0)
47                  printf("Tvoj kontajner je prazdny, nemas co odstranovat.\n");
48              else
49              {
50                  printf("Aky prvok chces odstranit z kontajnera?:\n");
51                  scanf("%d", &prvok);
52                  if (odstranPrvok(prvok, &pozicia, p_pole) == 0)
53                      printf("Taky prvok sa v kontajneri nenachadza.\n");
54              }
55              break;
56          case 3:
57              if (pozicia == 0)
58                  printf("Tvoj kontajner je prazdny, nemam co vypisat.\n");
59              else
60                  vypisPola(p_pole, pozicia);
61              break;
62          case 4:
63              if (pozicia == 0)
64                  printf("Tvoj kontajner je prazdny, nemam kde vyhladavat.\n");
65              else
66              {
67                  printf("Aky prvok chces vyhladat v kontajneri?:\n");
68                  scanf("%d", &prvok);
69                  if ((index = binarneVyhladanie(p_pole,0, velkost, prvok)) == -1)
70                      printf("Taky prvok sa v kontajneri nenachadza.\n");
71                  else
72                   printf("Index prveho vyskytu hladaneho prvku %d je: %d.\n",prvok, index);
73              }
74              break;
75          default:
76              odpoved = 1;
77          }
78      }
79      while (!odpoved);
80
81      // uvolnenie alokovanej oblasti
82      free(p_pole);
83      p_pole = NULL;
84
85      return 0;
86  }
87
88  // definicie funkcii
89  void vypisPola(int *p_pole, int pozicia)
90  {
91      int i;
92      printf("Prvky pola: ");
93      for (i = 0; i < pozicia; i++)
94          printf("%d ", *(p_pole + i));
95      printf("\n");
96  }
97
98  // ak sa prvok nenájde, tak vrati -1, inak vrati jeho index
99  int binarneVyhladanie(int *p_pole, int sh, int hh, int kluc)
```

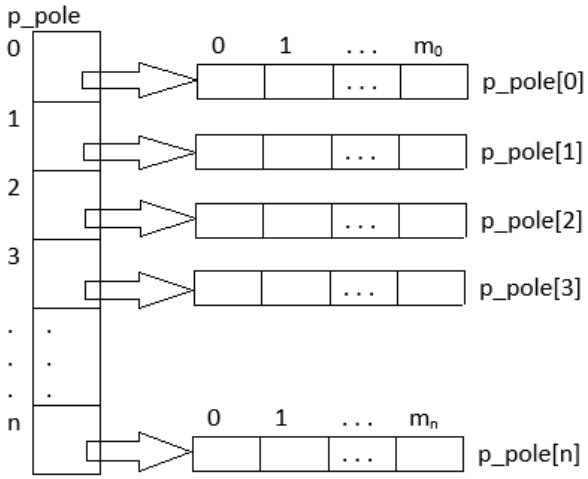
```
100 {
101     if (hh < sh)
102         return -1;
103     int stred = (sh + hh) / 2;  // int stred = sh + (hh - sh) / 2;
104     if (kluc == p_pole[stred])
105         return stred;
106     if (kluc > p_pole[stred])
107         return binarneVyhladanie(p_pole, (stred + 1), hh, kluc);
108     return binarneVyhladanie(p_pole, sh, (stred - 1), kluc);
109 }
110
111 // vlozenie prvku na spravne miesto, ak sa presiahne velkost pola, tak sa tato automaticky
zdvojnasobi
112 void vlozPrvokUsporiadane(int prvok, int *pozicia, int *p_pole, int *velkost)
113 {
114     int i, j;
115     if (*pozicia >= *velkost)
116     {
117         if ((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
118         {
119             printf("Nedostatok pamate. Program sa konci.\n");
120             return;
121         }
122         printf("Prvok bol vlozeny po zvacseni kontajnera.\n");
123     }
124
125     for (i = (*pozicia) - 1; (i >= 0 && p_pole[i] > prvok); i--)
126         p_pole[i + 1] = p_pole[i];
127     p_pole[i + 1] = prvok;
128     (*pozicia)++;
129 }
130
131 // funkcia odstranuje prvy vyskyt zadaneho prvku, ak odstranenie nenastane, tak vrati 0
132 int odstranPrvok(int prvok, int *pozicia, int *p_pole)
133 {
134     int i, j;
135     for (i = 0; i < *pozicia; i++)
136     {
137         if (prvok == p_pole[i])
138         {
139             for (j = i; j < *pozicia; j++)
140             {
141                 p_pole[j]=p_pole[j + 1];
142             }
143             (*pozicia)--;
144             printf("Prvy vyskyt prvku %d bol odstraneny.\n", prvok);
145             return 1;
146         }
147     }
148     return 0;
149 }
```

4.7 Dvojrozmerné dynamické pole

Uvažujme dynamické prideľovanie pamäte pre dvojrozmerné pole deklarované takto:

```
int **p_pole;
```

Grafická reprezentácia takéhoto poľa môže byť reprezentovaná takto (veľkosť vertikálneho poľa zodpovedá počtu riadkov, veľkosť horizontálnych polí počtu stĺpcov):



Obr. 63 Dvojrozmerné rovnomerné dynamické pole.

Algoritmus vytvorenia dvojrozmerného dynamického poľa je nasledujúci:

1. Deklarácia konštánt určujúcich počet riadkov a stĺpcov. Je samozrejme možné rozmery načítať priamo od používateľa, resp. zo súboru:

```
#define SIZE_X 4
#define SIZE_Y 5
```

2. Deklarácia dvojitého pointera predstavujúceho dvojrozmerné pole a deklarácia pomocných premenných:

```
int **p_pole, i = 0, j = 0;
```

3. Alokovanie (vyhradenie) pamäte pre pole smerníkov, toto zodpovedá počtu riadkov. Nezabúdajme na testovanie, či sa podarilo pamäť alokovať:

```
p_pole = (int **) malloc(sizeof(int *) * SIZE_X);
if (p_pole == NULL) exit(0);      // chyba
```

4. Alokovanie (vyhradenie) pamäte pre jednotlivé vektory poľa smerníkov, toto zodpovedá počtu stĺpcov:

```
for (i = 0; i < SIZE_X; i++)
{
    p_pole[i] = (int *) malloc(sizeof(int) * SIZE_Y);
    if (p_pole[i] == NULL) exit(0);
}
```

Algoritmus uvoľnenia dvojrozmerného dynamického poľa je nasledujúci:

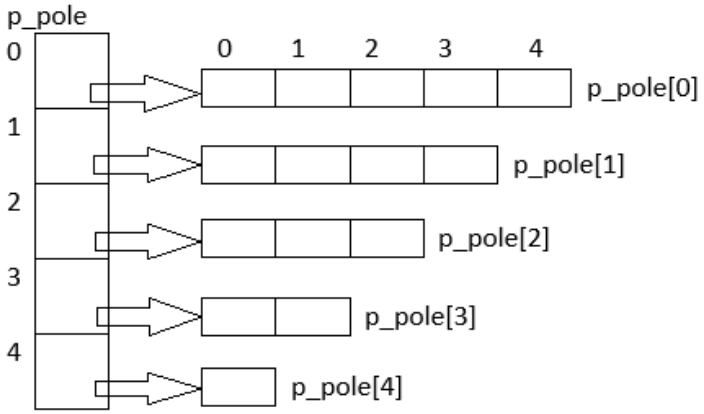
1. Uvoľnenie pamäte pre jednotlivé vektory poľa

```
for (i = 0; i < SIZE_X; i++)
{
    free(p_pole[i]);
}
```

2. Uvoľnenie pamäte pre pole smerníkov na vektory

```
free(p_pole);
p_pole = NULL;
```

Obdobným spôsobom je možné definovať aj dvojrozmerné pole s nerovnomernou štruktúrou, napríklad v tvare:



Obr. 64 Dvojrozmerné nerovnomerné dynamické pole.

4.7.1 Príklad – dynamické a statické dvojrozmerné pole

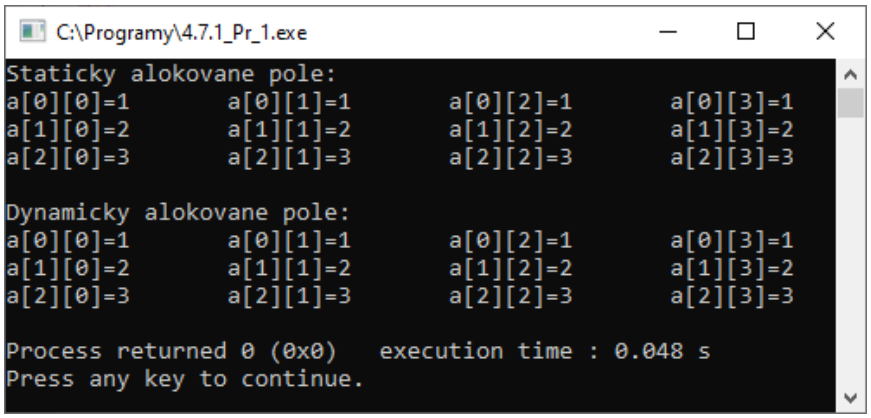
Zadanie problému: Vytvorte program, ktorý bude ilustrovať prácu s dynamickým aj so statickým dvojrozmerným poľom.

Jedna z možných implementácií 4.7.1_Pr_1.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE_X 3
5  #define SIZE_Y 4
6
7  // deklaracie funkcii
8  int** alokuj_pole(int m, int n);
9  void uvolni_pole(int** a, int m);
10 void vypis_pole(int** a, int m, int n);
11
12 int main(void)
13 {
14     // staticky alokovane 2D pole
15     printf("Staticky alokovane pole:\n");
16
17     int a[SIZE_X][SIZE_Y];
18
19     // naplnenie pola
```

```
20      for (int i = 0; i < SIZE_X; i++)
21          for (int j = 0; j < SIZE_Y; j++)
22              a[i][j] = i + 1;
23
24      // vypis pola
25      for (int i = 0; i < SIZE_X; i++)
26      {
27          for (int j = 0; j < SIZE_Y; j++)
28              printf("a[%d][%d] = %d\t", i, j, a[i][j]);
29          printf("\n");
30      }
31
32      // dynamicky alokovane 2D pole cez funkcie
33      printf("\nDynamicky alokovane pole:\n");
34
35      int** b = alokuj_pole(SIZE_X, SIZE_Y);
36
37      // naplnenie pola
38      for (int i = 0; i < SIZE_X; i++)
39          for (int j = 0; j < SIZE_Y; j++)
40              b[i][j] = i + 1;
41
42      vypis_pole(b, SIZE_X, SIZE_Y);
43
44      uvolni_pole(b, SIZE_X);
45
46      return 0;
47  }
48
49  // definicie funkcii
50  int** alokuj_pole(int m, int n)
51  {
52      int** b;
53
54      if ((b = (int**) malloc(m * sizeof(int*))) == NULL)
55      {
56          printf("Nedostatok pamate. Program sa konci.\n");
57          exit(1);
58      }
59
60      for (int i = 0; i < m; i++)
61          if ((b[i] = (int*) malloc(n * sizeof(int))) == NULL)
62          {
63              printf("Nedostatok pamate. Program sa konci.\n");
64              uvolni_pole(b, i);
65              exit(1);
66          }
67      return b;
68
69  }
70
71  void vypis_pole(int** a, int m, int n)
72  {
73      for (int i = 0; i < m; i++)
74      {
75          for (int j = 0; j < n; j++)
76              printf("a[%d][%d] = %d\t", i, j, a[i][j]);
77          printf("\n");
```

```
78      }
79  }
80
81  void uvolni_pole(int** a, int m)
82  {
83      for (int i = 0; i < m; i++)
84          free(a[i]);
85      free(a);
86      a = NULL;
87  }
```



Obr. 65 Konzolový výstup programu 4.7.1_Pr_1.c.

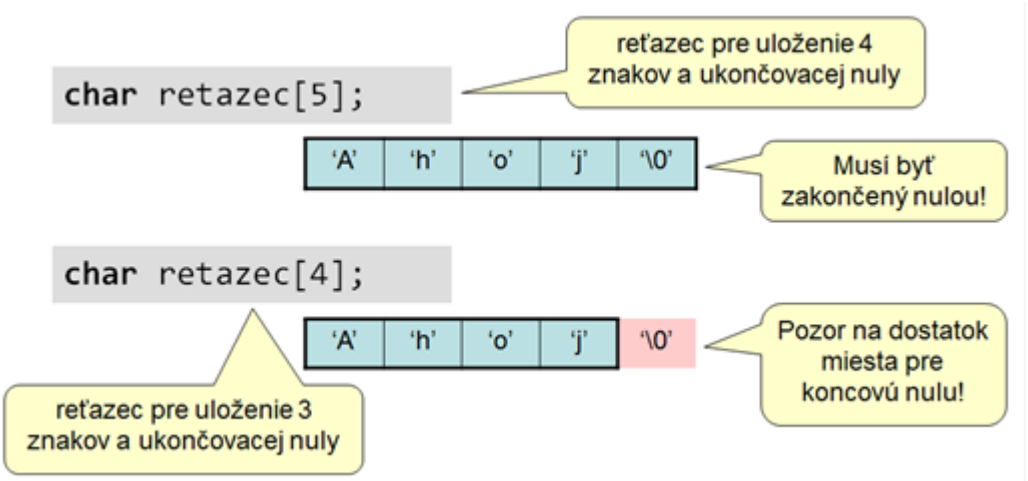
Pozn. autora: Funkcia *uvolni_pole()* zohľadňuje aj takú skutočnosť, že ak by nebolo dostatok priestorových nárokov počas priebežného alokovania pamäte pre jednotlivé riadky dvojrozmerného poľa, príde k uvoľneniu aj tohto čiastočne alokovaného priestoru – zabezpečené volaním funkcie na riadku č. 64.

4.8 Úlohy na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problém, ktorý do poľa načíta postupnosť čísiel a nájde maximálny (minimálny) prvok v poli. Umožnite veľkosť poľa zadať používateľovi, pole alokujte dynamicky a na nájdenie maximálneho a minimálneho prvku poľa použite:
   a) funkciu, ktorá pracuje s poľom s pomocou indexov,
   b) funkciu, ktorá využíva na prácu s poľom ukazovatele.
2. Algoritmicky riešte problém, ktorý do poľa načíta postupnosť čísiel a následne vypočíta priemer z prvkov poľa. Umožnite veľkosť poľa zadať používateľovi, pole alokujte dynamicky a na nájdenie priemeru použite:
   a) funkciu, ktorá pracuje s poľom s pomocou indexov,
   b) funkciu, ktorá využíva na prácu s poľom ukazovatele.

5 Reťazec a funkcie na prácu s reťazcom

Práca s reťazcami v jazyku *C* nie je taká jednoduchá a pohodlná, ako by sa mohlo očakávať. V jazyku *C* totižto neexistuje primitívny dátový typ pre reťazec. Reťazce sa načítavajú do poľa znakov, čiže špecifickým typom jednorozmerného poľa je reťazec – jednorozmerné pole typu *char* obsahujúce znaky ako prvky poľa tvoriace reťazec. Reťazec, ak má byť korektne vypísaný na obrazovku, alebo má korektne pracovať so vstavanými funkciami na prácu s reťazcami, je potrebné ukončiť znakom '\0', tzv. nulový znak, ukončovacia nula. Preto na uloženie reťazca musíme vždy alokovať pole o 1 B väčšie. Na použitie osobitých funkcií na prácu s reťazcami sa používa hlavičkový súbor *string.h*.



Obr. 66 Reprezentácia textového reťazca v jazyku *C*.

Textový reťazec ako konštanta, anglicky *literal* je text uzatvorený v úvodzovkách, napr. "Hello world.".

Príklady deklarácie a inicializácie:

```
char mesto[50];
// inicializacia pri deklaracii literalom, pocet prvkov pola je 6
char meno[] = "janko\0";
// inicializacia pri deklaracii inicializacnym zoznamom, pocet prvkov pola je 7
char meno2[] = { 'b', 'a', 't', 'm', 'a', 'n', '\0' };
```

Pozn.: Ak sa nezadá dĺžka reťazca, pri inicializácii sa určí automaticky (aj s miestom pre ukončovaciu nulu). V prípade inicializácie literálom nie je nevyhnutné zadávať ukončovaciu nulu, pretože prekladač ju doplní automaticky, t. j. `char meno[] = "janko";` je to isté ako `char meno[] = "janko\0";`

Pozn. autora: Dôležité je uvedomovať si rozdiel medzi jedným znakom, napr. 'a' ktorý sme schopný uložiť do premennej jednoduchého dátového typu. A reťazcom, napr. "a", predstavuje

jedno znakový reťazec, ktorý ale musí byť ukončený ukončovacou nulou a preto na jeho uloženie potrebujeme pole s veľkosťou 2 B.

Načítanie reťazca:

Načítanie reťazca v jazyku *C* spôsobuje viaceré problémy. Tieto sa pokúsim naznačiť pri jednotlivých možnostiach načítania reťazca, ktorých je viacero:

1. Načítanie po prvý biely znak s pomocou funkcie *scanf():*

   ```
   scanf("%s", s);
   ```

   preskočia sa všetky biele znaky (anglicky *whitespace characters*), načíta sa textový reťazec až po ďalší biely znak (najčastejšie medzera alebo enter) a uloží sa na adresu *s*.

   Pozn. 1: Použitie `scanf("%s", &s);` je chybné, *s* je už adresa do pamäte, ktorá musí byť už alokovaná v dostatočnej veľkosti (je to pole znakov).

   Pozn. 2: Použitie *scanf()* s formátom `scanf(" %10[^\n]s", s);` umožňuje načítať reťazec ktorý obsahuje aj medzery, načítanie je ukončené enterom, čiže načíta všetko okrem enteru. Okrem toho vďaka tomu, že sme vo formáte uviedli aj maximálnu dĺžku načítaného reťazca, nedôjde v prípade zadania dlhšieho reťazca než je alokované pole k jeho pretečeniu.

2. Ak chceme načítať celý riadok textu, pomôže nám štandardná funkcia *gets()* deklarovaná v hlavičkovom súbore *stdio.h*. Nevýhoda tejto funkcie spočíva v tom, že nemáme možnosť obmedzenia dĺžky načítavaného reťazca. Prototyp funkcie *gets()* je nasledujúci:

   ```
   char* gets(char *buffer);
   gets(s);
   ```

   Funkcia *gets()* číta znaky tak dlho, až sa nestlačí enter. Stlačenie enteru sa neuloží, ale nahradí sa nulovým znakom, ktorý ukončí reťazec.

   Funkcia *fgets()* rieši vyššie uvedený problém. Prototyp funkcie je:

   ```
   char *fgets(char *str, int n, FILE *stream);
   ```

   kde parameter *n* určuje, koľko znakov je možné maximálne prečítať. Čítanie sa končí buď načítaním $n - 1$ znakov, dosiahnutím konca riadka, alebo súboru, v prípade ak čítame zo súboru. Ak chceme čítať z klávesnice, ako parameter *stream* použijeme *stdin*.

3. Načítanie po znakoch s pomocou cyklu. Ak vyžadujeme zadať reťazec s presne definovanou dĺžkou, tak môžeme použiť cyklus *for*, v ktorom sú načítavané po jednotlivých znakoch:

   ```
   int i;
   char pole[11];
   ```

```c
for (i = 0; i < 10; i++)
{
    scanf(" %c", &pole[i]); // pole[i] = getchar(); // pole[i] = getc(stdin);
}
pole[i] = '\0';  // vlozenie ukoncovacej nuly
```

Častejšie sa však stretneme so situáciou, že vopred nevieme, aký dlhý reťazec používateľ zadá. Vtedy záleží už len na tom, ako vnímame reťazec. Ak ho chápeme ako slovo, tak ukončenie načítania môžeme podmieniť znaku medzery, ak ako vetu, či odsek, tak ukončenie podmienime znaku entera. V takomto prípade netreba zabudnúť aj na kontrolu veľkosti poľa, t. j. či stále máme kam zapisovať. Jedna z možných implementácii by mohla vyzerať:

```c
int i, MAX = 10;
char c, pole[MAX + 1];
// nacitanie slova
i = 0;
while (1)
{
        c = getchar();
        if (i >= MAX)          // indikuje nedostatok pamate
        {
            pole[i] = '\0';    // vlozenie uknocovacej nuly ak uz nemame miesto
            break;             // ukonci sa nacitavanie
        }
        // ulozenie ukoncovacej 0, ak je koniec slova
        if (c == ' ')
        {
            pole[i] = '\0';
            break;
        }
        pole[i++] = c;         // ulozenie znaku
}

// nacitanie vety, odseku
i = 0;
    while (1)
    {
        c = getchar();
       if (i >= MAX)           // indikuje nedostatok pamate,
       {
            pole[i] = '\0';    // vlozenie uknocovacej nuly ak uz nemame miesto
            break;             // ukonci sa nacitavanie
       }
        // ulozenie ukoncovacej 0, ak je koniec vety (odseku)
        if (c == '\n')
        {
            pole[i] = '\0';
            break;
        }
        pole[i++] = c; // ulozenie znaku
    }
```

Výpis reťazca:

Na výpis reťazca v jazyku *C* máme tiež viacero možností:

1. Výpis po koniec reťazca, t. j. po tzv. ukončovaciu nulu s pomocou funkcie *printf()*:

   ```
   printf("%s", s);
   ```

   preskočia sa všetky biele znaky (anglicky *whitespace characters*), vypíše sa textový reťazec až po ukončovaciu nulu uložený na adrese *s*.

   Pozn.: Použitie `printf("%s", *s);` je chybné, *s* musí predstavovať adresu pamäte, kde je reťazec uložený.

2. Ak chceme vypísať celý riadok textu, pomôže nám štandardná funkcia *puts()* deklarovaná v hlavičkovom súbore *stdio.h*. Prototyp funkcie *puts()* je nasledujúci:

   ```
   int puts(const char * str);
   puts(s);
   ```

   Funkcia *puts()* vytlačí celý riadok a ukončovaciu nulu nahrádza znakom nového riadka, t. j. odriadkuje. V prípade úspechu vracia nenulovú hodnotu, v prípade neúspechu vracia konštantu *EOF*.

3. Výpis po znakoch s pomocou cyklu. Je možné použiť cyklus *for*, pričom počet iterácii cyklu podmienime dĺžke reťazca, ktorú vieme zistiť s pomocou vstavanej funkcie *strlen()*, alebo kým nedosiahneme znak ukončovacej nuly:

   ```
   int i;
   char pole[11] = "retazec";
   for (i = 0; i < strlen(pole); i++)
   {
       printf("%c", pole[i]);      // putchar(pole[i]); // putc(pole[i], stdout);
   }
   ```

   alebo

   ```
   int i;
   char pole[11] = "retazec";
   for (i = 0; pole[i]; i++)
   {
       printf("%c", pole[i]);      // putchar(pole[i]); // putc(pole[i], stdout);
   }
   ```

   Pozn.: Podmienka v tvare *pole[i]*, čiže bez použitia relačného operátora sa na prvý pohľad môže zdať zvláštna. Avšak na základe skutočnosti, že každý reťazec musí byť ukončený ukončovacou nulou, čo je pre počítač obyčajná 0, a skutočnosti, že každý iný znak okrem 0 predstavuje pre jazyk *C* logickú hodnotu pravdy, je možné aj toto použitie. Kompletný zápis by vyzeral: `for (i = 0; pole[i] != '\0'; i++)`

   Samozrejme, že je možné použiť aj cyklus s podmienkou na začiatku:

   ```
   int i, MAX = 100;
   char c, pole[MAX + 1];
   ```

```
// vypis slova
i = 0;
while (pole[i]!= '  ')
{
    putchar(pole[i++]);
}
// vypis vety, odseku
i = 0;
while (pole[i] != '\0')
{
    putchar(pole[i++]);
}
```

Uloženie reťazca:

V prípade, ak máme deklarované pole pre reťazec, jeho uloženie nie je možné s pomocou priraďovacieho príkazu, ako by ste mohli očakávať:

```
pole = "text";   // tento prikaz sposobi chybu
```

V jazyku *C* nie je možné priradiť pole do poľa. Nič nám však nebráni priradiť jednotlivé znaky s pomocou cyklu, alebo použiť vstavanú funkciu na kopírovanie reťazca.

5.1 Základné funkcie na prácu s reťazcami

Základné funkcie na prácu s reťazcami sú deklarované v hlavičkovom súbore *string.h*. Medzi najčastejšie používané patria:

Funkcia na zistenie dĺžky reťazca. Prototyp:

```
int strlen(const char *);
```

Vráti dĺžku reťazca v bajtoch bez ukončovacej nuly.

```
strlen("Hello");        // vrati 5
```

Funkcia na kopírovanie reťazca *s2* do *s1* vrátane ukončovacej 0. Prototyp:

```
char* strcpy(char* s1, const char* s2);
```

Vráti pointer na prvý znak reťazca *s1*. Pri kopírovaní sa obsah zdrojového reťazca *s2* nijako nemení. Funkcia *strcpy()* nekontroluje, či cieľový reťazec disponuje dostatočným alokačným priestorom na absorpciu zdrojového reťazca.

```
strcpy(str, "Hello");   // skopiruje "Hello" do oblasti pamati, kam ukazuje str
```

Funkcia na spojenie (zreťazenie) reťazcov. Prototyp:

```
char *strcat(char *s1, const char *s2);
```

Pripojí reťazec *s2* za *s1*. Vráti pointer na prvý znak reťazca *s1*. Výsledný reťazec je ukončený nulovým znakom '\0'. Funkcia *strcat()* nekontroluje, či je cieľové znakové pole *s1* dostatočne veľké na to, aby dokázalo prijať výsledný textový reťazec.

```
strcat(str, " + Ahoj"); // pripoji "+ Ahoj" za str, v str bude "Hello + Ahoj"
```

Funkcia na nájdenie znaku v reťazci. Prototyp:

```
char *strchr(char *s1, char);
```

Hľadá zadaný znak v reťazci *s1*. Ak ho nájde, tak sa vráti pointer na jeho prvý výskyt, inak je vrátená hodnota *NULL*.

```
strchr(str, 'x');       // vrati NULL, pretoze x sa v retazci "Hello + Ahoj" nenachadza
```

Funkcia na porovnanie dvoch reťazcov. Prototyp:

```
int strcmp(const char *s1, const char *s2);
```

Lexikograficky porovná reťazce *s1* a *s2* a vráti 0, ak sú reťazce rovnaké, záporné číslo, ak je *s1* menšie ako *s2* a kladné číslo inak.

```
strcpy(s1, "abcdef");
strcpy(s2, "ABCDEF");
ret = strcmp(s1, s2);       // vrati zaporne cislo

strcpy(s1, "ABCDEF");
strcpy(s2, "abcdef");
ret = strcmp(s1, s2);       // vrati kladne cislo

strcpy(s1, "ABCDEF");
strcpy(s2, "ABCDEF");
ret = strcmp(s1, s2);       // vrati 0
```

Funkcia na nájdenie výskytu podreťazca v reťazci. Prototyp:

```
char *strstr(char *s1, char *s2);
```

Hľadá prvý výskyt podreťazca, uloženého v *s2* v reťazci *s1*. Ak ho nájde, tak sa vráti pointer na tento výskyt, inak je vrátená hodnota *NULL*.

Funkcie na prácu s **obmedzenou časťou reťazca** – v názve majú *n* (*number*). Majú možnosť zadať maximálny počet spracovávaných znakov. Napr.:

```
char * strncat(char *, char *, int);
int strncmp(char *, char *, int);
char * strncpy(char *, char *, int);
```

Funkcie na prácu **s reťazcom odzadu** – v názve majú *r* (*reverze*). Reťazec nespracovávajú od počiatočnej adresy reťazca, ale od adresy jeho koncového znaku ‘\0‘ smerom k začiatku reťazca. Napr. nájdenie znaku v reťazci odzadu:

```
char *strrchr(char *, char);
```

Ak sa znak v reťazci vyskytuje, vráti sa ukazovateľ na posledný výskyt tohto znaku, inak je vrátený *NULL*.

Často sa pri práci s reťazcami stretáme s funkciami na prevod reťazca na číslo. Deklarované sú v hlavičkovom súbore *stdlib.h*. Napr.:

```
int atoi(const char *string);       // konvertuje retazec na int
long atol(const char *string);      // konvertuje retazec na long
```

```
double atof(const char *string);     // konvertuje reťazec na double
```

Ďalšie často používané funkcie na prácu s reťazcami sú deklarované v hlavičkovom súbore *ctype.h*. Napríklad zistiť, či je daný znak abecedným znakom, číslom alebo interpunkčným znakom, je možné s pomocou funkcií:

```
void isalpha(int c);
int isdigit(int c);
int ispunct(int c);
```

V prípade úspechu vracajú nenulovú hodnotu, inak 0.

Veľkosť jednotlivých znakov v textovom reťazci je možné modifikovať prostredníctvom funkcií *toupper()* a *tolower()*. Funkcia *toupper()* prevádza malé písmeno na veľké. Prototyp:

```
int toupper(int c);
```

Funkcia *tolower()* prevádza veľké písmeno na malé. Prototyp:

```
int tolower(int c);
```

Skutočnosť, či je istý znak malým, resp. veľkým písmenom abecedy, testujú funkcie *isupper()* a *islower()*. Prototypy funkcií:

```
int isupper(int c);
int islower(int c);
```

Funkcie vracajú nenulovú hodnotu (*true*), ak znak *c* je malý/veľký, v opačnom prípade vracajú 0 (*false*).

5.1.1 Príklad – overenie správnosti zadaného hesla

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte riešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Zadanie problému: Riešte problém overenia zadaného hesla. Správne heslo je: *password*

Vstup (vstupné premenné):

heslo[MAX] – pole znakov na uloženie zadaného hesla

Výstup (výstupné premenné):

Výpis: „OK“ alebo „Toto nie je správne heslo.“

Vstupné podmienky:

heslo[i] $\in$ ASCII

Výstupné podmienky:

heslo[i] $\in$ ASCII $\wedge$ *heslo = "password"*

Pomocné premenné:

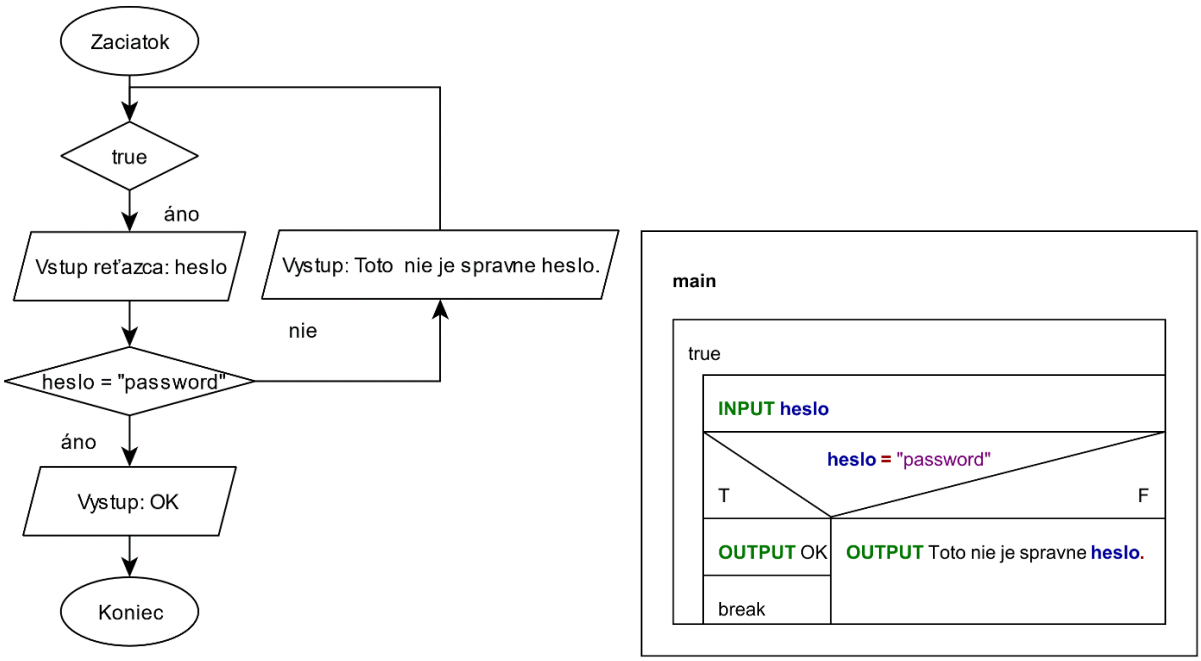
MAX – konštanta, určujúca maximálnu veľkosť poľa

Pozn.: Podmienka *heslo[i]* $\in$ ASCII, hovorí, že prípustnými znakmi sú všetky znaky *ASCII* tabuľky.

Rozbor úlohy: Úlohu je možné opätovne riešiť viacerými spôsobmi. Ja využívam nekonečný cyklus, ktorého hlavnou úlohu je overiť správnosť zadaného hesla. V prípade neúspechu je používateľ informovaný o neúspechu a dôjde znovu k vyžiadaniu zadania hesla, v opačnom prípade je informovaný o úspechu. Na overenie správnosti zadaného hesla z implementačného hľadiska je možné použiť vstavanú funkciu *strcmp(),* avšak zakreslenie algoritmu riešenia s použitím tejto funkcie nie je až tak vhodné. Asi len ťažko by sme presvedčili programátorov iných jazykov, že takto navrhnutý algoritmus spĺňa vlastnosť elementárnosti. Ako som už viackrát uviedla snažíme sa algoritmy reprezentovať tak, aby boli nezávislé od programovacieho jazyka. Treba však zdôrazniť, že každú vstavanú funkciu je vždy možné detailizovať a reprezentovať s pomocou elementárnych operácií, tak aby sa splnila vlastnosť elementárnosti.

Pozn. autora: Odporúčame pri návrhu algoritmu a jeho grafickej reprezentácie každú vstavanú funkciu reprezentovať s pomocou elementárnych operácií. Táto skutočnosť nijako neeliminuje programátora v prípade implementácie. Použitie vstavanej funkcie pri implementácii, ako vidíte v riešení nižšie, je viac ako očakávané. Zjednodušuje sa tým implementácia.

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



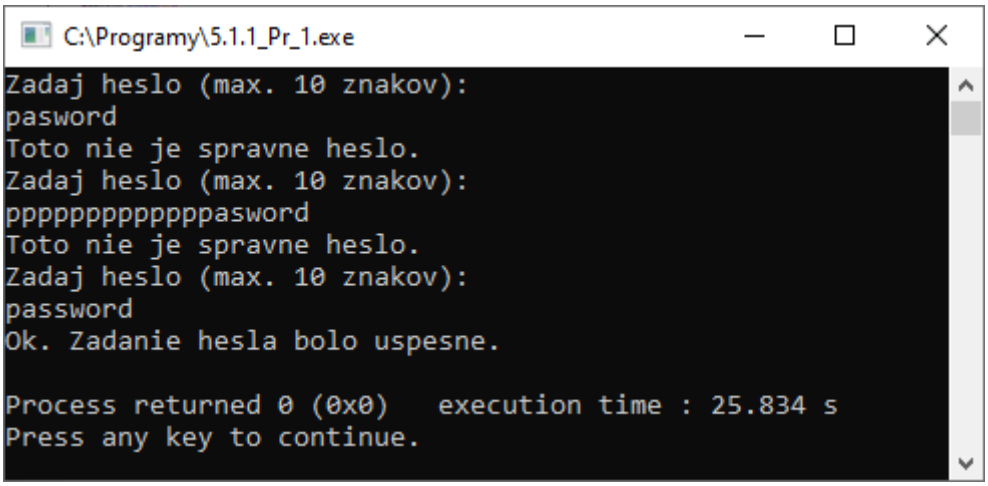
Obr. 67 Vývojový a NS diagram na overenie správnosti zadaného hesla.

Pozn. autora: Reprezentácia algoritmu na overenie správnosti zadaného hesla je aj v takejto minimalistickej podobe správna. Všimnite si, že pri kľúčovom slove *Vstup* vo vývojovom diagrame som doplnila aj ďalšie slovo *Vstup reťazca*, aby som zdôraznila, že nejde o načítanie obsahu premennej jednoduchého dátového typu. Vzhľadom na spôsob reprezentácie NS diagramu v softvéri, ktorý používam, toto žiaľ nie je možné.

Z uvedenej implementácie je zrejmé, že overiť správnosť zadaného hesla nie je možné v jazyku *C* takýmto jednoduchým porovnaním, avšak algoritmicky je reprezentácia problému zakreslená korektne.

Jedna z možných implementácií 5.1.1_Pr_1.c:

```c
#include <stdio.h>
#include <string.h>
#define MAX 10 // maximalna velkost pola

int main(void)
{
    char heslo[MAX];
    int i;

    while (1)
    {
        printf("Zadaj heslo (max. 10 znakov): \n");
        fflush(stdin);  // vyprazdnenie vstupneho buffera
        // nacita maximalne 10 znakov, aby nenastal zapis mimo alokovanej pamate
        scanf("%10s", heslo);
        /*
        scanf(" %10[^\n]s", heslo);
        printf("heslo = %s\n", heslo);
        */

        // otestovanie, ci je zadane heslo totozne s prototypom
        if (strcmp(heslo, "password") == 0)
            break;
        else
            printf("Toto nie je spravne heslo.\n");
    }

    printf("Ok. Zadanie hesla bolo uspesne.\n");

    return 0;
}
```

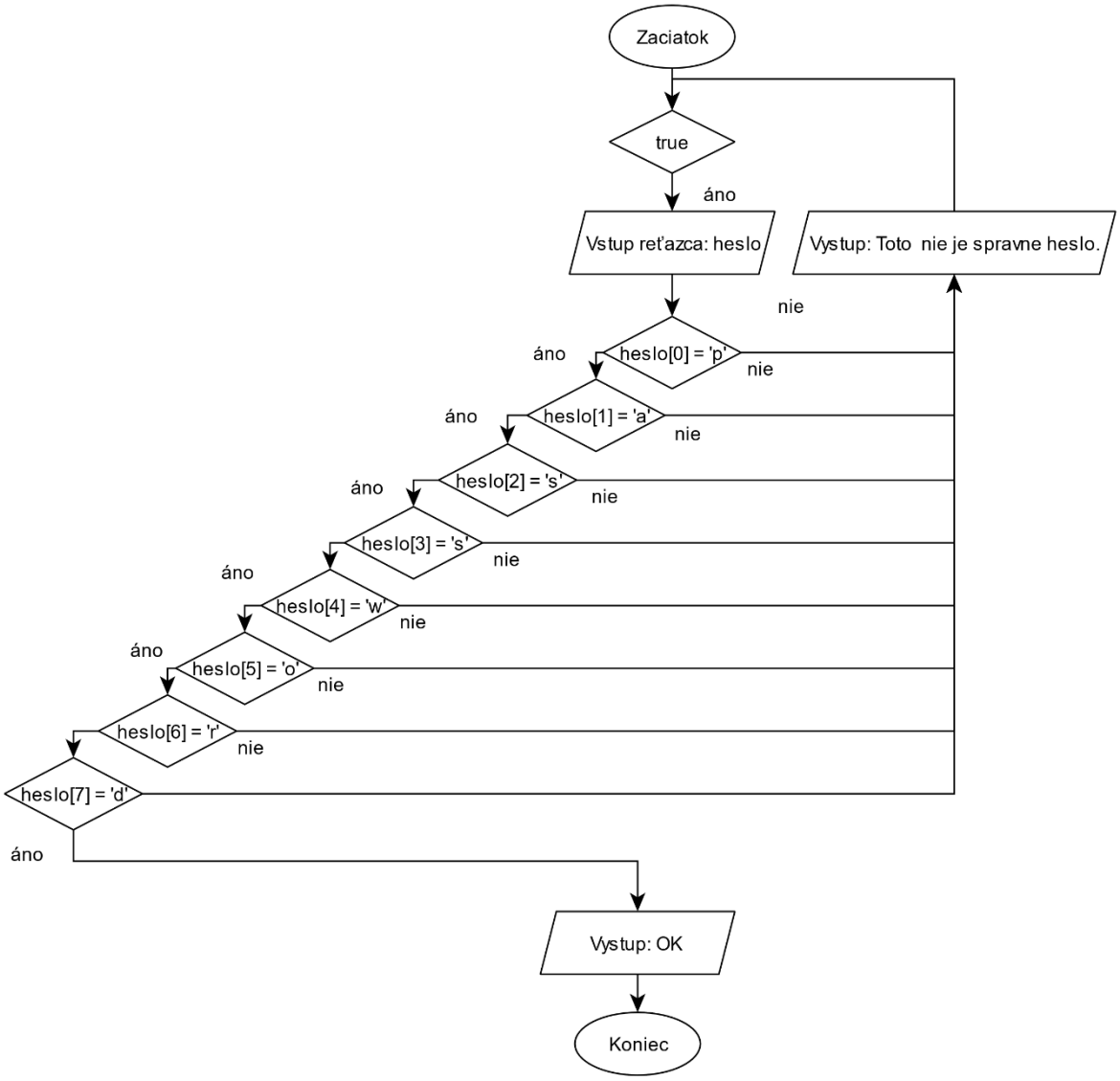


Obr. 68 Konzolový výstup programu 5.1.1_Pr_1.c.

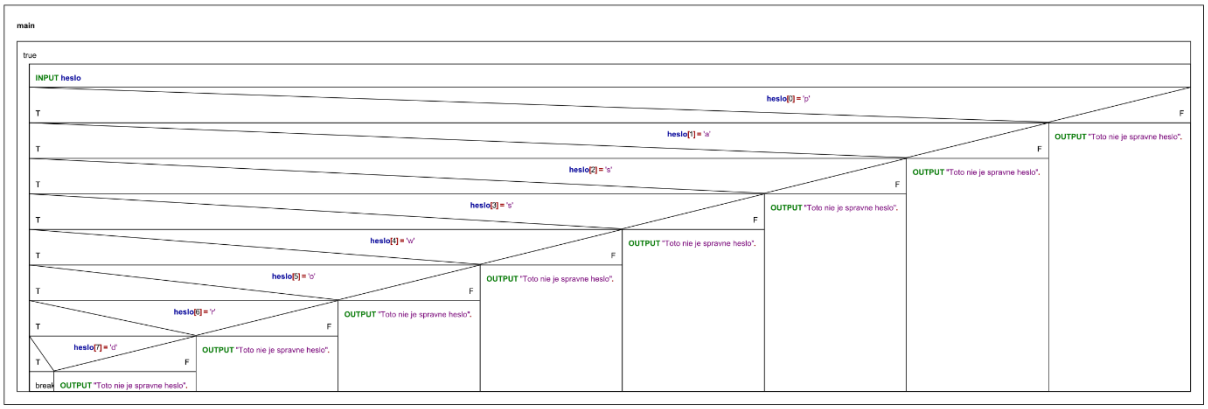
Pozn. autora: V prípade načítania reťazca som použila funkciu na štandardný vstup *scanf()*. V tomto prípade som si to mohla dovoliť z dôvodu, že heslom je slovo. Okrem toho načítavam (spracovávam) len prvých desať zadaných znakov. Toto nie je ošetrenie pre používateľa, aby

nemohol zadať reťazec dlhší ako desať znakov. Toto je ošetrenie zabezpečujúce to, aby sa údaje nezapisovali mimo alokovanej pamäte.

Z iného uhla pohľadu, by sme mohli na overenie správnosti zadaného hesla porovnať každý znak samostatne, t. j. ide o vyhodnotenie znak po znaku.



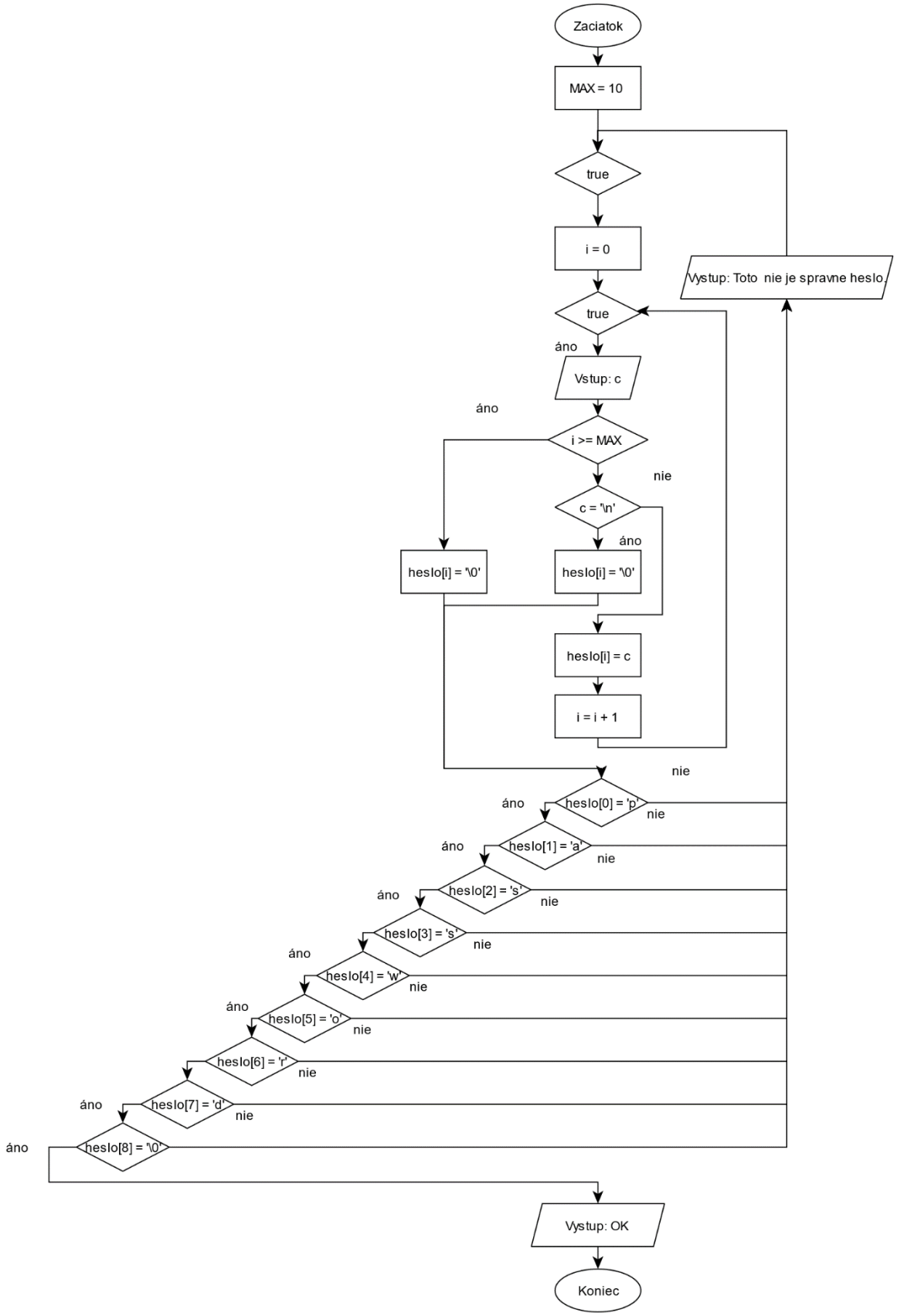
Obr. 69 Vývojový diagram na overenie správnosti zadaného hesla s testovaním znak po znaku.



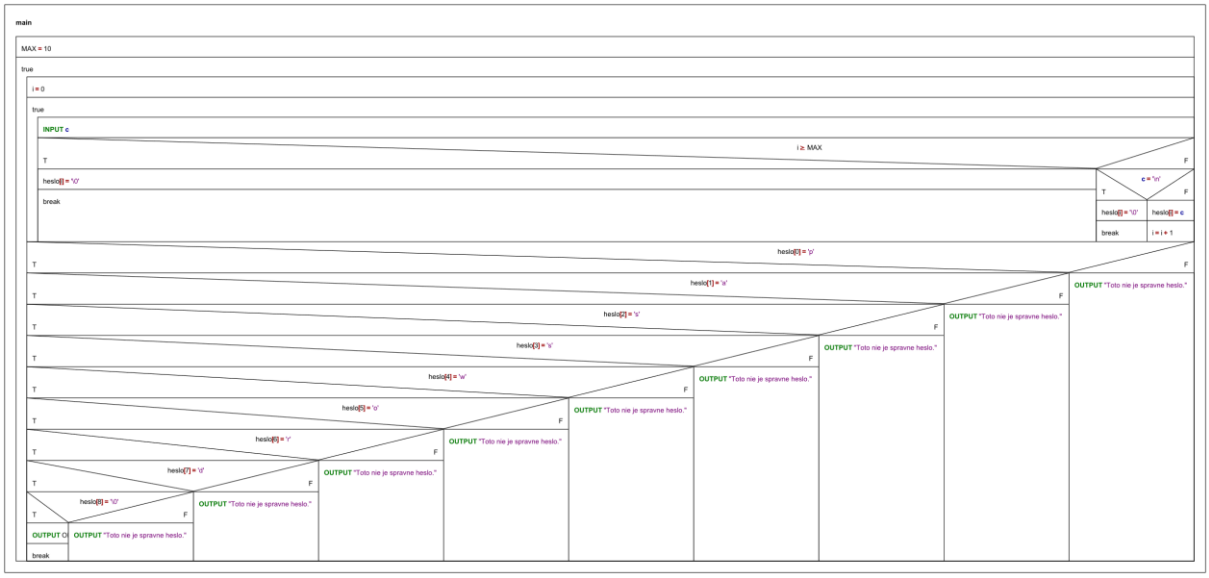
Obr. 70 NS diagram na overenie správnosti zadaného hesla s testovaním znak po znaku.

Pozn. autora: Príkazy *break* a *continue* skutočne reprezentujú funkčnosť skokových príkazov tak, ako by sme ich použili v jazyku *C*. Takéto použitie je prehľadnejšie, než uviesť, *„preruš telo cyklu a pokračuj testovaním podmienky cyklu“* pre *continue*, resp. *„preruš telo cyklu a pokračuj časťou za cyklom“* pre *break*.

V reprezentácii algoritmu nižšie som načítanie reťazca poňala tak, ako je to charakteristické pre jazyk *C*. Využila som nekonečný cyklus, pričom ukončenie načítania nastane buď pri zadaní enteru, alebo v prípade, ak presiahneme alokovanú dĺžku poľa. Z takejto reprezentácie je na prvý pohľad zrejmé, že je načítavaný reťazec.



Obr. 71 Vývojový diagram na overenie správnosti hesla doplnený o načítanie reťazca.

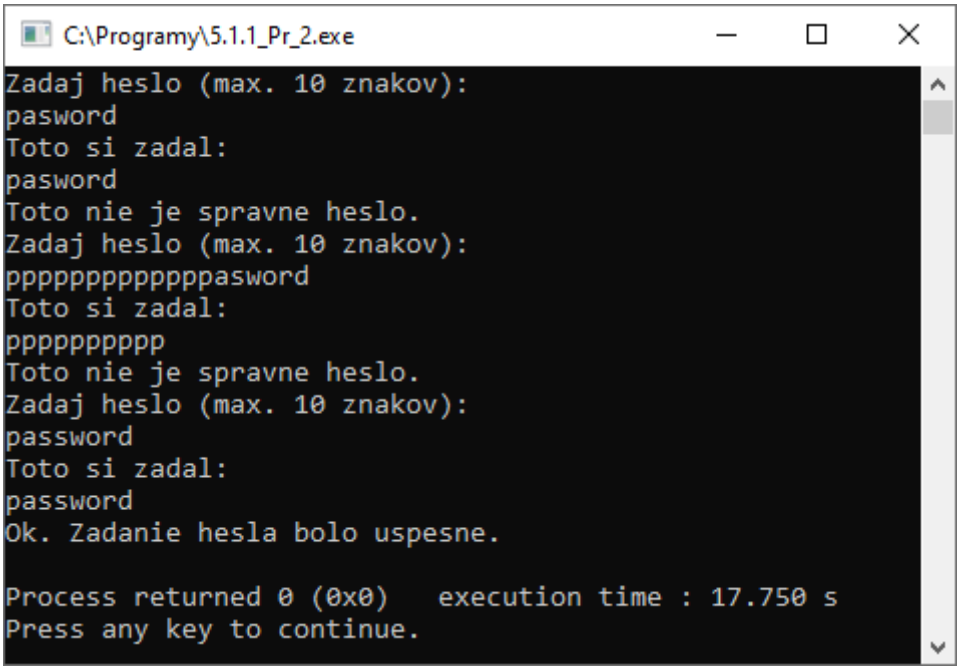


Obr. 72 NS diagram na overenie správnosti hesla doplnený o načítanie reťazca.

Iná implementácia s ošetrením toho, aby sa údaje nezapisovali mimo alokovanej pamäte v prípade, ak by používateľ zadal reťazec dlhší ako desať znakov [5.1.1_Pr_2.c](5.1.1_Pr_2.c):

```c
#include <stdio.h>
#include <string.h>
#define MAX 10 // maximalna velkost pola

int main(void)
{
    char c, heslo[MAX + 1];
    int i;

    while (1)
    {
        printf("Zadaj heslo (max. 10 znakov): \n");

        fflush(stdin);  // vycistenie vstupneho buffera
        // toto je osetrenie, aby sa nacitalo maximálne 10 znakov
        i = 0;
        while (1)
        {
            c = getchar();

            if (i >= MAX)
            {
                heslo[i] = '\0';
                break;  // indikuje nedostatok pamate, ukonci sa nacitavanie
            }

            // ulozenie ukoncovacej 0
            if (c == '\n')
            {
                heslo[i] = '\0';
                break;
            }

            heslo[i++] = c; // ulozenie znaku
```

```
35          }
36
37          printf("Toto si zadal:\n");
38          i = 0;
39          while (heslo[i] != '\0')
40          {
41              putchar(heslo[i++]);
42          }
43          printf("\n");
44
45          // otestovanie, ci je zadane heslo totozne s prototypom
46          if (strcmp(heslo, "password") == 0)
47              break;
48          else
49              printf("Toto nie je spravne heslo.\n");
50      }
51
52      printf("Ok. Zadanie hesla bolo uspesne.\n");
53
54      return 0;
55  }
```



Obr. 73 Konzolový výstup programu 5.1.1 _Pr_2.c.

Posúdenie vlastností algoritmu ponechávam na čitateľovi.

5.2 Úlohy na samostatné precvičovanie vedomostí

1. Algoritmicky riešte problém, ktorý umožní používateľovi zadať textový reťazec a algoritmus určí jeho dĺžku. Na výpočet dĺžky reťazca použite:
    a) vstavnú funkciu *strlen()*,
    b) vlastnú funkciu, ktorá k prvkom poľa bude pristupovať s pomocou indexov,
    c) vlastnú funkciu, ktorá s prvakmi reťazca bude pracovať s pomocou ukazovateľov.
2. Algoritmicky riešte problém, ktorý umožní používateľovi zadať dva textové reťazce *s1* a *s2* a algoritmus tieto reťazce spojí. Na spojenie reťazcov použite:
    a) vstavnú funkciu *strcat()*,
    b) vlastnú funkciu, ktorá k prvkom poľa bude pristupovať s pomocou indexov,
    c) vlastnú funkciu, ktorá s prvakmi reťazca bude pracovať s pomocou ukazovateľov.
3. Algoritmicky riešte problém, ktorý umožní používateľovi zadať textový reťazec a algoritmus vypíše tento reťazec od zadu. Na obrátenie reťazca použite funkciu.

6 Vstup zo súboru a výstup do súboru

V súčasnej informačnej dobe nie je efektívne a ani únosné zadávať väčšie množstvá údajov z klávesnice a preto sa často stretávame s inými úložiskami dát. Jedným z častých spôsobov ich organizácie a archivácie je uloženie do databáz. Tejto problematike sa však venovať v tejto učebnici nebudeme. V prípade našich jednoduchých programov, ktoré nevyžadujú veľké množstvo údajov, je vyhovujúcim úložiskom aj súbor ako vstup. Tiež práca s nim je jednoduchšia a nevyžaduje množstvo nových vedomostí pre začínajúcich programátorov, ktorým je táto učebnica určená. Obdobne, aj výstup programu na obrazovku často nie je dostačujúci a zápis výsledkov do súboru, prípadne do inej vhodnej formy, je žiaduci. V praxi sa často stretávame napr. s rôznymi formami reportov. Aj v tomto prípade siahnem po jednej z tých jednoduchších reprezentácií, ako je práca s textovým súborom.

Z pohľadu operačného systému každý súbor predstavuje postupnosť bajtov uložených na nejakom médiu (najčastejšie disku) v niekoľkých blokoch. Bloky majú rovnakú veľkosť a nemusia nevyhnutne ležať za sebou. To, ako sa s nimi pracuje, je teda čisto záležitosť operačného systému. Z hľadiska používateľského je súbor postupnosť po sebe idúcich bajtov od začiatku do konca súboru (Herout, 2010).

Vo všeobecnosti, čítanie a zápis do súboru (vstupno/výstupné I/O operácie) prebiehajú tak, že sa do pamäte (bufferu) načíta určitý blok súboru naraz a požadované znaky sa načítavajú už priamo z pamäte. Rovnako tak aj pri zápise. Najprv sa naplní buffer a až potom sa celý blok zapíše. Robí sa to preto, že sa minimalizuje prístup k médiu, čo zväčšuje jeho životnosť a čítanie a zápis sú rýchlejšie. Buffer je časť operačnej pamäte alokovaná systémom, kde sú dáta dočasne uložené pred zaslaním na miesto konečného určenia. V prípade čítania je to väčšinou dátová oblasť pamäte využívaná programom, v prípade zápisu ide najčastejšie o disk. Operačný systém pristupuje k periférnym zariadeniam teda len v prípade, ak potrebuje načítať, alebo zapísať celý obsah bufferu naraz. Buffer má najčastejšie veľkosť 512 B alebo 1 024 B. Neplatí to však vždy. Knižnica funkcií umožňuje používať aj nebuferované I/O operácie, kde každá I/O operácia je realizovaná okamžite.

Je nevyhnutné poznamenať, že vstupy z klávesnice a výstupy na obrazovku, čiže interaktívne I/O operácie, sa môžu považovať za špeciálny prípad súborových I/O operácií.

Na prácu so súbormi v jazyku *C* sa používa osobitný typ *FILE*, ktorý je súčasťou *stdio.h*. Rozoznávame dva druhy súborov:

- **Textové** – predstavujú štandardné súbory, ktorých obsah je tvorený textom – *ASCII* znakmi od 32 (medzera) po 127 (bez diakritiky). Tento typ súborov je pre nás jednoducho čitateľný, je možné ho vytvárať a upravovať s pomocou bežných textových editorov.
- **Binárne** – ich obsah nie je jednoducho čitateľný, keďže obsahujú iný typ informácií ako text. Binárne formáty sa používajú častejšie ako textové, keďže väčšinou predstavujú menšiu veľkosť a rýchlejšie spracovanie dát pri čítaní aj pri zápise. To je spôsobené najmä tým, že pri zápise čísla do textového súboru je nevyhnutné realizovať jeho konverziu z vnútornej reprezentácie čísla v počítači na textovú podobu (pri čítaní prebieha konverzia opačne, t. j. z textu na číslo). Pri binárnych súboroch tieto konverzie odpadajú, pretože sa do nich obsah zapisuje priamo po bajtoch. Sú vhodné najmä na ukladanie rozmerných dát – veľkých polí, štruktúr a pod. Pri ukladaní znakov nemajú príliš veľký význam, pretože znak v textovom aj v binárnom súbore, zaberá jeden bajt. Treba si však byť vedomý ich nevýhod: okrem komplikovaného čítania, aj nemožnosť ich vytvárania, prípadne opravovania s pomocou bežne používaných editorov.

Práca so súbormi prináša viaceré úskalia, ktoré sú spojené najmä s ich prenositeľnosťou, čo súvisí s ich vnútornou reprezentáciou pod rôznymi operačnými systémami. Toto však táto učebnica nerieši. Na získanie bližších informácií odporúčam prečítať napr. knihu od Pavla Herouta Učebnice jazyka C, 1. díl a 2 díl.

Ja sa na súbor pozriem ako na vstup, resp. výstup v prípade algoritmického riešenia problému. Pri algoritmickom riešení problému nie je ani tak významné, odkiaľ dáta vstupujú, prípadne v akej podobe vystupujú, čo naznačuje aj nemožnosť zakreslenia tejto skutočnosti s pomocou NS diagramu. V prípade použitia vývojových diagramov je možné súbor ako vstup/výstup predsa len reprezentovať. Daná reprezentácia však nijak detailne nepopisuje, ako treba túto problematiku riešiť implementačne. Napríklad v jazyku *C* je práca so súborom rozdelená do troch krokov: otvorenie súboru, práca so súborom a uzatvorenie súboru, s čím je spojená zručnosť a poznanie ďalších vstavaných funkcií. Tieto sa v závislosti od použitého programovacieho jazyka značne líšia. To sú hlavné dôvody, prečo sa detailne nevenujem práci so súborom z implementačného hľadiska.

6.1 Príklad – súčet dvoch matíc, prvky matíc a aj ich rozmery sú načítavané zo súboru, výstup je realizovaný na obrazovku aj do súboru

Algoritmicky riešte (definícia vstupov a výstupov spolu s určením vstupno-výstupných podmienok) a zapíšte riešenie s pomocou vývojového diagramu (VD) a NS diagramu. Posúďte vami navrhnutý algoritmus na základe jeho vlastností, uveďte ktoré vlastnosti sú nutné, odporúčané, resp. očakávané.

Pozn. autora: Všímajte si rozdiely medzi týmto riešením a riešením príkladu v kapitole 1.2.1.

Zadanie problému: Riešte problém súčtu dvoch matíc.

Pozn. autora: Často sa študenti pri takto formulovanom zadaní pýtajú, či majú byť rozmery matíc, prípadne ich prvky načítavané z klávesnice, alebo zo súboru. Odpoveď býva rôzna. Algoritmus súčtu matíc sa totižto nijak nezmení. Zmena nastane len v spôsobe získania dát, resp. ich reprezentácie. Ak teda nie je učiteľovým cieľom odskúšať znalosť študentov z funkcionality spojenej so súbormi v konkrétnom programovacom jazyku, tak je to na nich. V mnohých prípadoch, ak sa študenti rozhodnú prácu so súbormi ilustrovať algoritmicky, vedie to k chybným výsledkom. Diagramy sú plné volaní vstavaných funkcií, čím sa algoritmus stáva implementačne závislým a dokonca pri neznalosti daných funkcií čitateľom aj neelementárnym. Vlastnosť elementárnosti je porušená aj v prípade ovládania daných funkcií a ich správnej reprezentácii v diagrame, pretože využitie vstavaných funkcií neodzrkadľuje všetky elementárne kroky, ktoré sú za daným volaním funkcie skryté. Napríklad funkcia deklarovaná v *stdio.h* s prototypom *char *fgets(char *str, int n, FILE *stream)* umožňuje načítať celý riadok súboru. Z jej volania nie je možné vyčítať k načítaniu koľkých znakov reálne dôjde. Prototyp funkcie totižto hovorí, že *str* je smerník na znakové pole, do ktorého bude uložený načítaný riadok textu, *n* je maximálny počet znakov, ktoré budú načítané, *stream* je smerník na inštanciu štruktúry *FILE*. Funkcia *fgets()* číta znaky zo súboru až do konca riadka '\n', maximálne však *n*−1 znakov (vrátane znaku nového riadka). K načítanej postupnosti znakov funkcia *fgets()* pridáva ukončovací znak '\0'. Ak je načítanie textového reťazca úspešné, vracia funkcia *fgets()* smerník na pole textových znakov (*char* str*), alebo *NULL*, ak je dosiahnutý koniec súboru.

V prípade ak chceme čítať riadok po znakoch, musíme sa o to postarať sami. Vtedy však môžeme naraziť na problém, ako je vlastne riadok ukončený. V literatúre sa koniec riadka označuje ako *End-of-Line* (*EOLN*). Avšak konštanta *EOLN* nie je nikde definovaná, na rozdiel

od konštanty konca súboru *End-of-File* (*EOF)*. Ako štandardný znak pre koniec riadka jazyk *C* používa znak '\n', ktorý sa dá použiť vo väčšine prípadov pri práci s textovým súborom.

Vstup (vstupné premenné):

rozmery_matic.txt – textový súbor s dvomi kladnými celočíselnými hodnotami oddelenými buď medzerou alebo enterom, ktoré predstavujú rozmery vstupných matíc *A* a *B*, súbor sa nachádza v priečinku kde je spustiteľný kód programu

m, n – rozmery matice *A* a matice *B*, počet riadkov a stĺpcov, hodnoty sú buď prečítané zo súboru *rozmery_matic.txt*, resp. v prípade neúspechu priamo z klávesnice od používateľa

matica1.txt – textový súbor v ktorom sú prvky (reálne čísla) vstupnej matice *A*, počet prvkov zodpovedá hodnote súčinu *m* a *n*, jednotlivé prvky sú oddelené buď medzerou alebo enterom, súbor sa nachádza v priečinku, kde je spustiteľný kód programu

matica2.txt – textový súbor v ktorom sú prvky (reálne čísla) vstupnej matice *B*, počet prvkov zodpovedá hodnote súčinu *m* a *n*, jednotlivé prvky sú oddelené buď medzerou, alebo enterom, súbor sa nachádza v priečinku, kde je spustiteľný kód programu

A[m][n] – vstupná matica *A*, prvky matice sú načítané zo súboru *matica1.txt*, resp. v prípade neúspechu priamo z klávesnice od používateľa

B[m][n] – vstupná matica *B*, prvky matice sú načítané zo súboru *matica2.txt*, resp. v prípade neúspechu priamo z klávesnice od používateľa

Výstup (výstupné premenné):

sucet_matic.txt – textový súbor v ktorom sú výsledky súčtu matíc *A* a *B*, t. j. prvky matice *C*, súbor sa nachádza v priečinku kde je spustiteľný kód programu

C[m][n] – výstupná matica *C*, daná súčtom matice *A* a matice *B*

Vstupné podmienky:

$m, n \in \mathrm{Z}^{+} \wedge \in$ *rozmery_matic.txt,* $A[i][j] \in \mathrm{R} \wedge \in$ *matica1.txt,* $B[i][j] \in \mathrm{R} \wedge \in$ *matica2.txt, rozmery_matic.txt* $\wedge$ *matica1.txt* $\wedge$ *matica2.txt* sa nachádzajú v priečinku, kde je spustiteľný kód programu, jednotlivé hodnoty v nich uvedené sú oddelené medzerou alebo enterom

Výstupné podmienky:

$C[i][j] \in \mathrm{R} \wedge C[i][j] = A[i][j] + B[i][j] \wedge \in$ *sucet_matic.txt*

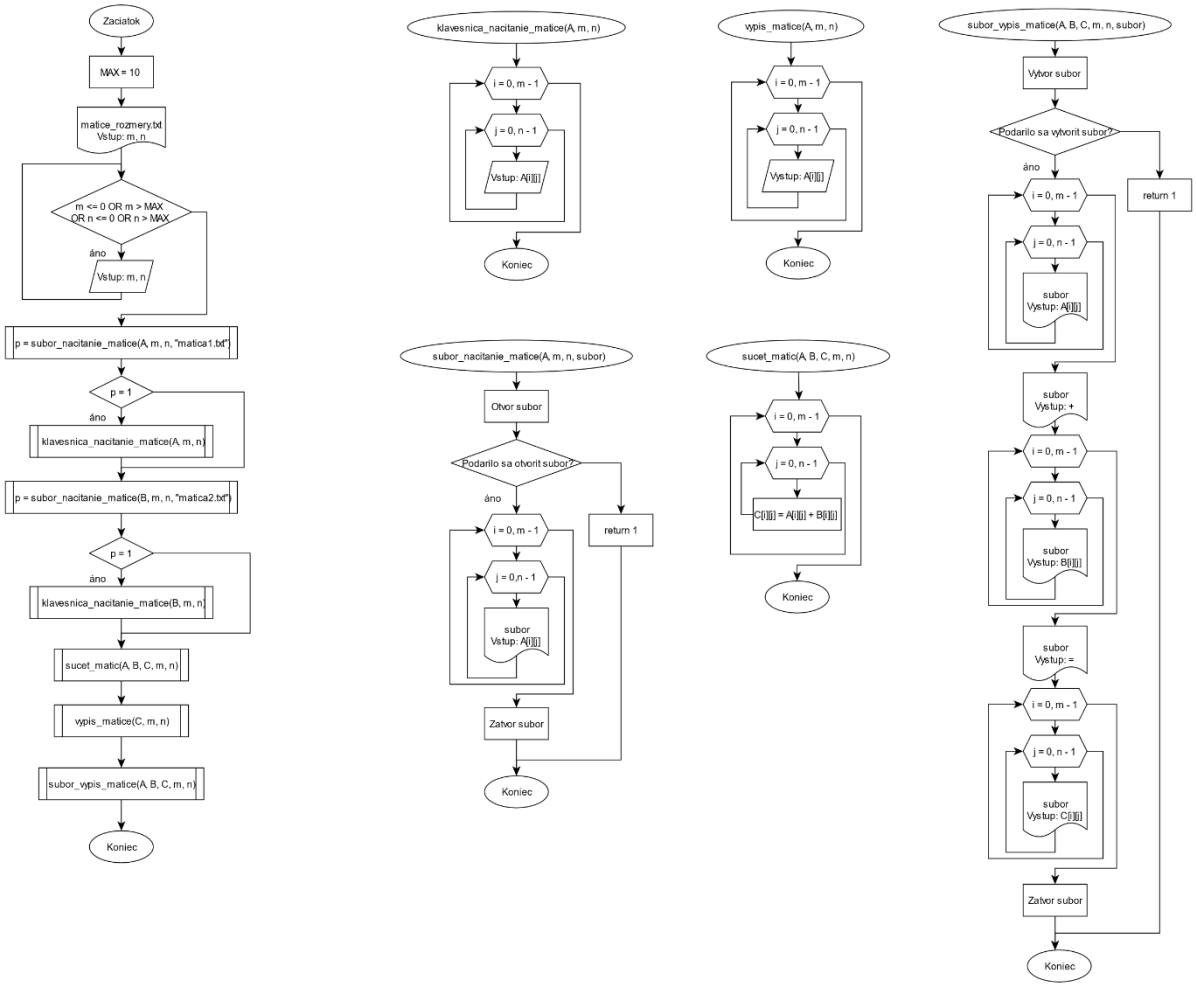
Pomocné premenné:

i – riadiaca premenná, $i \in \langle 0, m)$

j – riadiaca premenná, $j \in \langle 0, n)$

MAX – konštanta na určenie maximálnej veľkosti matice (na úplnosť uvádzame aj túto premennú, je však zrejmé, že jej použitiu sa vieme vyhnúť použitím dynamického poľa)

Algoritmus riešenia – grafická reprezentácia s pomocou vývojového a NS diagramu:



Obr. 74 Vývojový diagram na súčet matíc s pomocou funkcií načítanie zo súboru aj zápis do súboru.

Obr. 75 NS diagram na súčet matíc s pomocou funkcií načítanie zo súboru aj zápis do súboru.

Pozn. autora: Ako možno vidieť z uvedeného NS diagramu, na vstup/výstup zo súboru nemáme k dispozícii žiadnu inú značku, ako na klasický vstup/výstup z klávesnice. V takomto prípade odporúčam uviesť názov súboru s ktorým sa pracuje (či už priamo jeho názov, alebo premennú ak sme vo funkcii – ilustrujeme oba prípady), s relatívnou resp. absolútnou cestu (v závislosti od stanovených podmienok). V tomto prípade je využitá relatívna cesta. Kombinácií práce so súbormi je viacero, napr. v tomto prípade používame na načítanie/výpis *for* cyklus, keďže sme zadefinovali podmienky tak, že v súbore musí byť taký počet prvkov, ktorý zodpovedá rozmerom matíc, s ktorými sa pracuje. Ak by sme zvažovali situáciu, že tam môže byť prvkov menej, resp. viac, museli by sme algoritmus riešenia reprezentovať inak. Tieto varianty práce so súborom ponechávame plne na čitateľovi.

Overenie správnosti algoritmu:

Ako som naznačila už v úvode, v algoritme súčtu matíc nenastáva žiadna zmena, preto nepovažujem za dôležité z tohto hľadiska vykonávať overenie správnosti algoritmu. Tú som už v tejto učebnici robila. Chcem však upozorniť na skutočnosť, že som sa snažila zvýšiť robustnosť riešenia ošetrením v prípadoch zlyhania práce so súbormi. Táto sa deje pomerne často z viacerých dôvodov. Už len skutočnosť, že používateľ zabudne vytvoriť dané súbory, by viedla k zlyhaniu programu. Preto je program (algoritmus) navrhnutý tak, aby v prípade, ak zlyhá práca so súbormi, boli údaje vyžiadané priamo od používateľa a program vždy viedol k výsledku. Ďalším mojim cieľom bolo poukázať na jednom diagrame na rozdiel vstupu/výstupu z klávesnice voči súboru.

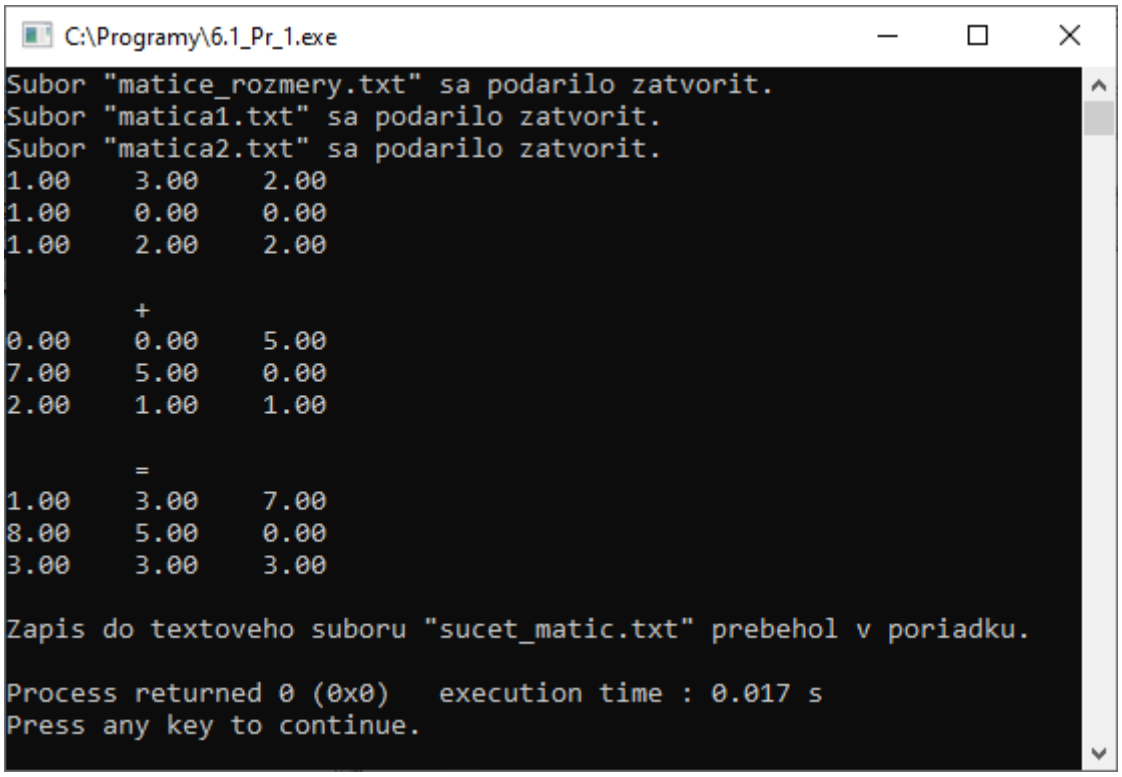
Jedna z možných implementácií 6.1_Pr_1.c:

```c
#include<stdio.h>

const int MAX = 10;

// deklaracie funkcii
int subor_nacitanie_matice(float A[][MAX], int m, int n, char *subor);
void klavesnica_nacitanie_matice(float A[][MAX], int m, int n);
void sucet_matic(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n);
void vypis_matice(float A[][MAX], int m, int n);
int subor_vypis_matice(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n);

int main(void)
{
    float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    int m, n;
    FILE *ft;

    // nacitanie rozmerov matic zo suboru, v pripade neuspechu z klavesnice
    ft = fopen("matice_rozmery.txt", "r");
```

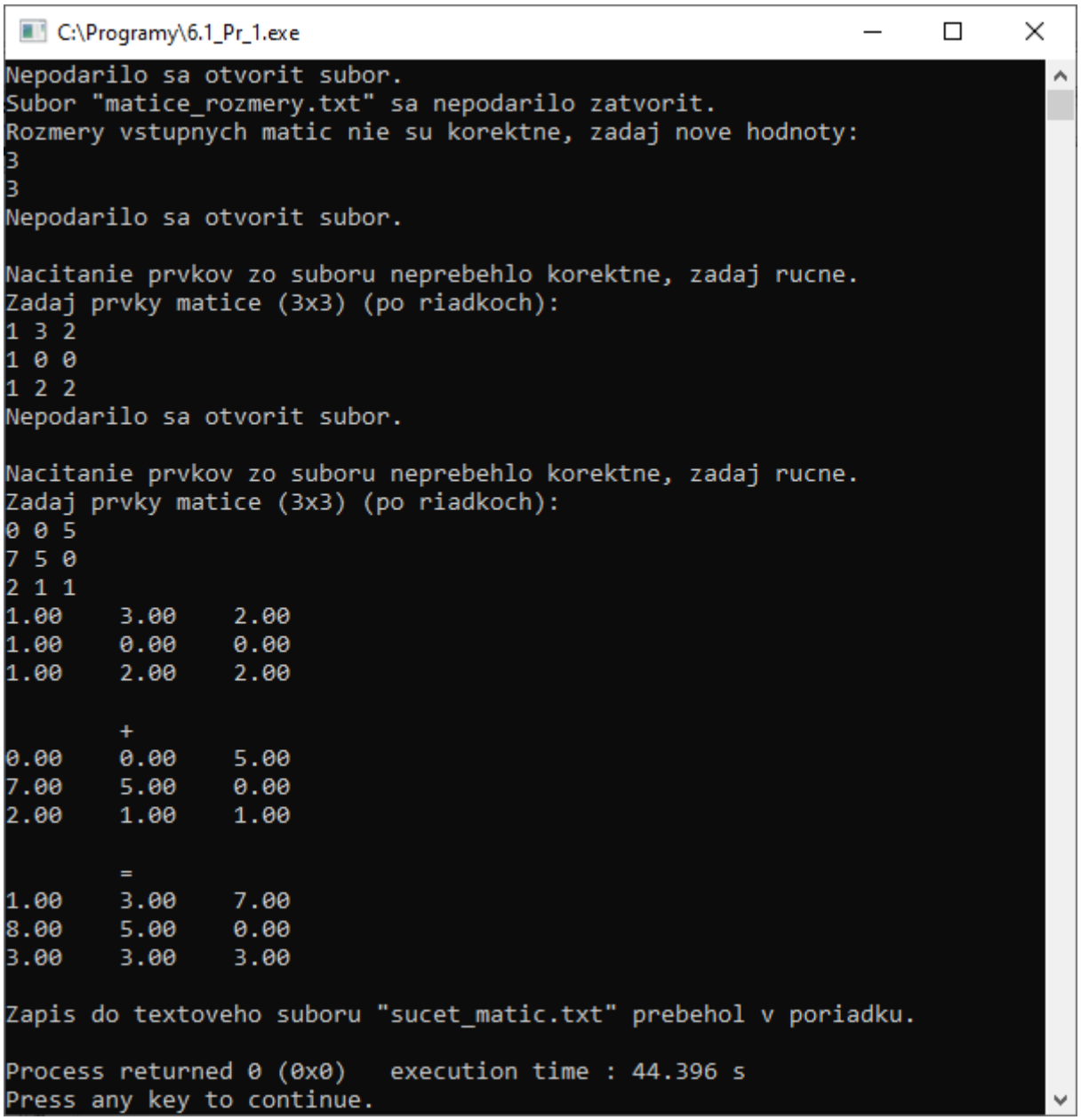
```
20      if (!ft)
21      {
22          printf("Nepodarilo sa otvorit subor.\n");
23      }
24      fscanf(ft, "%d %d", &m, &n);
25      if (fclose(ft) == 0)
26          printf("Subor \"matice_rozmery.txt\" sa podarilo zatvorit.\n");
27      else
28          printf("Subor \"matice_rozmery.txt\" sa nepodarilo zatvorit.\n");
29
30      while (m <= 0 || m > MAX || n <= 0 || n > MAX)
31      {
32          printf("Rozmery vstupnych matic nie su korektne, zadaj nove hodnoty: \n");
33          scanf("%d %d", &m, &n);
34      };
35
36      // nacitanie prvkov matic zo suboru, v pripade neuspechu z klavesnice
37      if (subor_nacitanie_matice(A, m, n, "matica1.txt") == 1)
38      {
39          printf("\nNacitanie prvkov zo suboru neprebehlo korektne, zadaj ich rucne.\n");
40          klavesnica_nacitanie_matice(A, m, n);
41      }
42
43      if (subor_nacitanie_matice(B, m, n, "matica2.txt") == 1)
44      {
45          printf("\nNacitanie prvkov zo suboru neprebehlo korektne, zadaj ich rucne.\n");
46          klavesnica_nacitanie_matice(B, m, n);
47      }
48
49      // scitanie matic
50      sucet_matic(A, B, C, m, n);
51
52      // vypis vysledku na obrazovku
53      vypis_matice(A, m, n);
54      printf("\n\t+\t\n");
55      vypis_matice(B, m, n);
56      printf("\n\t=\t\n");
57      vypis_matice(C, m, n);
58
59      // vypis vysledku do suboru
60      if (subor_vypis_matice(A, B, C, m, n) == 1)
61          printf("\nZapis do textoveho suboru neprebehol.\n");
62
63      return 0;
64  }
65
66  // definicie funkcii
67  // funkcia na nacitanie prvkov matice zo suboru, v pripade neuspechu vrati 1
68  int subor_nacitanie_matice(float A[][MAX], int m, int n, char *subor)
69  {
70      FILE *ft;
71      int i, j, c;
72
73      ft = fopen(subor, "r");
74      if (!ft)
75      {
76          printf("Nepodarilo sa otvorit subor.\n");
77          return 1;
```

```
78      }
79
80      for (i = 0; i < m; i++)
81      {
82          for (j = 0; j < n; j++)
83          {
84              fscanf(ft, "%f", &A[i][j]);
85          }
86      }
87
88      if (fclose(ft) == 0)
89          printf("Subor \"%s\" sa podarilo zatvorit.\n", subor);
90      else
91          printf("Subor \"%s\" sa nepodarilo zatvorit.\n", subor);
92  }
93
94  // funkcia na nacitanie prvkov matice z klavesnice
95  void klavesnica_nacitanie_matice(float A[][MAX], int m, int n)
96  {
97      int i, j;
98      printf("Zadaj prvky matice (%d x %d) po riadkoch: \n", m, n);
99      for (i = 0; i < m; i++)
100     {
101         for (j = 0; j < n; j++)
102         {
103             scanf("%f", &A[i][j]);
104         }
105     }
106 }
107
108 // funkcia na sucet matic
109 void sucet_matic(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n)
110 {
111     int i, j;
112     for (i = 0; i < m; i++)
113     {
114         for (j = 0; j < n; j++)
115         {
116             C[i][j] = A[i][j] + B[i][j];
117         }
118     }
119 }
120
121 // funkcia na vypis prvkov matice na obrazovku
122 void vypis_matice(float A[][MAX], int m, int n)
123 {
124     int i, j;
125     for (i = 0; i < m; i++)
126     {
127         for (j = 0; j < n; j++)
128         {
129             printf("%.2f\t", A[i][j]);
130         }
131         printf("\n");
132     }
133 }
134
135 // funkcia na vypis suctu matic do suboru, v pripade neuspechu vrati 1
```

```
136 int subor_vypis_matice(float A[][MAX], float B[][MAX], float C[][MAX], int m, int n)
137 {
138     FILE *ft;
139     int i, j;
140     ft = fopen("sucet_matic.txt", "w");
141 
142     if (!ft)
143     {
144         printf("Nepodarilo sa vytvorit subor.\n");
145         return 1;
146     }
147 
148     for (i = 0; i < m; i++)
149     {
150         for (j = 0; j < n; j++)
151         {
152             fprintf(ft, "%.2f\t", A[i][j]);
153         }
154         fprintf(ft, "\n");
155     }
156     fprintf(ft,"\n\t+\t\n");
157     for (i = 0; i < m; i++)
158     {
159         for (j = 0; j < n; j++)
160         {
161             fprintf(ft, "%.2f\t", C[i][j]);
162         }
163         fprintf(ft, "\n");
164     }
165     fprintf(ft, "\n\t=\t\n");
166     for (i = 0; i < m; i++)
167     {
168         for (j = 0; j < n; j++)
169         {
170             fprintf(ft, "%.2f\t", B[i][j]);
171         }
172         fprintf(ft, "\n");
173     }
174 
175     printf("\nZapis do textoveho suboru \"sucet_matic.txt\" prebehol v poriadku.\n");
176 }
```



Obr. 76 Konzolový výstup programu 6.1_Pr_1.c.



Obr. 77 Konzolový výstup programu 6.1_Pr_1.c – ak zlyhá práca so súbromi.

Pozn. autora: Zhodnotenie vlastností algoritmu ponechávame tiež na čitateľovi, v podstate k výrazným zmenám voči príkladu v kapitole 1.2.1 nedochádza.

7 Zoznam použitej literatúry a informačných zdrojov

BELAN, Anino, 2011. Kurz Jazyk C. Učebný text pre kvartu a kvintu osemročného gymnázia. 2. vydanie. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://docplayer.gr/24355101-Kurz--jazyka-c-ucebny-text-pre-kvartu-a-kvintu-osemrocneho-gymnazia.html⟩.

GEEKSFORGEEKS. 2019. Is sizeof for a struct equal to the sum of sizeof of each member? [online]. [cit. 18.5.2020]. Dostupné na: ⟨https://www.geeksforgeeks.org/is-sizeof-for-a-struct-equal-to-the-sum-of-sizeof-of-each-member/⟩.

GUNIŠOVÁ, Valentína a Ján GUNIŠ, 2019. Rekurzívne algoritmy. Oddelenie didaktiky informatiky a podporných technológií. Univerzita P. J. Šafárika. [online]. [cit. 13.2.2020]. Dostupné na: ⟨https://di.ics.upjs.sk/⟩.

HÁLA, Tomáš, 2002. Učebnice Pascalu. Praha : Computer Press. Vydaní druhé. ISBN 80-226-733-7.

HEROUT, Pavel, 2010. Učebnice jazyka C 1. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-383-8.

HEROUT, Pavel, 2010. Učebnice jazyka C 2. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-367-8.

HOROVČÁK, Pavel a Igor PODLUBNÝ, 1997. Úvod do programovania v jazyku C. [online]. [cit. 26.7.2019]. Dostupné na: ⟨http://people.tuke.sk/igor.podlubny/C/⟩.

JANOUŠEK, Jaroslav, ČEKANOVÁ, Adriana a Viera PALMÁROVÁ. Učebnica jazyka C. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://spseke.sk/tutor/projekt/⟩.

KERNIGHAN, W. Brian a Dennis M. RITCHIE, 2019. Programovací jazyk C. Brno : Computer Press. ISBN 978-80-251-4965-2.

KNUTH, Donald Ervin, 2008. Umění programování 1. díl, Základní algoritmy. Brno : Computer Press, ISBN 978-80-251-2025-5.

MADARAS, Tomáš, 2011. Matice. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjjpcaVqPzxAhWDi8MKHSgsAJ4QFjACegQIBBAD&url=http%3A%2F%2Fumv.science.upjs.sk%2Fmadaras%2FMZIa%2FMZIa2011_4.pdf&usg=AOvVaw2uJWo9Ff3TjTq8QIrZmn3N⟩.

MACHOVÁ, Jana a Mária SPIŠÁKOVÁ, 2011. Procedúry a funkcie v Pascale (Zbierka úloh) Dostupné na: ⟨https://encyklopediapoznania.sk/data/eknihy/informatika/procedury_a_funkcie_v_pascale2_-_zbierka_uloh.pdf⟩.

PALMÁROVÁ, Viera, 2003. Dynamické údajové štruktúry. [online]. [cit. 18.7.2020]. Dostupné na: ⟨http://cec.truni.sk/stoffov/dynamicke-udajove-struktury/start.html⟩.

SEDGEWICK, Robert, 2003. Algoritmy v C, Části 1–4, Základy datové struktury, třídení, vyhledávaní. Praha : SoftPress s. r. o. ISBN 80-86497-56-9.

TutorialsPoint, 2021. C Standard Library Reference Tutorial © 2021. [online]. [cit. 26.7.2020]. Dostupné na: ⟨https://www.tutorialspoint.com/c_standard_library/⟩.

WIKIPÉDIA, 2020. Matica. © 2020. [online]. [cit. 26.7.2020]. Dostupné na: ⟨https://sk.wikipedia.org/wiki/Matica_(matematika)⟩.

WIKIPÉDIA, 2021. Fibonacciho postupnosť. © 2021. [online]. [cit. 3.4.2021]. Dostupné na: ⟨https://sk.wikipedia.org/wiki/Fibonacciho_postupnos%C5%A5⟩.

WIRTH, Niklaus, 1989. Algoritmy a štruktúry údajov. Bratislava, Alfa. 2. vydanie. ISBN 80-05-00153-3.

WRÓBLEVSKI, Piotr, 2004. Algoritmy. Datové struktury a programovací techniky. Brno : Computer Press. ISBN 80-251-0343-9.

Názov:	Algoritmizácia a základy štruktúrovaného programovania v jazyku C 2. diel (vysokoškolská učebnica)
Autor:	Ing. Jana Jurinová, PhD.
Recenzenti:	doc. Ing. Michal Čerňanský, PhD. Mgr. Ing. Roman Horváth, PhD.
Vydavateľ:	Univerzita sv. Cyrila a Metoda v Trnave
Rok:	2021
Rozsah:	185 strán/ 10,79 AH
Náklad:	50 ks
Grafická úprava obálky:	Ing. Jana Jurinová, PhD.
Tlač:	elektronická verzia