

Programovanie v jazyku C v riešených príkladoch (1)

© 2018, Anna Bou Ezzeddine, Jozef Tvarožek.

Recenzenti:

doc. RNDr. Petr Šaloun, PhD.

doc. Ing. Slavomír Šimoňák, PhD.

Predhovor

Programovací jazyk C vznikol už pred vyše 40 rokmi, v rokoch 1969 až 1973, avšak doteraz je jedným z najrozšírenejších a najpopulárnejších programovacích jazykov. Jazyk C bol pôvodne určený na implementáciu operačného systému UNIX, no v súčasnosti je všeobecne používaný pri implementácii systémového aj aplikačného softvéru. Významnou výhodou jazyka C je pomerne priamy prístup k systému, čo umožňuje programátorom efektívne pracovať so systémovými zdrojmi. Naučiť sa programovať v jazyku C teda neznamená len získať schopnosť zapisovať v ňom programy (poznať syntax jazyka), ale znalosti jazyka C umožnia človeku hlbšie porozumieť dôležitým vnútorným procesom prebiehajúcim v počítačoch: ako pracuje pamäť, disk, súborový systém, sieťové rozhrania, komunikácia medzi procesmi atď. Znalosť jazyka C tiež predstavuje výborný odrazový mostík na porozumenie ďalším významným programovacím jazykom, ktoré boli odvodené od jeho základných princípov: C++, Java, Objective-C, C# a JavaScript a ktoré v súčasnosti pokrývajú absolútnu väčšinu vytváraného softvéru na svete.

Cieľom tejto publikácie je na praktických príkladoch naučiť sa programovať v jazyku C. Je určená ako pomôcka pre úvodný kurz programovania v jazyku C. Vhodná je pre začínajúcich programátorov, ale veríme, že svoje si nájdu aj mierne pokročilí. V publikácii sme sa formou, nie bežnou v dostupných učebniciach snažili hlbšie rozobrať a precvičiť najmä problémové oblasti, ktoré zvyčajne začínajúcim študentom spôsobujú najväčšie ťažkosti. Pri zápisoch budeme používať ISO štandardizovanú verziu ANSI C (ISO/IEC 9899:1990), ktorá z dôvodu najlepšej prenositeľnosti na rôzne systémy predstavuje najrozšírenejší variant zápisu jazyka C.

Obsah je rozdelený do 6 kapitol. V úvodnej kapitole (0) uvádzame niekoľko motivujúcich programov, ktoré demonštrujú použitie viacerých prvkov pri zápise programov v jazyku C. V nasledujúcich kapitolách (1 až 4) postupne rozoberáme tieto oblasti jazyka C: premenné a výrazy, smerníky, riadiace štruktúry, funkcie, rekurzia, polia, reťazce a dlhé čísla. V každej z týchto kapitol najskôr uvádzame krátke

vysvetlenie a ukážky programov, za ktorými nasledujú úlohy pre samostatnú prácu. Riešenia úloh uvádzame na konci každej kapitoly a veríme, že práve tieto riešenia budú pre čitateľa významným prostriedkom pre hlbšie porozumenie jednotlivých aspektov programovania v jazyku C. V poslednej kapitole (Projekty) uvádzame niekoľko väčších projektov spolu s podrobným postupom riešenia, ktoré sú určené predovšetkým ako možnosť integrovať získané znalosti z ostatných kapitol.

Tešíme sa, že ste sa rozhodli preniknúť do zákutí programovania v jazyku C práve s nami. Jazyk C je malý jazyk, obsahuje pomerne málo kľúčových slov a len základné typy riadiacich štruktúr. Napriek tomu si pri jeho použití môže programátor „ľahko streliť do nohy“, keď aj jeden zatúlaný (chybný) znak v zdrojovom kóde, môže pri vykonaní programu spôsobiť neočakávané správanie, ktoré v konečnom dôsledku spôsobí finančné alebo materiálne škody. Je preto veľmi náročné písať programy, ktoré sú spoľahlivé a bezpečné, v zmysle, že vo všetkých predvídateľných aj nepredvídateľných situáciách nespôsobia až také vážne škody. Programovanie v jazyku C je umenie, ktoré vyžaduje roky práce a skúseností.

V Bratislave, 25. augusta 2018.

Anna Bou Ezzeddine,
Jozef Tvarožek

Stručne o autoroch:

Anna Bou Ezzeddine prednáša programovanie v jazyku C na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave (FIIT STU) a má za sebou vyše 25 rokov praxe s jazykom C, vyše 15 rokov praxe vo vyučovaní programovania v jazyku C a niekoľko tisíc úspešných študentov.

Jozef Tvarožek štvrtý rok prednáša základy programovania v jazyku C na FIIT STU v Bratislave, má za sebou vyše 20 rokov praxe s programovaním v jazyku C a vytvoril viac ako milión riadkov zdrojového kódu.

Webová stránka knihy obsahuje ďalšie podporné materiály:

www.turing.sk/c

Obsah

0	Motivácia	7
1	Jednoduché premenné	19
	1.1 Práca s premennými	19
	1.2 Vstup a výstup	23
	1.3 Smerníky	31
2	Riadiace štruktúry	47
	2.1 Podmienky	47
	2.2 Cykly	51
3	Funkcie	71
	3.1 Jednoduché funkcie	72
	3.2 Rekurzívne funkcie	80
	3.3 Rozsah platnosti identifikátorov	85
	3.4 Smerník na funkciu	94
4	Polia a reťazce	123
	4.1 Jednorozmerné polia	124
	4.2 Reťazce	136
	4.3 Dlhé čísla	145
	4.4 Smerník na smerník	149
	4.5 Viacrozmerné polia	156
	Projekty	207
A	Maximum	208
B	Koleso šťastia	220
C	Kameň, papier, nožnice	226
	Literatúra	233

Kapitola 0

Motivácia

V knihách by sa kapitoly mali číslovať od 1, ale pri programovaní v jazyku C začíname od nuly. S nulou sa pri programovaní v jazyku C stretávame na viacerých dôležitých miestach: prvý prvok poľa je na indexe 0 (napr. **pole[0]**) smerník, ktorý neukazuje na platný údaj (tzv. **NULL**) má hodnotu 0, reťazec ako postupnosť znakov má koniec vyznačený znakom `\0`, globálne a statické premenné sú inicializované na hodnotu 0, a tiež, podobne ako v matematickej logike, hodnota 0 predstavuje nepravdu (`false`) v logických výrazoch.

V tejto prvej kapitole (s poradovým číslom 0) uvádzame niekoľko motivačných príkladov ako príležitosť na prvé zoznámenie s programovaním v jazyku C pre tých, ktorí sa s ním ešte nestretli. Uvedené programy nie sú prehľadom všetkých prvkov jazyka C, ich cieľom je umožniť čitateľovi vnímať rôznorodosť zápisu programov v jazyku C. Druhý príklad v tejto kapitole predstavuje nenápadný úvod do algoritmizácie: dopracujeme sa k rovnakému výsledku viacerými spôsobmi a porovnáme ich vhodnosť (efektívnosť). V poslednom treťom príklade si v programe, ktorý spracuje údaje zo súboru, ukážeme niektoré funkcie zo štandardnej knižnice.

Prvý program

Program je zmysluplný vtedy, keď po jeho vykonaní môžeme pozorovať nejaký výsledok. Vzniknutý výsledok môže mať rôznu formu: výpis na obrazovku, zápis súboru na disk (napr. vytvorenie ZIP archívu), tlač dokumentu na tlačiarňu, odoslanie obsahu webovej stránky po sieťovom rozhraní a pod.

Programy vytvárame v čitateľnom, zvyčajne textovom, zápise tzv. zdrojovom kóde, ktorý je prekladačom (kompilátorom) preložený do binárneho strojového kódu, ktorý potom už dokáže vykonať počítač. Samotný program tvoria len „neživé“ príkazy pre počítač a bez spustenia programu neprebehne žiaden výpočet. Vykonávaný

program nazývame tiež proces: obsahuje kód programu a navyše uchováva aktívny stav pamäte a aktivitu práve vykonávaného programu.

Uvažujme nasledujúci program – zdrojový kód programu v jazyku C – ktorý na obrazovku vypíše správu **Ahoj!**

```
1 // kod1.c -- Janko Hrasko, 4.7.2018
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Ahoj!");
7     return 0;
8 }
```

Najdôležitejšia je hlavná funkcia **main()**, ktorá predstavuje vstupný bod, ktorým sa začne vykonávanie programu. Príkazy, ktoré funkcia obsahuje (riadky 5-8), tzv. telo funkcie, sú postupne vykonávané v poradí ako sú uvedené v zdrojovom kóde: program vypíše správu **Ahoj!** na obrazovku (riadok 6) a činnosť ukončí vrátením čísla 0, tzv. návratová hodnota (riadok 7), čím oznámi operačnému systému, že vykonanie prebehlo úspešne (hodnota 0). V prípade neúspešného vykonania zvyčajne programy vrátia nenulovú hodnotu, pričom rôznymi číslami možno zachytiť rozličné typy chyby, ktoré pri vykonávaní mohli nastať (napr. nedostatok miesta na disku, nedostatok pamäte, chyba pri zápise na sieťové rozhranie a pod.)

Pravidlá, podľa ktorých zapisujeme zdrojový kód, nazývame syntax jazyka. Programátor v jazyku C musí presne dodržiavať tieto pravidlá, inak by prekladač nedokázal zo zdrojového kódu vytvoriť spustiteľný strojový kód, a teda program by nebolo možné vykonať. Napr. ak by v riadku 6 alebo 7 nebola na konci príkazu uvedená ; (bodkočiarka) nedokázal by kompilátor rozlíšiť zapísané príkazy.

Ďalšie časti uvedeného zdrojového kódu sú nevyhnutnou súčasťou programu, aby mohol správu **Ahoj!** na obrazovku vypísať. V riadku 4 je uvedená tzv. hlavička funkcie **main()**: prvé slovo **int** signalizuje, že funkcia vracia celé číslo (typ **int**) a slovo **void** v zátvorkách označuje funkciu bez argumentov. Zložené zátvorky v riadkoch 5 (otváracia) a 8 (ukončovacia) vymedzujú telo funkcie tzv. blok. V riadku 2 je uvedená direktíva pre tzv. preprocesor, ktorý ešte pred prekladom (kompiláciou) zdrojového kódu vloží do kódu funkcie na prácu so vstupmi a výstupmi z knižnice

stdio.h, ktoré napr. umožňujú v programe použiť funkciu **printf()** na výpis správy na obrazovku. V riadku 1 je uvedený tzv. komentár, ktorý uvádza nepovinnú informáciu, ktorá je ale zvyčajne užitočná pre programátora na rýchlejšie porozumenie zdrojovému kódu. Na začiatku zdrojového kódu je vhodné v komentári uviesť meno autora a dátum vytvorenia, resp. poslednej zmeny. Komentáre sú zo zdrojového kódu odstránené pred kompiláciou, a neovplyvňujú vykonanie programu.

Na vytvorenie spustiteľného strojového kódu môžu byť na rôznych operačných systémoch (Microsoft Windows, Linux, macOS, a pod.) dostupné rôzne nástroje a štandardné knižnice. Najpoužívannejšie sú nástroje **gcc** (GNU Compiler Collection), **clang** (Clang) a Microsoft Visual Studio.

Upravený prvý program – načítanie vstupu

Prvý program, ktorý sme uviedli v tejto kapitole, pri každom spustení (vykonaní) vždy len vypíše správu **Ahoj!** a skončí. Programy sú užitočnejšie, keď dokážu vykonávať všeobecnejšie úlohy. Rozšírime program o načítanie vstupu, čo mu umožní reagovať na vstupy od používateľa.

Uvažujme nasledujúci upravený program:

```
1 // kod1.c (upraveny) - Janko Hrasko, 4.7.2018
2 #include <stdio.h>
3
4 int main(void)
5 {
6     char str[100];
7     printf("Ako sa volas? ");
8     scanf("%s", str);
9     printf("Ahoj, %s!", str);
10    return 0;
11 }
```

Ako bude prebiehať vykonanie programu? Vstupný bod je funkcia **main()**, ktorej príkazy sa začnú postupne vykonávať:

- v riadku 6 vyhradíme v pamäti miesto pre 100 znakov, ktoré nazveme **str**,
- v riadku 7 vypíšeme na obrazovku správu **Ako sa volas?**
- v riadku 8 čakáme na vstup od používateľa, ktorý (potom ako ho používateľ zadá) uložíme do **str**. Pri zadaní vstupu z klávesnice používateľ musí zadať nejaké slovo a stlačiť ENTER. Zadané slovo potom program uloží do pamäte,

ktorú sme nato pripravili (vyhradili) v riadku 6. Napr. uvažujme, že používateľ zadal slovo **Jana**, tak po vykonaní riadku 8 bude oblasť pamäti **str** obsahovať znaky **J, a, n, a**, ktoré zodpovedajú zadanému slovu,

- v riadku 9 vypíšeme pozdrav podľa toho, aké slovo zadal používateľ na vstupe, teda napr. vypíše: **Ahoj, Jana!**
- v riadku 10 vykonávaný program ukončí svoju činnosť a vráti 0 (bez chyby).

Pri rôznych vykonaniach programu môže používateľ zadať na vstup rôzne slovo a program bude reagovať príslušným pozdravom. Tento upravený program teda predstavuje veľmi jednoduchú formu rozhovoru, ktorú by sme mohli doplnením ďalších príkazov ďalej rozširovať. Rozšírením tohto prístupu s využitím pokročilých prístupov umelej inteligencie sú tzv. chatboty, ktoré dokážu s používateľmi viesť netriviálne rozhovory, pri ktorých ľudia môžu ľahko nadobudnúť pocit, že sa rozprávajú so skutočným človekom.

Druhý program – výpočet prvočísel

Ukážeme si teraz už zložitejší program na výpočet prvočísel. Prvočíslo je také prirodzené číslo väčšie ako 1, ktoré okrem 1 a samého seba nie je deliteľné iným prirodzeným číslom. Teda napr. čísla 2, 3, 5 a 7 sú prvočísla, ale 4, 6, 8 alebo 10 nie sú prvočísla lebo sú deliteľné 2, a ani 9 nie je prvočíslo, lebo je deliteľné 3.

Nasledujúci program vypíše všetky prvočísla od 2 do zadaného čísla **n**:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, j, n;
6      scanf("%d", &n);
7      for (i = 2; i <= n; i++)
8      {
9          for (j = 2; j*j <= i; j++)
10             if (i % j == 0)
11                 break;
12             if (j*j > i)
13                 printf("%d\n", i);
14         }
15     return 0;
16 }
```

Vykonanie programu začína funkciou **main()**. V riadku 5 vyhradíme v pamäti miesto pre tri premenné **i**, **j** a **n**, s ktorými budeme v programe ďalej pracovať. V riadku 6 najskôr načítame zo vstupu číslo do premennej **n**. Vstup môže zadať používateľ z klávesnice alebo sa prečíta zo súboru, výsledkom vykonania príkazu v riadku 6 je, že premenná **n** (oblasť pamäte pre celé číslo, ktorú sme označili identifikátorom **n**) bude obsahovať načítanú hodnotu, napr. hodnotu 100.

V riadkoch 7 až 14 je cyklus pre riadiacu premennú **i**, ktorý postupne pre **i = 2, ..., n** zopakuje telo cyklu v riadkoch 8 až 14. Pre každú z hodnôt **i** sa vykoná v riadkoch 9 až 11 samostatný cyklus pre riadiacu premennú **j**, postupne pre **j = 2, ..., $\lfloor \sqrt{i} \rfloor$** . Pre konkrétnu hodnotu **j** (a teda aj konkrétnu hodnotu **i**) sa vykoná v riadku 10 príkaz podmienky: ak zvyšok po delení **i** deliteľom **j** je 0, tak pokračuj na príkaz v riadku 11, ktorý ukončí vykonávanie cyklu **j** predtým ako by presiahol hodnotu $\lfloor \sqrt{i} \rfloor$. Teda v prípade, že **j** je deliteľom **i**, tak hodnota premennej **j** v riadku 12 bude menšia alebo rovná ako $\lfloor \sqrt{i} \rfloor$. Podmienka v riadku 12 sa vykoná len v prípade ak aktuálna hodnota **j** je väčšia ako $\lfloor \sqrt{i} \rfloor$ a teda číslo **i** sa funkciou **printf()** vypíše v riadku 13 len ak sme nenašli (okrem 1 a **i**) iného deliteľa čísla **i**.

K rovnakému výsledku sa dopracuje nasledujúci program iným výpočtom:

```
1  #include <stdio.h>
2
3  char sito[1000001];
4  int main(void)
5  {
6      int i, j, n;
7      scanf("%d", &n);
8      for (i = 2; i <= n; i++)
9          if (sito[i] == 0)
10             {
11                 printf("%d\n", i);
12                 for (j = i + i; j <= n; j += i)
13                     sito[j] = 1;
14             }
15     return 0;
16 }
```

Vykonanie programu opäť začína funkciou **main()**. Pred vykonaním príkazov sa však vyhradí pamäť pre pole 1 000 001 celých čísel, ktorú nazveme **sito**. Premenná **sito**

nebude obyčajná jednoduchá premenná ako sú premenné **i**, **j** a **n** deklarované v riadku 6 vo funkcii **main()**. Názov **sito** bude v programe reprezentovať dlhú postupnosť 1 000 001 čísel, pričom s každým z týchto čísel môžeme použitím indexu (zapisujeme v hranatých zátvorkách **[]**) pracovať ako so samostatnou jednoduchou premennou: napr. **sito[0]** je prvé číslo v poli, **sito[1]** druhé číslo a **sito[1000000]** posledné číslo. Hodnoty každého čísla v poli **sito** budú pri spustení programu nastavené na 0, pretože **sito** je tzv. globálna premenná, deklarovaná mimo tela funkcie.

Program pokračuje vykonávaním príkazov vo funkcii **main()**. V riadku 6 najskôr vyhradíme pamäť pre tri premenné **i**, **j** a **n**, s ktorými v programe pracujeme a následne v riadku 7 načítame celé číslo zo vstupu do premennej **n**.

Hlavný výpočet prebieha v cykle pre riadiacu premennú **i** v riadkoch 8-14. Postupne pre každú hodnotu **i = 2, ..., n** sa opakuje telo cyklu (riadky 9-14): podmienka v riadku 9 umožní vykonanie bloku príkazov (riadky 10-14) len ak **sito[i]** je 0, teda ak hodnota **i**-teho (resp. **i+1**-vého ak uvažíme, že indexujeme od 0) čísla v poli **sito** je 0. Ak nie, pokračujeme ďalšou iteráciou cyklu pre ďalšiu hodnotu **i**. Ak áno, tak program vypíše v riadku 11 pomocou funkcie **printf()** číslo **i** (číslo **i** je teda zjavne prvočíslo) a ďalej v cykle pre riadiacu premennú **j** v riadkoch 12-13 priraduje do poľa **sito** na príslušné miesta **sito[j]** hodnoty 1.

Otázka: Do ktorých miest v poli **sito** cyklus zapíše hodnotu 1?

Pre **j = 2i, 3i, ..., n** (ak je **n** deliteľné **i**), teda každý násobok **j** čísla **i** bude mať v prvku **sito[j]** uloženú hodnotu 1 a pri neskoršej kontrole v riadku 9 toto (zložené) číslo v cykle pre **i** preskočíme. Inými slovami, pre násobky nájdených prvočísel do poľa **sito** uložíme 1 a teda ich pri ďalšom hľadaní väčších prvočísel preskočíme. Prvočíslo je v tomto programe také číslo, ktoré nebolo vyznačené ako násobok nejakého iného menšieho prvočísla. Uvedený postup (algoritmus) sa nazýva Eratostenovo sito.

Otázka: Oba uvedené programy pre zadané číslo **n** do 1 000 000 vypíšu rovnakú (správnu) postupnosť prvočísel, ktorý z nich je lepší?

Programy môžeme posudzovať podľa rozličných aspektov a ani jeden z nich nie je vo všetkých ohľadoch jednoznačne „lepší“.

Pripomeňme si ako programy pracujú. Prvý program, určuje prvočísla postupne: pre každé prirodzené číslo väčšie ako 2 hľadá jeho deliteľov iných ako 2 a samotné číslo, ak ho nájde číslo nie je prvočíslom, inak je číslo prvočíslom a vypíše ho. Druhý program s využitím pomocného poľa vyznačuje všetky netriviálne násobky čísel väčších ako 2, pričom čísla, ktoré nie sú vyznačené vypíše ako prvočísla.

V informatike a softvérovom inžinierstve definujeme množstvo kritérií podľa ktorých môžeme hodnotiť algoritmy a programy. Skúsme sa v nasledujúcom texte zamyslieť nad niektorými z nich.

Všeobecnosť použitia (univerzálnosť). Oba programy pre zadané n vypíšu prvočísla do rozsahu n . Druhý program však používa pri výpočte pomocné pole a program nepracuje správne pre hodnoty n väčšie ako je veľkosť poľa (10 000 000). Prvý program teda pracuje pre väčší rozsah vstupných hodnôt n a je preto všeobecnejšie použiteľný.

Zrozumiteľnosť, udržiavateľnosť. Oba programy majú 16 riadkov zdrojového kódu, sú teda pomerne krátke a zrozumiteľné. Všeobecne sú dlhšie programy náročnejšie na porozumenie. V tomto prípade má prvý program 214 znakov a druhý má 234 znakov, čiže prvý program je o trochu kratší. Viac ako konkrétna hodnota veľkosti (dĺžky) zdrojového kódu je dôležitejšia prehľadnosť a zrozumiteľnosť kódu, teda ako rýchlo možno zdrojovému kódu porozumieť, nájsť a opraviť v ňom prípadnú chybu, resp. kód ďalej upravovať (napr. podľa požiadaviek zákazníka, pre ktorého bol vytvorený), tzv. udržiavať. Z tohto hľadiska je postup (vychádzajúci z formálnej definície prvočísel) zvolený v prvom programe jednoduchší na porozumenie. Eratostenovo sito v druhom programe je v porovnaní s tým už odvodený postup, ktorý vyžaduje hlbšie pochopenie vlastností prvočísel.

Efektivita, výkonnosť. Pre číslo $n=100$ oba programy nemerateľne rýchlo vypíšu všetky prvočísla do 100, podobne pre $n=1000$. Čo sa stane pre $n=10\ 000\ 000$? To závisí od počítača, na ktorom programy vykonávame. Pre zmyslupnejšie vyhodnotenie efektivity výpočtu zanedbajme výpis čísel funkciou `printf()`. Programy skúsime spustiť: druhý program vypočíta všetky prvočísla pomerne rýchlo,

naopak prvý program bude pracovať aj niekoľko sekúnd, v porovnaní s druhým programom 10 až 15-krát pomalšie. Je to spôsobené tým, že pri určovaní prvočísel vo vnútornom cykle vykonávame výrazne väčší počet operácií a tiež sú tie operácie náročnejšie (násobenie a delenie je pre počítač náročnejšie ako len sčítanie, ktoré vykonávame v druhom programe). Počet vykonaných operácií a čas výpočtu pre $n=10\ 000\ 000$ sú uvedené v nasledujúcej tabuľke:

	Prvý program	Druhý program
Počet sčítaní	1 736 874 713	30 794 896
Počet porovnaní	1 756 874 711	50 794 895
Počet násobení	3 513 084 844	0
Počet delení	1 746 210 133	0
Čas behu Core i7-4702MQ (2,2 GHz)	9,73 s	0,58 s

Všimnime si, že napriek tomu, že prvý program vykoná viac ako 50-krát toľko operácií, čas výpočtu je len 15-krát pomalší. Dôvod je ten, že v prípade druhého programu používame pomocné pole veľkosti 10 000 001 prvkov (približne 10MB) a prístup k nemu cez pamäťový radič je výrazne pomalší ako bežná aritmetická operácia, ktorú vykonáva procesor vo svojich registroch.

Na tejto ukážke teda môžeme pomerne efektne pozorovať ako využitie pomocnej pamäte urýchľuje výpočty. Úlohou programátora pri návrhu riešení (programov) pre zadané úlohy je vhodne navrhnuť použitie dostupných zdrojov v počítači, tak aby vykonávanie bolo čo najefektívnejšie. Bežné počítače majú v súčasnosti (píše sa rok 2018) v priemere 8GB alebo viac operačnej pamäte a použitie 10 MB v tomto prípade je len veľmi malá časť. V prípade, že program môže (vzhľadom na iné bežiacie programy na počítači) použiť aj väčšiu časť pamäte: napr. 100MB alebo aj 1000MB, neváhajte ju použiť. Na počítači s procesorom i7-4702MQ (2,2 GHz) z roku 2013 s 8GB pamäte (DDR3 1600) trvá druhému programu výpočet pre $n=1\ 000\ 000\ 000$ čas 42,41 sekundy.

Tretí program – spracovanie údajov

Prostredníctvom štandardnej knižnice jazyka C môžeme používať funkcie na prácu s reťazcami, vstupno-výstupné operácie resp. prácu so súbormi, matematické výpočty, správu pamäte, prácu s dátumom a časom, procesmi a ďalšími zdrojmi operačného systému. V poslednom treťom programe si ukážeme ako využitím rôznych prvkov jazyku C môžeme spracovať jednoduchý register osôb uložený v súbore.

Uvažujme nasledujúce údaje, ktoré máme uložené v súbore **osoby.txt**:

1	10
2	1973 Larry Page
3	1975 Angelina Jolie
4	1973 Krasimira Tsaneva-Atanasova
5	1971 Elon Musk
6	1995 Tiera Guinn
7	1962 Lisa Randall
8	1955 Bill Gates
9	1990 Emma Watson
10	1984 Mark Zuckerberg
11	1964 Jeff Bezos

Súbor obsahuje údaje o zaujímavých osobnostiach spoločenského života. V prvom riadku je uvedený počet záznamov a každý z nasledujúcich riadkov obsahuje dátum narodenia a meno osoby. Tieto údaje predstavujú akúsi primitívnu formu databázy, ktorú v súčasnosti používa každý netriviálny systém na správu dát: systémy pre správu zákazníkov, účtovnícke systémy, redakčné systémy na internetových portáloch atď.

Výstupom spracovania bude nasledujúca základná štatistika o veku osôb:

1	Dnes je rok 2018
2	Najmladsia zena: Tiera Guinn (1995)
3	Najstarsi muz: Bill Gates (1955)
4	Priemerny vek: 43.8 rokov

Program vypíše aktuálny rok, nasledovaný menom a rokom narodenia najmladšej ženy v databáze, nasledovaným menom a rokom narodenia najstaršieho muža v databáze a nakoniec vypíše priemerný vek všetkých osôb.

V nasledujúcej ukážke nemusíme nutne porozumieť vytvorenému programu, uvádzame to len ako ukážku funkcií v jazyku C a najmä pestrosti výsledkov, ktoré môžeme dosiahnuť. Všimnime si, že programu sme nezadali aký je dátum a aktuálny

rok – musí to zistiť z operačného systému, nezadali sme mu ani kto z osôb je žena a kto je muž – musí to zistiť zo zadaných údajov a nakoniec nezadali sme mu ani kto je najmladšia žena ani najstarší muž – musí to zistiť zo zadaných údajov.

Spracovanie rozdelíme do viacerých funkcií, čím pomerne zložitý problém rozdelíme na jednoduchšie úlohy. Údaje pre každú osobu si zo súboru načítame do štruktúry (**struct Osoba**, riadky programu 6-11), ktorá združuje viacero jednoduchých premenných (**rok** narodenia, **meno**, ako aj príznak, či osoba je **zena**) do jedného celku a umožňuje nám s nimi pracovať v programe spolu.

Funkcia **nacitaj()** otvorí na čítanie súbor **osoby.txt**, prečíta z neho počet osôb (**n**), vyhradí v pamäti miesto pre údaje o každej osobe (pole **o**) a potom postupne pre každú z osôb v súbore načíta údaje o **i**-tej osobe do prvku **o[i]**.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5
6  struct Osoba
7  {
8      int rok;
9      char *meno;
10     int zena;
11 };
12
13 struct Osoba *o;
14 int n;
15
16 void nacitaj(void)
17 {
18     FILE *f = fopen("osoby.txt", "rt");
19     char buf[100];
20     int i;
21     fscanf(f, "%d", &n);
22     o = malloc(n * sizeof(struct Osoba));
23     for (i = 0; i < n; i++)
24     {
25         fscanf(f, "%d %[^\\n]s", &o[i].rok, buf);
26         o[i].meno = strdup(buf);
27         sscanf(o[i].meno, "%s", buf);
28         o[i].zena = buf[strlen(buf) - 1] == 'a';
29     }
30 }
```

Všimnime si, že funkcia **nacitaj()** popri načítaní **i**-tej osoby ešte v riadkoch 27-28 heuristicky určí, či osoba je žena alebo nie. Použité heuristické pravidlo považuje za ženu osobu, pre ktorú prvá časť mena osoby končí na znak '**a**'. Toto pravidlo neplatí univerzálne (napr. pre ženu Hillary alebo muža Luka) ale v našom prípade presne pokryje všetky ženy v dátovej sade. Reálne používané programy zvyčajne používajú samostatný príznak, ktorý by bol uvedený v súbore.

Program tiež obsahuje pomocnú funkciu **aktualny_rok()**, ktorá z operačného systému určí aktuálny rok a zistenú hodnotu vráti ako tzv. návratovú hodnotu, ktorú môže hlavná funkcia **main()** naplniť do lokálnej premennej (**rok**) a použiť v ďalších príkazoch:

```
31 int aktualny_rok(void)
32 {
33     time_t t;
34     time(&t);
35     return localtime(&t)->tm_year + 1900;
36 }
```

Spracovanie v hlavnej funkcii potom najskôr zistí aktuálny rok volaním funkcie **aktualny_rok()**, ďalej volaním funkcie **nacitaj()** načíta údaje zo súboru do pamäte do globálneho poľa údajov (**o**). Keď sú údaje načítané v pamäti bežiacieho programu, možno k nim pristupovať a spracúvať ich:

```
37 int main(void)
38 {
39     int i, min, max, sucet, rok = aktualny_rok();
40     nacitaj();
41     for (i = sucet = 0, min = max = -1; i < n; i++)
42     {
43         if (o[i].zena && (min < 0 || o[min].rok < o[i].rok))
44             min = i;
45         if (!o[i].zena && (max < 0 || o[max].rok > o[i].rok))
46             max = i;
47         sucet += rok - o[i].rok;
48     }
49
50     printf("Dnes je rok %d\n", rok);
51     printf("Najmladsia zena: %s (%d)\n", o[min].meno, o[min].rok);
52     printf("Najstarsi muz: %s (%d)\n", o[max].meno, o[max].rok);
53     printf("Priemerny vek: %.1lf rokov\n", (double)sucet / n);
54     return 0;
55 }
```

Určenie základných štatistických informácií už prebieha pomerne priamočiaro. V cykle (riadky 41-48) postupne prejdeme každú osobu a priebežne si budeme pamätať index najmladšej ženy (**min**), index najstaršieho muža (**max**), súčet rokov (**sucet**). Pre **i**-tu osobu (**i=0, ..., n-1**) rozlíšime prípady:

- riadky 43-44: v prípade, že **i**-ta osoba je žena, tak porovnáme, či jej rok narodenia je väčší ako doteraz zapamätaný (a teda je mladšia), ak áno, zapamätáme si ju ako priebežne najmladšiu,
- riadky 45-46: v prípade, že **i**-ta osoba je muž, tak porovnáme, či jeho rok narodenia je menší ako doteraz zapamätaný (a teda je starší), ak áno, zapamätáme si ho ako priebežne najstaršieho.

Nakoniec vypíšeme aktuálny rok, meno aj rok narodenia najmladšej ženy, meno aj rok narodenia najstaršieho muža, ako aj priemerný vek (zistený súčet vydáme počtom osôb).

Povzbudenie na záver

Programy uvedené v tejto kapitole sú ako ukážka pestrosti problémov, ktoré možno programovaním (v jazyku C) riešiť.

V nasledujúcich kapitolách budeme na ukážkových príkladoch a úlohách pre samostatnú prácu spoznávať prvky programovania v jazyku C. Preberané princípy nie sú špecifické len pre jazyk C ale sú do veľkej miery použiteľné aj pri programovaní v iných programovacích jazykoch, nielen preto, že jazyk C poslúžil ako základ mnohým ďalším jazykom, ale najmä preto, že koncepty ako premenná, podmienka, iterácia (cyklus), funkcia, rekurzia, pole, reťazec a smerník sú všeobecné „platné“ koncepty, ktoré sa používajú prakticky všade aj v každodennom živote.

Nakoniec priateľské upozornenie: „**Programovanie nie je šport pre divákov.**“, inšpirované výrokom (o matematike), ktorý vyslovil svetoznámy maďarský matematik György Pólya (1887-1985). Lepšie programovať sa naučíte jedine tak, že prestanete byť divákmi, ktorí len čítajú knihy a internetové fóra, ale začnete písať zdrojový kód vlastnými rukami, nohami, nosom alebo akokoľvek inak bez toho, aby ste hotové kódy opisovali z kníh alebo kopirovali z internetu... Prajeme veľa úspechov!

Kapitola 1

Jednoduché premenné

1.1 Práca s premennými

V tejto kapitole predstavíme ukladanie premenných v pamäti a prácu s nimi. Cieľom je napísať jednoduché programy v jazyku C, v ktorých budú použité rôzne typy premenných a aritmetické operácie s týmito premennými. Rozoberieme tiež načítanie zo vstupu a výpis na výstup.

Uvažujme nasledujúcu postupnosť príkazov:

1	<code>int a, b, c, d;</code>
2	<code>a = 47;</code>
3	<code>b = 42;</code>
4	<code>c = a + b;</code>
5	<code>d = c + b;</code>

V riadku 1 definujeme štyri premenné typu `int`. Týmto príkazom vyčleňujeme štyri miesta v pamäti, ktoré nazveme **a**, **b**, **c** a **d**. Názov premennej nazývame tiež aj identifikátor premennej: **a**, **b**, **c** a **d** sú identifikátory premenných, pomocou ktorých môžeme s premennými v zdrojovom kóde pracovať. Do každého z týchto štyroch miest (resp. štyroch premenných) môžeme podľa určeného typu načítať a zapísať celé čísla (kladné aj záporné aj nulu).

Každá premenná musí mať vyhradené miesto v pamäti. Jeho veľkosť je určená dátovým typom premennej. Tri najzákladnejšie dátové typy sú:

- znak (**char**, z anglického character) má veľkosť 1 bajt (angl. byte),
- celé číslo (**int**, z anglického integer) má veľkosť 4 bajty (angl. bytes),
- desatinné číslo s dvojitou presnosťou (**double**) má veľkosť 8 bajtov.

Reprezentácia desatinného čísla so základnou presnosťou (**float**) sa kvôli hrubým nepresnostiam vo výpočtoch v súčasnosti už nezvykne používať.

Pri zápise desatinných čísel sa pre oddelenie desatinnej časti používa bodka, nie čiarka ako v slovenskom jazyku (t. j. v zdrojovom kóde píšeme **0.47** nie **0,47**).

Na dané miesto v pamäti sa jednoznačne odkazujeme pomocou adresy.

Upozornenie: Pri práci s premennými je veľmi dôležité neustále vnímať rozdiel medzi hodnotou premennej a adresou premennej.

Hodnota premennej je samotný údaj, ktorý sme si do premennej uložili (napr. číslo 47) a táto hodnota je uložená niekde v pamäti, pričom presné miesto v pamäti, kde je hodnota uložená, určuje adresa premennej.

Adresa premennej je informácia o polohe (presnom umiestnení) tejto premennej v pamäti (napr. 140720561767844, čo sa v hexadecimálnej sústave zapisuje 0x7ffe5f48f644). V prípade premennej, ktorá má veľkosť 1 bajt (byte), zodpovedá adresa presne umiestneniu hodnoty. V prípade, že premenná má väčšiu veľkosť (napr. celé číslo – **int** – má veľkosť 4 bajty) zodpovedá adresa premennej prvému bajtu premennej a potom v pamäti nasledujú ďalšie bajty, ktoré sú umiestnené bezprostredne za sebou. Adresu premennej možno v zdrojovom kóde vyjadriť symbolom **&** (ampersand) pri názve premennej, teda napr. **&a** má hodnotu 140720561767844.

Uloženie premenných definovaných v riadku 2 môže byť znázornené takto:

int d	⊗	140720561767832
int c	⊗	140720561767836
int b	⊗	140720561767840
int a	⊗	140720561767844

1	int a, b, c, d;
2	a = 47;
3	b = 42;
4	c = a + b;
5	d = c + b;

V tomto prípade je premenná **a** uložená na adrese 140720561767844, **b** na adrese 140720561767840, **c** na adrese 140720561767836 a premenná **d** na adrese 140720561767832. Pre každú premennú typu **int** sú v pamäti vyčlenené 4 bajty. Symbol ⊗ označuje, že na danom mieste v pamäti je tzv. neurčitá hodnota (angl. indeterminate value) a použitie neurčitej (neinicializovanej) hodnoty v programe môže spôsobiť tzv. nedefinované správanie (angl. undefined behavior) celého programu. Môžeme si to predstaviť tak, že sú tam nepoužiteľné a potenciálne nebezpečné údaje, akési „smeti“, ktoré tam napr. zostali od predchádzajúceho použitia tejto pamäte, a ak by sme takého hodnoty používali, program by sa správal nepredvídateľne a vypočítané hodnoty (výsledky programu) by mohli byť nesprávne.

Upozornenie: Pri písaní zdrojového kódu programu chceme, aby výsledky vykonania nášho programu boli spoľahlivé, teda, aby program jednoznačne (vždy rovnako) pracoval presne podľa zdrojového kódu programu, a preto je kritické vyhnúť sa nedefinovanému správaniu programu.

Významná časť nedefinovaného správania, ktoré sa v programoch nachádza, je spôsobená použitím neurčitých hodnôt, preto je veľmi dôležité každú premennú, ktorú v programe používame, inicializovať – nastaviť jej počiatočnú hodnotu hneď pri definovaní premennej, resp. pred prvým použitím premennej v zdrojovom kóde.

V riadkoch 2 až 5 sú premenné inicializované na tieto hodnoty:

int d	131	140720561767832
int c	89	140720561767836
int b	42	140720561767840
int a	47	140720561767844

1	int a, b, c, d;
2	a = 47;
3	b = 42;
4	c = a + b;
5	d = c + b;

Úloha 1-1

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int x, y, z;
2	x = 2;
3	x = x + x;
4	x = x + x;
5	y = x;
6	z = x + y;

int x		140732672863916
int y		140732672863912
int z		140732672863908

Úloha 1-2

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int x, y;
2	x = y;

int x		140737258267468
int y		140737258267464

Úloha 1-3

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int x = 10, y = 5, z;
2	++x;
3	x++;
4	y = ++x;
5	y = x++;
6	z = --y + x++;

int x	<input type="text"/>	140734926495068
int y	<input type="text"/>	140734926495064
int z	<input type="text"/>	140734926495060

Pomôcka: Unárne operátory typu inkrement ++ a dekrement -- sa používajú v dvoch formách, ako:

- prefixová forma **++p** (resp. **--p**) hodnota premennej **p** je inkrementovaná a potom je táto zvýšená hodnota použitá (napr. vo výraze), resp. hodnota premennej **p** je dekrementovaná pred jej použitím,
- postfixová forma **p++** (resp. **p--**) pri ktorej sa najskôr použije pôvodná hodnota premennej **p** a až potom je inkrementovaná (a zvýšená hodnota sa použije až pri nasledujúcom použití premennej), podobne v prípade dekrementovania: najskôr sa použije pôvodná hodnota a až potom sa zníži.

Inkrementovanie v prípade dátového typu obsahujúceho číslo (napr. **int**) zvýši hodnotu o 1, podobne dekrementovanie zníži hodnotu o 1. Ako uvidíme neskôr, v kapitole 2, v prípade iných dátových typov (smerníkov) môže inkrementovanie (resp. dekrementovanie) zmeniť hodnotu aj o viac ako 1.

Úloha 1-4

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int x = 6, y = 4;
2	double z = 2;
3	y += x;
4	y *= x-3;
5	y /= ++x;
6	x /= z;
7	z /= x;
8	x %= y;

int x	<input type="text"/>	140727123873612
int y	<input type="text"/>	140727123873608
double z	<input type="text"/>	140727123873600

Pomôcka: Operátory +=, -=, *=, /=, %= sú spojením aritmetickej operácie a priradenia novej hodnoty do jedného príkazu. Znak operácie (napr. +) a znak rovná sa (=) sa zapisujú v tomto poradí bezprostredne pri sebe bez medzery.

Úloha 1-5

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int v = 800;
2	unsigned char a = 300, k = 220;
3	v += k;
4	k += v;
5	v = a + k;

int v	<input type="text"/>	140720552750924
unsigned char a	<input type="text"/>	140720552750923
unsigned char k	<input type="text"/>	140720552750922

Úloha 1-6

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int v = 120;
2	double k = 400, s = 600;
3	v += s/k;
4	k += v + v;
5	v += k/s;

int v	<input type="text"/>	140733686302204
double k	<input type="text"/>	140733686302192
double s	<input type="text"/>	140733686302184

1.2 Vstup a výstup

Program je zmysluplný vtedy, keď po jeho vykonaní môžeme pozorovať nejaký výsledok. Vzniknutý výsledok môže mať rôznu formu, najčastejšie formou výpisu na obrazovku. Hovoríme, že program vypísal výsledok na výstup. Výstupom môže byť už spomínaná obrazovka počítača, alebo zápis súboru na disku, alebo odoslanie údajov (paketov) po sieťovom rozhraní.

Ešte užitočnejší je program vtedy, keď mu môžeme zadať s akými hodnotami vykoná zadané výpočty. Bez zadania vstupu by program pri každom vykonaní vykonával výpočty nad rovnakými údajmi a teda na výstup by dal vždy rovnaký výsledok. Je teda veľmi užitočné, ak program načíta údaje zo vstupu, spracuje ich, a na výstup vypíše výsledok spracovania vstupných údajov. Vstup môže byť celkom jednoduchý ako jedno celé číslo, text, alebo aj komplikovanejší ako množina súborov určená na vytvorenie ZIP archívu.

V predchádzajúcej časti (1.1) sme vždy v programe pracovali s konkrétnymi hodnotami premenných. V tejto časti si ukážeme ako môžeme načítať hodnoty premenných zo vstupu a predstavíme rozličné spôsoby ako môžeme vypísať hodnoty premenných na výstup.

Načítanie a výpis premenných nie je priamou súčasťou syntaxe jazyka C a je potrebné v programe použiť knižničné funkcie. Norma ANSI C určuje sadu knižničných funkcií, ktoré majú byť spolu s prekladačom jazyka C dodané a programy ich môžu používať. Súbor týchto funkcií tvorí štandardnú knižnicu jazyka C. Táto knižnica obsahuje funkcie na vykonávanie vstupno-výstupných (I/O) operácií (**stdio.h**), vykonávanie matematických výpočtov (**math.h**), funkcie na prácu s reťazcami znakov (**string.h**) a ďalšie (napr. **stdlib.h**).

Na začiatok budeme na načítanie vstupných hodnôt potrebovať funkciu **scanf()** a na výpis funkciu **printf()**. Obe funkcie sú súčasťou hlavičkového súboru **stdio.h**. S ďalšími vstupno-výstupnými funkciami sa postupne stretne v ďalších častiach. Hlavičkové súbory sa do programu vkladajú pomocou tzv. direktívy preprocesora **#include**, ktorá zabezpečí, aby preprocesor ešte pred kompiláciou nášho zdrojového kódu začlenil do procesu vytvárania programu aj obsah príslušného súboru (napr. hlavičkového súboru **stdio.h**).

Náš program teda začíname riadkom:

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

Takto môžeme v programe používať vstupno-výstupné (I/O) knižničné funkcie.

Funkcia **printf()** určená na výpis na výstup má nasledujúci tvar:

```
int printf(char *formatovaci_retazec, argumenty...);
```

Funkcia má návratovú hodnotu typu `int` – po vykonaní funkcia `printf()` vracia počet vypisovaných znakov, čo nás zvyčajne zaujíma iba málokedy. Pre nás je teraz dôležitejší tvar formátovacieho reťazca a argumenty. Tento reťazec obsahuje znaky, ktoré sa majú vypisovať a špecifikátory formátu akým budú vypisované odpovedajúce argumenty. Argumenty predstavujú **zoznam premenných alebo výrazov** oddelených čiarkami. Špecifikátory začínajú znakom `%`. Sú priradované argumentom zľava doprava a mal by ich byť rovnaký počet ako argumentov. Všeobecný tvar špecifikácie formátu (jedného argumentu) je nasledujúci:

`%[-][sirka-vstupu][.][presnost] špecifikator format`

Nepovinné časti sú uvedené v zátvorkách. Štandardne sú číselné výstupy zarovnané sprava. Ak ich chceme zarovnať zľava použijeme znamienko minus (`-`) hneď za znakom `%`. Najčastejšie používané špecifikátory sú: `%c`, `%d`, `%f`, `%lf`, `%g`, `%s`, `%u`.

Ich použitie si ukážeme v nasledujúcom programe.

```

1  #include <stdio.h>
2
3  int main (void)
4  {
5      char c = 'a';
6      int i = 33, u = -22;
7      double r = 55;
8      printf("%c %c+1 %*c\n", c, c, 5, c);
9      printf("ASCII hodnota znaku %c je %d\n", c, c);
10     printf("<%u> <%d>\n", u, u);
11     printf("<-%d> <-%5d> <%+05d> <%+5d>\n", i, i, i, i);
12     printf("<%g> <%lf> <%.3lf>\n", r, r, r);
13     printf("<-%10d> %c <%.10lf>\n", i, c, r);
14     printf("<%+0*d>\n", i/5, i);
15     return 0;
16 }
```

Výstup programu je nasledovný:

```

1  a a+1      a
2  ASCII hodnota znaku a je 97
3  <4294967274> <-22>
4  <-33> <33   > <+0033> <  +33>
5  <55> <55.000000> <55.000>
6  <-          33> a <55.0000000000>
7  <+00033>
8
```

Experimentujte s použitím rozličných špecifikátorov v rôznych prípadoch použitia a s výpisom rôznych typov premenných v rôznych formátoch.

Formátovací reťazec umožňuje tiež použiť rôzne špeciálne znaky, vďaka ktorým môžeme vypísať aj také znaky, ktoré z dôvodu syntaxe (zápisu) reťazcov v jazyku C nemôžeme priamo použiť. Tieto znaky sú zapísané po znaku \ (lomka). Ako vypíšeme na výstup znak %, ktorý vo formátovacom reťazci slúži na špecifikáciu formátu výpisu argumentu? Preskúmajte nasledujúci program:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5      printf("Posun na nový riadok\n");
6      printf("Vypis apostrof: '\n");
7      printf("Vypis uvozovky: \"\n");
8      printf("Vypis lomka: \\n");
9      printf("Vypis \t tabulator\n");
10     printf("Pracujeme na 100%%.\n");
11     return 0;
12 }
```

Výstup programu je tento:

```
1  Posun na nový riadok
2  Vypis apostrof: '
3  Vypis uvozovky: "
4  Vypis lomka: \
5  Vypis  tabulator
6  Pracujeme na 100%.
7
```

Funkcia **scanf()** určená na načítanie hodnôt zo vstupu do pamäte (premenných) má nasledujúci tvar:

```
int scanf(char *formatovaci_retezec, argumenty...);
```

Na prvý pohľad vyzerá funkcia **scanf()** identicky ako funkcia **printf()**, interpretácia jej parametrov aj návratovej hodnoty je však iná.

Formátovací reťazec pre funkciu **scanf()** spravidla neobsahuje znaky, ale iba špecifikátory podobné ako pri funkcii **printf()** (%c, %d, %f, %lf, %g, %s, %u). Týmto spôsobom určujeme do akého typu premennej bude hodnota zo vstupu načítaná.

Argumenty predstavujú **zoznam adries premenných** oddelených čiarkami. Návratová hodnota funkcie **scanf()** je typu **int** (celé číslo) a je veľmi dôležitá, pretože predstavuje počet úspešne načítaných prvkov.

Nasledujúcim príkazom načítame tri celé čísla zo vstupu do premenných **x**, **y**, **z** a počet načítaných čísel si uložíme do premennej **pocet**:

```
int pocet = scanf("%d %d %d", &x, &y, &z);
```

Čísla na vstupe musia byť oddelené bielymi znakmi (ako napr. medzera, viac medzier, nový riadok, tabulátor). Do premennej **pocet** si uložíme návratovú hodnotu, ktorú vráti volanie funkcie pre konkrétny vstup. Hodnota premennej **pocet** sa teda môže líšiť podľa toho, aké údaje sme programu zadali na vstupe. Napr. ak by na vstupe boli len dve čísla, tak vyššie uvedený príkaz načíta hodnotu prvého čísla do pamäte premennej **x**, hodnotu druhého do pamäte premennej **y**, ale pamäť premennej **z** zostane nezmenená, pričom funkcia vráti číslo 2 (lebo načítala dve čísla), a teda hodnota premennej **pocet** bude inicializovaná hodnotou 2.

Pri použití funkcie **scanf()** musíme uvádzať adresy premenných, do ktorých chceme podľa formátovacieho reťazcu hodnoty načítať. Preto v zápise príkazu používame symbol **&** (ampersand), ktorý slúži na vyjadrenie hodnoty adresy premennej.

Dôležité upozornenie: Nepoužitie alebo nesprávne používanie znaku **&** je jedna z najčastejších chýb pri načítaní pomocou funkcie **scanf()**.

Otázka: Prečo je potrebné pri načítaní hodnôt zo vstupu (funkciou **scanf()**) uviesť zoznam adres premenných? Napr. **scanf("%d", &n);**

Odpoveď: Veľmi zjednodušene si môžeme vykonanie (každého) programu predstaviť ako postupnosť inštrukcií, ktorých hlavná úloha je upraviť obsah pamäte. Keď chceme v programe upraviť nejakú oblasť pamäti, tak musíme vedieť, ktorú konkrétne pamäť upravujeme. Teda v prípade načítania vstupu funkciou **scanf()** potrebujeme pri volaní funkcie **scanf()** okrem formátu uviesť aj adresu pamäte, do ktorej má hodnoty údajov zo vstupu zapísať. Mohli by sme teda vykonať aj príkaz

`scanf("%d", 123456);` ktorý by hodnotu celého čísla zo vstupu uložil do pamäte na adrese 123456. Je to však nesprávne a vyložene nebezpečné načítanie.

Nadväzujúca otázka: Prečo nie je možné (správne) uvádzať adresy v programe ako konkrétne čísla? Napr. `scanf("%d", 140720561767844);`

Odpoveď: V zdrojovom kóde programu nemôžeme uviesť hodnoty adries ako konkrétne čísla, pretože každé spustenie programu v operačnom systéme môže prideliť programu iný blok pamäte vzhľadom na absolútne poradie v rámci celej pamäte počítača, a teda zapísanie do vopred zadanej konkrétnej pamäte, ktorá nie je pridelená bežiacemu programu môže spôsobiť neplatný prístup do pamäte (predčasné ukončenie programu) alebo môže spôsobiť nedefinované správanie.

Zdrojový kód programu v jazyku C uľahčuje programátorovi prácu s pamäťou tým, že programátor nezadáva konkrétne hodnoty adries pamäte ale umožňuje mu úpravy pamäte vykonávať zrozumiteľne – využitím identifikátorov premenných. Použitím identifikátora (názvu) premennej vieme v zdrojovom kóde ľahko získať hodnotu adresy (umiestnenia v pamäti) premennej. Napr. **&a** vyjadruje hodnotu adresy premennej **a** pri vykonaní programu, a preto môže pri rôznych vykonaniach (spusteniach) programu nadobúdať rôzne hodnoty, napr.: 140720561767844 alebo 140733686302204, alebo akékoľvek iné číslo, ktoré zodpovedá umiestneniu v pamäti.

Každá premenná, ktorú v programe používame musí byť deklarovaná (určený jej typ – napr. `int`) a definovaná (program vyhradí pre identifikátor premennej rozsah pamäte, ktorý zodpovedá typu premennej – napr. `int` je 32-bitové číslo, teda 4 bajty). Keď v zdrojovom kóde do premennej priradíme hodnotu (napr. inicializácia hodnotou 0), tak program pri vykonaní tohto príkazu do príslušnej pamäte uloží túto hodnotu. Napr. inicializácia premennej priradením: `n = 0;` naplní obsah pamäte, ktorý je vyhradený pre premennú **n**. Adresu premennej **n**, resp. umiestnenie (poradie byte v rámci pamäte počítača) vyhradenej pamäti pre premennú **n** vyjadrujeme v zdrojovom kóde **&n**.

Príkaz `scanf("%d", &n);` môžeme teda stručne interpretovať ako príkaz, ktorý do premennej **n** (do jej pamäte) priradí hodnotu celého čísla, ktoré sa nachádza na vstupe, pričom načítavanie vstupu sa posunie za načítané číslo.

Úloha 1-7

Na vstupe je desatinné číslo. Doplňte chýbajúce časti v nasledujúcom programe, ktorý načíta číslo zo vstupu a na výstup vypíše iba jeho celú časť. Zamyslite sa nad rôznymi spôsobmi ako možno program doplniť.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6      double realne;
7      int cele;
8      scanf("_____", _____);
9      // vypis cislo bez desatinnej casti
10     printf("_____\n", _____);
11     return 0;
12 }
```

Vstup:

1	28.50
---	-------

Výstup:

1	28
2	

Koľko rôznych spôsobov ste našli? Existujú aspoň štyri: skúste okrem bežných typových konverzií použiť aj funkcie z matematickej knižnice **math.h**. Môžeme na výpis celočíselnej časti desatinného čísla použiť aj nasledujúci príkaz?

```
printf("%.0lf\n", realne);
```

Úloha 1-8

Index telesnej hmotnosti (angl. Body mass index, BMI) určuje obezitu človeka podľa vzorca:

$$BMI = \frac{\text{hmotnosť v kg}}{(\text{výška v m})^2}$$

Nasledujúci program načíta dve desatinné čísla: výšku osoby v cm a hmotnosť v kg, a na výstup vypíše hodnotu BMI zaokrúhlenú na tri desatinné miesta. Doplňte chýbajúce časti programu.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      _____ v, m;
7      scanf("_____", _____, _____);
8      v _____ 100.0;
9      printf("_____\n", _____);
10     return 0;
11 }

```

Vstup:

1	170.0 58.8
---	------------

Výstup:

1	20.346
2	

Zamyslite sa aj nad alternatívnou úlohou: Na vstupe je hodnota BMI a hmotnosť v kg, na výstup vypíšete výšku človeka v cm zaokrúhlenú na jedno desatinné miesto. Ukážka vstupu: 20.346 58.8

Výstup pre ukážkový vstup: 170.0

Možno doplniť pôvodný program, aby riešil túto alternatívnu úlohu? Ak nie, aké úpravy je potrebné vykonať?

Úloha 1-9

Určite výstup vykonania nasledujúceho programu s uvedeným vstupom.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b, c, d, e;
6      scanf("%d %d %d %d %d", &d, &c, &a, &b, &e);
7      b *= c += e -= a / 2 - 5;
8      printf("%d %d %d %d %d\n", a, b, c, d, e);
9      return 0;
10 }

```

Vstup:

1	0 1 2 2 4
---	-----------

Pomôcka: Výraz s viacerými operátormi priradenia (=) vyhodnocujúte sprava doľava.

Zamyslite sa aj nad alternatívnou úlohou: Určite vstup pre ktorý uvedený program vypíše nasledujúci výstup: a) 4 3 2 1 0, b) 3 4 2 3 4.

Úloha 1-10

Na vstupe je najviac 5 čísel. Každé z nich je alebo celé číslo alebo desatinné číslo. Napíšte program, ktorý na výstup vypíše súčet všetkých čísel na vstupe. Program by mal pracovať správne bez ohľadu na počet čísel na vstupe, ktorých je najviac 5.

Úloha 1-11

Doplňte chýbajúce časti programu, ktorý vypíše uvedený výstup.

1	#include <stdio.h>
2	
3	int main(void)
4	{
5	int i=42;
6	printf("<-%>\n", i);
7	printf("<%>\n", i);
8	printf("<%>\n", i);
9	printf("<%>\n", i);
10	printf("<%>\n", i);
11	printf("<%>\n", i/5, i);
12	return 0;
13	}

Výstup:	
1	<-42>
2	< 42>
3	<42 >
4	<+0042>
5	< +42>
6	<+0000042>
7	

1.3 Smerníky

V predchádzajúcich častiach sme používali premenné, ktorých hodnota bola v programe použitá priamo. V jazyku C však existuje špeciálny typ premennej tzv. smerník (angl. pointer), ktorého hodnota je adresa v pamäti. Umožňuje s hodnotami premenných pracovať nepriamo tak, že hodnota smerníka (adresa v pamäti) zodpovedá pamäti, v ktorej sa príslušná hodnota (premennej) nachádza.

Úloha smerníka je pamätať si adresu v pamäti, na ktorej sa nachádza hodnota, s ktorou chceme v programe pracovať. Smerníky predstavujú hlavný nástroj efektívneho prístupu k pamäti. Zvládnutie práce so smerníkmi predstavuje základ zručného písania programov v jazyku C.

Cieľom tejto časti je pomocou jednoduchých príkladov pochopiť základy používania smerníkov. Po naštudovaní tejto časti by ste mali vedieť ako máte prístupovať k hodnote premennej, ak máte k dispozícii jej adresu a ako s touto




premennou pracovať. V ďalších kapitolách budeme hovoriť o pokročilejších možnostiach používania smerníkov v jazyku C a pre ich pochopenie je nutné túto časť detailne naštudovať.


Uvažujme nasledujúcu postupnosť príkazov:

1	<code>int x, y;</code>
2	<code>int *p;</code>
3	<code>p = &x;</code>
4	<code>*p = 5;</code>
5	<code>y = *p + 2;</code>

V riadku 1 sú definované dve premenné typu `int`. V riadku 2 je v definícii premennej `p` použitý symbol `*` (hviezdička), ale nie je to operátor násobenia (museli by tam byť dva operandy; operand pri násobení môže byť identifikátor premennej alebo aritmetický výraz). V tomto prípade `*` označuje definíciu smerníku, teda, že premenná `p` je premenná typu `int*` – smerník na `int`. Hodnota premennej typu `int*` je adresa v pamäti kde je uložené celé číslo (`int`).

Po definíciách premenných v riadkoch 1 a 2 môžeme hodnoty premenných a ich uloženie v pamäti znázorniť takto:

<code>int x</code>		140726106253484
<code>int y</code>		140726106253480
<code>int *p</code>		140726106253472

Všetky pamäťové miesta obsahujú iba neurčité hodnoty , resp. „smeti“ s ktorými nie je možné pracovať bez inicializácie. Pripomeňme si z časti 1.1: ak by sme pracovali s takýmito neurčitými (resp. neinicializovanými) hodnotami, spôsobí to nedefinované správanie programu a vypočítané výsledky nie sú spoľahlivé.

V riadku 3 je použitý ampersand (`&`) ako tzv. **referenčný operátor**.

So symbolom `&` sa môžeme v jazyku C stretnúť aj pri logických a bitových operáciách. V tomto prípade nemá ale symbol `&` s týmito operáciami nič spoločné. V tomto prípade je operátor `&` aplikovaný na premennú `x` a vyjadruje adresu tejto premennej. Preto po vykonaní priradenia v riadku 3 budú premenné v pamäti obsahovať tieto hodnoty:

int x	<table border="1"><tr><td>⊗</td></tr></table>	⊗	140726106253484
⊗			
int y	<table border="1"><tr><td>⊗</td></tr></table>	⊗	140726106253480
⊗			
int *p	<table border="1"><tr><td>140726106253484</td></tr></table>	140726106253484	140726106253472
140726106253484			

1	int x, y;
2	int *p;
3	p = &x;
4	*p = 5;
5	y = *p + 2;

Teraz **p** obsahuje hodnotu 140726106253484, teda adresu premennej **x**. V tomto prípade premenná (smerník) **p** je typu **int*** a teda smerník (premenná) **p** ukazuje na premennú **x** typu **int**. Hovoríme, že premenná **p** je **smerníkom (ukazovateľom)** na typ **int**.

V riadku 4 je opäť použitý symbol ***** (hviezdčička) ale inak ako v riadku 2. V riadku 2 je ***** použitá ako časť definície premennej (nie je to operátor, iba symbol, ktorý prekladaču oznamuje, že daná premenná bude smerník). V riadku 4 je symbol ***** použitý ako tzv. **dereferenčný operátor**. Pomocou dereferenčného operátora pristúpime k hodnote, na ktorú smerník ukazuje. V riadku 4 príkazom priradenia uložíme hodnotu 5 do pamäte, na ktorú ukazuje smerník **p**, teda do premennej **x**. Po vykonaní priradenia v riadku 4 budú premenné v pamäti obsahovať tieto hodnoty:

int x	<table border="1"><tr><td>5</td></tr></table>	5	140726106253484
5			
int y	<table border="1"><tr><td>⊗</td></tr></table>	⊗	140726106253480
⊗			
int *p	<table border="1"><tr><td>140726106253484</td></tr></table>	140726106253484	140726106253472
140726106253484			

Všimnime si, že týmto spôsobom (využitím smerníka ukazujúceho na **x**) sme bez priameho použitia premennej **x** upravili jej hodnotu v pamäti. Ak chceme v programe pomocou smerníka meniť hodnotu nejakej premennej, je nevyhnutné pred priradením s využitím smerníku (riadok 4) nasmerovať smerník na príslušné miesto v pamäti (riadok 3), v ktorom chceme zmenu vykonať.

Nakoniec v riadku 5 sa používa symbol ***** trochu odlišným spôsobom ako v riadku 4. V riadku 4 zapisujeme hodnotu 5 na miesto, kde ukazuje smerník, naopak v riadku 5 len čítame hodnotu z pamäťového miesta, na ktoré ukazuje smerník **p** (ukazuje na premennú **x**) a túto hodnotu zvýšenú o 2 priradíme do premennej **y**. Po vykonaní priradenia v riadku 5 bude obsah pamäti takýto:

int x	<table border="1"><tr><td>5</td></tr></table>	5	140726106253484
5			
int y	<table border="1"><tr><td>7</td></tr></table>	7	140726106253480
7			
int *p	<table border="1"><tr><td>140726106253484</td></tr></table>	140726106253484	140726106253472
140726106253484			

Úloha 1-12

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int x;
2	int *p;
3	p = &x;
4	*p = 2;
5	x = *p + x;

int x	<input type="text"/>	140721498110700
int *p	<input type="text"/>	140721498110688

Úloha 1-13

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int a, b = 8;
2	int *p = &b;
3	a = 2**p + 1;

int a	<input type="text"/>	140725930423004
int b	<input type="text"/>	140725930423000
int *p	<input type="text"/>	140725930422992

Úloha 1-14

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

1	int i, j, *pi, *pj;
2	pi = &i;
3	*pi = 10;
4	*pi += 1;
5	(*pi)++;
6	j = *pi+1;
7	pj = pi;
8	*pj = *pj+j;
9	pj = &j;
10	*pj = *pi**pi;

int i	<input type="text"/>	140729974839292
int j	<input type="text"/>	140729974839288
int *pi	<input type="text"/>	140729974839280
int *pj	<input type="text"/>	140729974839272

Úloha 1-15

Analyzujte nasledujúci program. Obsahuje nejakú chybu? Ak áno, ako ju môžeme z programu odstrániť? Premyslite si, ako písať zdrojový kód tak, aby ste sa chybám tohto typu pri písaní programov vyhýbali.

1	int *p;
2	int a;
3	*p = 5;

Úloha 1-16

Doplňte chýbajúce časti nasledujúceho programu, ktorý pomocou smerníkov vymení hodnoty dvoch premenných a vypíše najskôr ich pôvodné a potom aj ich vymenené hodnoty na výstup podľa uvedenej ukážky vstupu a výstupu.

1	#include <stdio.h>
2	
3	int main(void)
4	{
5	int *p1, *p2, *p3, a, b;
6	scanf("%d %d", &a, &b);
7	p1 = <input type="text"/> ; p2 = <input type="text"/> .
8	printf("%d %d\n", *p1, *p2);
9	// presmerovanie smerníkov
10	p3 = <input type="text"/> ; p1 = <input type="text"/> ; p2 = <input type="text"/> .
11	printf("%d %d\n", *p1, *p2);
12	return 0;
13	}

Vstup:

1	10 45
---	-------

Výstup:

1	10 45
2	45 10
3	

Riešenia úloh (Kapitola 1)**Úloha 1-1 (riešenie)**

int x	<input type="text" value="8"/>	140732672863916
int y	<input type="text" value="8"/>	140732672863912
int z	<input type="text" value="16"/>	140732672863908

Úloha 1-2 (riešenie)

Premenná **y** nie je pred jej použitím v riadku 2 inicializovaná, a teda obsahuje neurčitú hodnotu, ktorej použitie (priradenie do **x**) spôsobí nedefinované správanie programu.

Zdrojový kód uvedený v úlohe je ukážkou neuváženého použitia neinicializovanej pamäte. Túto chybu môžeme odstrániť tak, že premennú **y** pred jej použitím v riadku 2 inicializujeme na nejakú hodnotu (napr. 0). Upravený program by potom vyzeral takto:

1	int x, y = 0;
2	x = y;

Úloha 1-3 (riešenie)

Priebežné úpravy hodnôt premenných uvádzame v komentároch na konci riadkov:

1	int x = 10, y = 5, z;	
2	++x;	// x = 11
3	x++;	// x = 12
4	y = ++x;	// x = 13, y = 13
5	y = x++;	// y = 13, x = 14
6	z = --y + x++;	// y = 12, z = 12+14, x = 15

int x	<table border="1"><tr><td>15</td></tr></table>	15	140734926495068
15			
int y	<table border="1"><tr><td>12</td></tr></table>	12	140734926495064
12			
int z	<table border="1"><tr><td>26</td></tr></table>	26	140734926495060
26			

V riadku 5 sa najprv priradí hodnota **x** (13) do premennej **y** a až potom sa zvýši hodnota **x** na 14. V riadku 6 sa najskôr zníži hodnota **y** na 12, potom sa vyhodnotí výraz na pravej strane priradenia do premennej **z** (12+14), a nakoniec sa zvýši hodnota premennej **x** na 15.

Úloha 1-4 (riešenie)

Priebežné úpravy hodnôt premenných uvádzame v komentároch na konci riadkov:

1	int x = 6, y = 4;	
2	double z = 2;	
3	y += x;	// y = 10
4	y *= x-3;	// y = 30
5	y /= ++x;	// x = 7, y = 4
6	x /= z;	// x = 3
7	z /= x;	// z = 0.666667
8	x %= y;	// x = 3

int x	<table border="1"><tr><td>3</td></tr></table>	3	140727123873612
3			
int y	<table border="1"><tr><td>4</td></tr></table>	4	140727123873608
4			
double z	<table border="1"><tr><td>0.666667</td></tr></table>	0.666667	140727123873600
0.666667			

V riadku 5 sa najskôr inkrementuje hodnota v premennej **x** a až potom sa ňou predelí hodnota v premennej **y**. Keďže premenná **y** je celé číslo (typu **int**), tak delenie je tzv. celočíselné delenie. Preto sa do premennej **y** priradí hodnota 7 (výsledok $30/4$ bez uvažovania desatinnej časti).

V riadku 7 prebieha bežné delenie desatinných čísel, pretože premenná na ľavej strane (**z**) je desatinné číslo (typu **double**). Premenná na pravej strane (**x**) je síce celé číslo (typ **int**) ale vzhľadom na typ premennej, do ktorej priraďujeme, sa premenná **x** pred delením implicitne pretypuje (zmení typ) na desatinné číslo (**double**), a do delenia potom vstupujú už dve desatinné čísla ($2.0/3.0 = 0.666667$).

V poslednom riadku 8 sa do premennej **x** priradí zvyšok po delení 3 číslom 4, a teda opätovne sa priradí tá istá hodnota (3). Pre iné počiatočné nastavenia hodnoty premennej **y** by sa mohla v tomto kroku priradiť iná hodnota.

Úloha 1-5 (riešenie)

Priebežné úpravy hodnôt premenných uvádzame v komentároch na konci riadkov:

1	<code>int v = 800;</code>	
2	<code>unsigned char a = 300, k = 220;</code>	
3	<code>v += k;</code>	<code>// v = 1020</code>
4	<code>k += v;</code>	<code>// k = 216</code>
5	<code>v = a + k;</code>	<code>// v = 260</code>

<code>int v</code>	260	140720552750924
<code>unsigned char a</code>	44	140720552750923
<code>unsigned char k</code>	216	140720552750922

V tejto úlohe je potrebné vnímať obmedzenie dátového typu **unsigned char**, ktorý má veľkosť 1 bajt (8 bitov), a preto dokáže reprezentovať nanajvýš 256 rôznych hodnôt: v tomto prípade čísla z množiny $\{0, \dots, 255\}$. Do premennej **a** teda nie je možné v riadku 2 inicializovať hodnotu 300 ale jej hodnota po inicializácii bude $300 \bmod 256 = 44$. V riadku 4 pripočítavame k aktuálnej hodnote premennej **k** (220) číslo 1020. Výsledný súčet (1240) však nie je možné v dátovom type **unsigned char** reprezentovať. Do premennej **k** sa zapíše hodnota najnižších 8 bitov súčtu: $1240 \bmod 256 = 216$. V riadku 5 priraďujeme do premennej **v** typu **int** (veľkosti 4 bajty), pričom premenné vo výraze na pravej strane priradenia sú typu **unsigned char**

(veľkosť 1 bajt). V rámci jedného príkazu sa premenné menšej veľkosti implicitne typovo prevedú na typ s najväčšou veľkosťou vo výraze (v tomto prípade `int`), a teda súčet `a+k` ($44+216 = 260$) bude bez straty informácie priradený do premennej `v`.

Úloha 1-6 (riešenie)

Priebežné úpravy hodnôt premenných uvádzame v komentároch na konci riadkov:

1	<code>int v = 120;</code>	
2	<code>double k = 400, s = 600;</code>	
3	<code>v += s/k;</code>	<code>// 121</code>
4	<code>k += v + v;</code>	<code>// 642.0</code>
5	<code>v += k/s;</code>	<code>// 122</code>

<code>int v</code>	122	140733686302204
<code>double k</code>	642.0	140733686302192
<code>double s</code>	600.0	140733686302184

V tejto úlohe jediná strata informácie nastáva na riadkoch 3 a 5, kde výsledok delenia je pripočítaný k premennej `v` bez desatinnej časti.

Úloha 1-7 (riešenie)

Explicitná typová konverzia (skrátенý program):

6	<code>double realne;</code>
7	<code>int cele;</code>
8	<code>scanf("%lf", &realne);</code>
10	<code>printf("%d\n", (int)realne);</code>

Vstup:

1	28.50
---	-------

Implicitná typová konverzia (skrátенý program):

6	<code>double realne;</code>
7	<code>int cele;</code>
8	<code>scanf("%lf", &realne);</code>
10	<code>printf("%d\n", cele = realne);</code>

Výstup:

1	28
2	

Riešenie využitím funkcie dolná celá časť `floor()` z `math.h` (skrátенý program):

6	<code>double realne;</code>
7	<code>int cele;</code>
8	<code>scanf("%lf", &realne);</code>
10	<code>printf("%.0f\n", floor(realne));</code>

[illegible]

Existujú aj iné funkcie v knižnici **math.h**, ktoré by ste mohli použiť.

Nakoniec preskúmajme, či je možné na výpis celočíselnej časti desatinného čísla použiť aj nasledujúci príkaz?

```
printf("%.01f\n", realne);
```

Uvedený príkaz zodpovedá výpisu vstupného desatinného čísla zaokrúhleného na 0 desatinných miest. Príkaz by teda mohol pracovať správne podľa požiadaviek úlohy. Vykonalme ho pre rôzne vstupy: pre vstup 28.1 vypíše 28, pre vstup 28.5 vypíše 28, pre vstup 28.9 vypíše 28. Ešte vyskúšajte, čo príkaz vypíše pre vstup 27.5, a podľa toho sa rozhodnite, či je vyššie uvedený príkaz správne riešenie tejto úlohy.

Úloha 1-8 (riešenie)

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     double v, m;
7     scanf("%lf %lf", &v, &m);
8     v /= 100.0;
9     printf("%.3lf\n", m / (v*v));
10    return 0;
11 }

```

 V_{stup}

1	170.0	58.8
---	-------	------

Výstup:

1	20.346
---	--------

Zo zadania úlohy vyplýva, že na vstupe môžu byť desatinné čísla, preto v definícii premenných v riadku 6 použijeme dátový typ **double**. Potom doplníme formátovací reťazec a adresy premenných pri načítaní v riadku 7. Riadok 8 obsahuje trochu

zvláštny kód: na ľavej strane premennú **v**, na pravej strane hodnotu 100.0. Všimnime si, že vzorec na výpočet BMI obsahuje výšku v metroch, ale na vstupe je uvedená v cm, preto riadok 8 využijeme na konverziu hodnoty v premennej **v** na správne jednotky. Následne doplníme formátovací reťazec pre výpis výsledku v riadku 9 (zadanie uvádza, že výsledok program vypíše zaokrúhlený na tri desatinné miesta). Nakoniec doplníme samotný výpočet indexu podľa vzorca uvedeného v zadaní úlohy.

Riešenie alternatívnej úlohy: Výpočet bude v tomto prípade prebiehať trochu inak: pre dané hodnoty BMI a hmotnosti by mal program určiť výšku v cm. Nasledujúci program predstavuje priamočiary výpočet z pôvodného vzorca:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      double bmi, m;
7      scanf("%lf %lf", &bmi, &m);
8      printf("%.11f\n", 100.0 * sqrt(m/bmi));
9      return 0;
10 }
```

Vstup:	
1	20.346 58.8

Výstup:	
1	170.0
2	

Tento program ľahko upravíme do formy pôvodnej doplňovačky tak, že premenujeme premennú **bmi** na **v**, a „zbavíme sa“ nepotrebného príkazu na riadku 8 napr. doplnením znamienka **+**, čo spôsobí, že výsledok aritmetického výrazu **v + 100.0** sa nepoužije, a neupraví sa hodnoty premenných v pamäti programu.

Výsledný doplnený program pre alternatívnu úlohu vyzerá takto:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      double v, m;
7      scanf("%lf %lf", &v, &m);
8      v + 100.0;
9      printf("%.11f\n", 100.0*sqrt(m/v));
10     return 0;
11 }
```

Vstup:	
1	20.346 58.8

Výstup:	
1	170.0
2	

Úloha 1-9 (riešenie)

Výstup programu pre vstup 0 1 2 2 4 je: 2 18 9 0 8. V prvom kroku sa vyhodnotí aritmetický výraz úplne vpravo: $a/2-5 = 2/2-5 = -4$. Pokračujeme sprava, teda od hodnoty v premennej **e** sa odpočíta hodnota výrazu (-4) , premenná **e** nadobudne hodnotu $4-(-4) = 8$. Táto hodnota (8) sa pripočíta k hodnote v premennej **c** (1), premenná **c** nadobudne hodnotu $(1+8=9)$. Nakoniec sa hodnota premennej **b** (2) vynásobí hodnotou premennej **c** (9) a premenná **b** nadobudne hodnotu $2*9 = 18$.

Riešenie alternatívnej úlohy: a) Výstup 4 3 2 1 0 nie je možné dosiahnuť s akýmkoľvek vstupom, pretože na výstupe musí byť hodnota **b** (1) násobkom hodnoty **c** (3), čo pre tento výstup neplatí.

b) Vstup, pre ktorý program vypíše na výstup 3 4 2 3 4, zistíme analýzou zdrojového kódu. Poznáme výstup (3 4 2 3 4), teda poznáme hodnoty premenných v okamihu výpisu v riadku 8. Hodnoty premenných **a** a **d** v programe neupravujeme, a teda s istotou môžeme povedať, že ich hodnoty na vstupe sú rovnaké ako ich hodnoty na výstupe, preto $a=3$ a $d=3$. Vieme teda určiť hodnotu výrazu $a/2-5 = -4$. Hodnota **e** na výstupe je 4, a túto hodnotu sme získali tak, že od hodnoty **e** na vstupe sme odpočítali hodnotu tohto výrazu (-4) , preto hodnota **e** na vstupe musela byť 0 (lebo $0 - (-4) = 4$). K hodnote premennej **c** na vstupe sme pripočítali už upravenú hodnotu premennej **e** (4), preto hodnotu **c** na vstupe získame odpočítaním 4 od hodnoty **c** na výstupe: $2-4 = -2$. Nakoniec, určíme hodnotu premennej **b** na vstupe: jej hodnotu na vstupe sme vynásobili hodnotou **c** na výstupe (2) a dostali sme 4, preto pôvodná hodnota **b** na vstupe je 2 (lebo $2*2 = 4$). Hodnoty premenných $a=3$, $b=2$, $c=-2$, $d=3$, $e=0$ zodpovedajú vstupu 3 -2 3 2 0.

Úloha 1-10 (riešenie)

Na vstupe máme 5 čísel, pričom každé z nich je celé číslo (typ `int`) alebo desatinné číslo (typ `double`). Pri načítaní nevieme vopred povedať, aké budú typy jednotlivých čísel, preto zvolíme pri načítaní takú reprezentáciu načítavaného čísla, ktorá nestratí žiadnu informáciu. Všimnime si, že celé číslo môžeme bez straty informácie načítať ako desatinné číslo, a teda pre načítanie vstupu podľa požiadaviek úlohy môžeme bez nejakej ujmy uvažovať, že na vstupe sú len desatinné čísla.

Uvažujme nasledujúcu postupnosť príkazov, v ktorej pomocou funkcie **scanf()** načítame zo vstupu 5 desatinných čísel a na výstup vypíšeme ich súčet:

1	double a, b, c, d, e;
2	scanf("%lf %lf %lf %lf %lf", &d, &c, &a, &b, &e);
3	printf("%lf\n", a+b+c+d+e);

Príkazy zrejme pracujú správne ak je na vstupe presne 5 čísel. Čo sa stane v prípade, ak je na vstupe čísel menej? Napr. v prípade, že na vstupe sú práve 4 čísla, tak funkcia **scanf()** nenaplní hodnotu do premennej **e**, ktorá je určená pre piate číslo. Hodnota premennej **e** bude neinicializovaná a pri použití neinicializovanej hodnoty pri výpočte súčtu v riadku 3 môže program vypísať nesprávny výsledok (iný výsledok ako súčet štyroch čísel na vstupe). Podobný problém nastáva vo všetkých prípadoch, v ktorých je počet čísel na vstupe menší ako 5, a to ten, že niektoré premenné nebudú načítané a teda ich hodnota zostane neinicializovaná, a jej použitie pre výpočet súčtu **a+b+c+d+e** môže spôsobiť nedefinované správanie programu.

Ak je na vstupe menej ako 5 čísel, potrebovali by sme, aby chýbajúce čísla neovplyvňovali súčet v riadku 3. To môžeme docieľiť napr. tak, že nastavíme počiatočnú hodnotu všetkých premenných na 0, a teda v prípade, že ich hodnotu **scanf()** nenačíta zo vstupu, zostane ich hodnota nastavená na 0, a súčet na riadku 3 správne určí súčet čísel na vstupe. Výsledný správny program:

1	#include <stdio.h>
2	
3	int main(void)
4	{
5	double a=0.0, b=0.0, c=0.0, d=0.0, e=0.0;
6	scanf("%lf %lf %lf %lf %lf", &d, &c, &a, &b, &e);
7	printf("%lf\n", a+b+c+d+e);
8	return 0;
9	}

Inicializáciu premenných na hodnotu 0 sme mohli docieľiť aj jednoduchšie (bez explicitného priradenia hodnoty 0.0 do každej premennej) a to tak, že by sme definíciu premenných presunuli pred funkciu **main()**, napr. na riadok 2, čím by sa stali globálne, ktoré sú pri spustení programu automaticky vynulované.

Úloha 1-11 (riešenie)

Doplnené časti programu, ktorý vypíše uvedený výstup:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i=42;
6      printf("<-%d>\n", i);
7      printf("<%-d>\n", i);
8      printf("<%-5d>\n", i);
9      printf("<%+05d>\n", i);
10     printf("<+%5d>\n", i);
11     printf("<+%0*d>\n", i/5, i);
12     return 0;
13 }
```

Výstup:

```

1  <-42>
2  <  42>
3  <42  >
4  <+0042>
5  <  +42>
6  <+000042>
7
```

Úloha 1-12 (riešenie)

```

int x  4  140721498110700
int *p 140721498110700 140721498110688
```

Úloha 1-13 (riešenie)

```

int a  17  140725930423004
int b   8  140725930423000
int *p 140725930423000 140725930422992
```

Možno nie úplne jasné je vyhodnotenie výrazu **2**p** v riadku 3, ktoré bez medzier môže vyzerat trochu zmätočne: dereferenčný operátor ***** (hviezdička) má vyššiu precendenciu (prioritu pri vyhodnocovaní vo výrazoch) ako operátor násobenia ***** (hviezdička), a teda najskôr sa zo smerníku **p** určí hodnota na ktorú ukazuje (8) a až potom sa táto hodnota vynásobí 2. Do premennej **a** sa preto priradí $2*8+1 = 17$.

Úloha 1-14 (riešenie)

Priebežné úpravy hodnôt premenných uvádzame v komentároch na konci riadkov:

1	int i, j, *pi, *pj;
2	pi = &i; // pi ukazuje na i
3	*pi = 10; // i = 10
4	*pi += 1; // i = 11
5	(*pi)++; // i = 12
6	j = *pi+1; // j = 13
7	pj = pi; // pj ukazuje na i
8	*pj = *pj+j; // i = 25
9	pj = &j; // pj ukazuje na j
10	*pj = *pi*pi; // j = 625

int i	25	140729974839292
int j	625	140729974839288
int *pi	140729974839292	140729974839280
int *pj	140729974839288	140729974839272

Úloha 1-15 (riešenie)

Pri vykonaní programu nastane chyba na riadku 3, v ktorom priradujeme hodnotu 5 do pamäte, na ktorú ukazuje smerník **p**. Hodnota smerníka **p** nie je inicializovaná a môže ukazovať na neznáme (resp. akékoľvek) miesto v pamäti. Priradenie hodnoty na neznáme miesto v pamäti môže spôsobiť nedefinované správanie: môže vyvolať neplatný prístup do pamäte a program predčasne ukončiť, alebo môže pokračovať s akousi „skrytou vadou“, ktorá sa môže nepredvídateľne prejaviť neskôr pri vykonávaní programu.

Systematický postup ako túto chybu odstrániť je dodržiavať pravidlo, že premenné pri definovaní (resp. pred prvým použitím) inicializujeme. V tomto prípade to môže byť na základnú, tzv. default, hodnotu pre dátový typ: **NULL** smerník – zodpovedá číselnej hodnote 0. Dostaneme teda nasledujúci program:

1	int *p = NULL;
2	int a;
3	*p = 5;

Pri vykonaní opäť nastane chyba na riadku 3, ale v porovnaní s predchádzajúcim prípadom, keď smerník **p** nebol inicializovaný, priradujeme hodnotu 5 už do konkrétneho miesta v pamäti, ktoré určite nie je v rozsahu pamäte programu a teda program zaručene predčasne skončí neplatným prístupom do pamäte. Vyhli

sme sa tým možnosti, že by program pokračoval ďalej, ale vykazoval by nedefinované správanie, pričom by sme nevedeli, či výsledok nakoniec bude správny!

Chybu úplne odstránime vtedy, keď pred každým priradením do smerníka bude príslušný smerník ukazovať na pamäť v rozsahu vykonávaného programu. V tomto prípade by mal ukazovať na premennú **a** napr. takto:

1	int a;
2	int *p = &a;
3	*p = 5;

Úloha 1-16 (riešenie)

Doplnené časti programu, ktorý pomocou smerníkov vymení hodnoty dvoch premenných a vypíše najskôr ich pôvodné a potom aj ich vymenené hodnoty na výstup podľa uvedenej ukážky vstupu a výstupu:

1	#include <stdio.h>
2	
3	int main(void)
4	{
5	int *p1, *p2, *p3, a, b;
6	scanf("%d %d", &a, &b);
7	p1 = &a; p2 = &b;
8	printf("%d %d\n", *p1, *p2);
9	// presmerovanie smerníkov
10	p3 = p1; p1 = p2; p2 = p3;
11	printf("%d %d\n", *p1, *p2);
12	return 0;
13	}

Vstup:

1	10 45
---	-------

Výstup:

1	10 45
2	45 10
3	

Kapitola 2

Riadiace štruktúry

2.1 Podmienky

Vykonanie programu vykoná príkazy programu postupne za sebou. Niekedy chceme v špecifických prípadoch podmieniť vykonávanie niektorých príkazov, napr.:

1	pocetDniRoku = 365;
2	if (priestupnyRok)
3	pocetDniRoku++;

Vykonanie tohto programu môže vykonať rôzne príkazy podľa aktuálneho stavu premenných. Najskôr (riadok 1) priradíme do premennej **pocetDniRoku** hodnotu 365. Potom (riadok 2) ak hodnota premennej **priestupnyRok** je true (nenulová), tak hodnotu v premennej **pocetDniRoku** zvýšime o 1. Naopak, ak hodnota premennej **priestupnyRok** je false (nulová), hodnotu v premennej **pocetDniRoku** nezvýšime.

Príkaz vetvenia sa skladá z kľúčového slova **if**, logického výrazu (podmienka), ktorá sa vyhodnotí, a z príkazov, ktoré sa vykonajú práve vtedy, keď podmienka je pravdivá. Základný príkaz vetvenia môžeme rozšíriť o vetvu **else** – príkazy, ktoré sa vykonajú keď podmienka nie je pravdivá:

```
if (podmienka)
    prikaz1;
else
    prikaz2;
```

Ak **podmienka** nadobúda nenulovú hodnotu (true) vykonáva sa **prikaz1**, inak **prikaz2**. V programe sa takto môžu vykonať rôzne inštrukcie podľa aktuálneho stavu hodnôt premenných. Viacero príkazov môžeme združiť do zloženého príkazu tak, že ich ohraničíme zloženými zátvorkami **{}**. Zložený príkaz, nazývaný aj blok, môže na začiatku obsahovať deklarácie premenných. Zložený príkaz, resp. blok, môžeme v zdrojovom kóde použiť tam, kde sa používa jednoduchý príkaz, a teda vetvy **prikaz1** a **prikaz2** môžu byť okrem jednoduchých aj zložené príkazy (bloky).

Akým spôsobom zapíšeme a vyhodnotíme logické výrazy v jazyku C?

Pomocou relačných operátorov $>$ (väčší), $>=$ (väčší alebo rovný), $<$ (menší), $<=$ (menší alebo rovný), $=$ (rovný), $!=$ (nerovný) sa porovnávajú dve hodnoty. Výsledok porovnania môže byť pravda (true), alebo nepravda (false). Týmto spôsobom vytvárame jednoduché logické výrazy. V prípade, že logický výraz vyhodnocuje hodnotu premennej (ako bolo uvedené vyššie v príklade s prestupným rokom), tak hodnota logického výrazu pre číslo 0 je nepravda (false) a pre akékoľvek nenulové číslo je hodnota logického výrazu pravda (true).

Podmienku môžeme použiť aj v (aritmetickom) výraze ako tzv. podmienený výraz, ktorý zapisujeme pomocou ternárneho operátora znakmi otáznik a dvojbodka nasledovne: **vysledok = podmienka ? vyraz1 : vyraz2**; Najskôr sa vyhodnotí logický výraz **podmienka**, ak je nenulový (true), tak sa vyhodnotí **vyraz1** a priradí sa do premennej **vysledok**, inak sa do premennej **vysledok** priradí **vyraz2**.

Zložené logické výrazy vznikajú spojením jednoduchých logických výrazov pomocou logických operátorov do jedného celku. Pre podporu základných logických operácií sa v zložených výrazoch používajú tieto operátory: logický súčin AND (**&&**), logický súčet OR (**||**) a negácia (**!**). Vyhodnocovanie výrazov sa vykonáva podľa nasledujúcej pravdivostnej tabuľky, kde 1 predstavuje hodnotu pravda (true) a 0 hodnotu false (nepravda):

p	q	p && q	p q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Zložené logické výrazy sú vyhodnocované v jazyku C zľava doprava, pričom sa používa **skrátene vyhodnocovanie výrazov** (angl. short-circuiting): program sa snaží vyhodnotiť logický výraz čo najefektívnejšie (vykonaním čo najmenšieho počtu operácií). Akonáhle môžeme určiť celkovú hodnotu logického výrazu vyhodnocovanie sa ukončí. Ak logický výraz obsahuje viacero operandov, môžeme použiť tieto dve pravidlá skrátene:

- výsledok operácie AND (**&&**) je 0 (false), ak aspoň jeden z operandov je 0,
- výsledok operácie OR je 1 (true), ak aspoň jeden z operandov je 1.

Úloha 2-1

Doplňte chýbajúce časti v nasledujúcom programe, ktorý načíta rok (celé číslo) zo vstupu a na výstup vypíše či je priestupný alebo nie. Rok je priestupný ak je deliteľný 4, ale roky deliteľné 100 nie sú priestupné okrem rokov deliteľných 400, ktoré sú priestupné.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int rok;
6      scanf("%d", &rok);
7      if ( ( ) && ( ) || ( ) )
8          printf("Rok %d je priestupny", rok);
9      else
10         printf("Rok %d nie je priestupny", rok);
11     return 0;
12 }
```

Úloha 2-2

Doplňte chýbajúce časti v nasledujúcom programe, ktorý načíta znak zo vstupu a na výstup vypíše či načítaný znak je číslica, písmeno alebo iný znak.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char c;
6      scanf("%c", );
7      if ( && )
8          printf("cislica\n");
9      else if ( || )
10         printf("pismeno\n");
11     else
12         printf("iny znak\n");
13     return 0;
14 }
```

Úloha 2-3

Doplňte chýbajúce časti v nasledujúcom programe, ktorý načíta dve celé čísla zo vstupu a na výstup vypíše párne čísla zo vstupu oddelené čiarkou. Ak boli obe čísla na vstupe párne, vypíše ich v rovnakom poradí, navyše oddelené čiarkou. Ak bolo z čísel na vstupe len jedno párne, vypíše len to konkrétne párne číslo.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int x, y, ciarka = 0;
6      scanf("%d %d", &x, &y);
7      if ( )
8      {
9          printf("%d", x);
10         ciarka = ;
11     }
12     if (( ))
13     {
14         if ( )
15             printf(",");
16         printf("%d", y);
17         ciarka = ;
18     }
19     return 0;
20 }

```

Úloha 2-4

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

```

1  int i = 1, j = 1, k = 1;
2  if (i++ && !j || k++)
3      j++;
4  if (i++ || j && !i || k++)
5      j++;

```

int i	<input type="text"/>	140736829153164
int j	<input type="text"/>	140736829153160
int k	<input type="text"/>	140736829153156

Úloha 2-5

Určite hodnoty premenných po vykonaní nasledujúcich príkazov.

```

1  int i = -1, j = 1, k = 1;
2  if (i+2 && j++ && k--)
3      j++;
4  if (!i && j++ || !j && i-- || k++)
5      j++;

```

int i	<input type="text"/>	140736139359756
int j	<input type="text"/>	140736139359752
int k	<input type="text"/>	140736139359748

2.2 Cykly

Zatiaľ sme v programoch používali len príkazy, ktoré sa vykonajú v programe raz a vykonávanie pokračuje nasledujúcim príkazom. V praxi však často potrebujeme prehľadne zapísať príkazy, ktoré sa vykonajú opakovane.

V jazyku C existujú tri typy cyklov: **for**, **while** a **do-while**. Cyklus vhodný pre vopred daný počet opakovaní (napr. príkazy chceme opakovať 20-krát, alebo **x**-krát, kde **x** je premenná) – zvyčajne použijeme cyklus **for**.

Všeobecný formát cyklu **for** je nasledujúci:

```
for (inicializacia; podmienka; krok)
    prikaz; alebo { blok prikazov }
```

Príkaz **inicializacia** (napr. inicializácia riadiacej premennej) sa vykoná iba raz na začiatku vykonávania **for** cyklu. Vyhodnotenie a testovanie logického výrazu **podmienka** sa vykoná vždy pred vykonaním tzv. tela cyklu. Telo cyklu tvorí alebo jeden príkaz ukončený bodkočiarkou, alebo blok (príkazy ohraničené zloženými zátvorkami). Príkazy sú vykonané iba ak je testovaná **podmienka** splnená. Po vykonaní príkazov sa vykoná príkaz **krok**, napr. inkrementovanie riadiacej premennej. Cieľom príkazu **krok** je posunúť sa na ďalší krok v spracovaní. Ak **podmienka** nie je splnená, vykonávanie cyklu sa skončí a vykonávanie programu pokračuje príkazom uvedeným tesne za cyklom. Výrazy **inicializacia**, **podmienka** aj **krok** môžu byť prázdne.

V prípade, že telo cyklu nevyžaduje prvotnú inicializáciu (príkaz **inicializacia**) ani posun (príkaz **krok**) môžeme použiť niektorý z dvoch variantov **while** cyklu, ktorý okrem tela cyklu vyžaduje len podmienku určujúcu, či sa má vykonať ďalšia iterácia (tela) cyklu:

- a) `while (podmienka) prikaz; alebo { blok prikazov; }`
- b) `do prikaz; alebo { blok prikazov; } while (podmienka);`

Telo cyklu sa bude vykonávať iba v prípade ak **podmienka** je splnená (true). To znamená, že telo cyklu **while** (a) nemusí byť vykonané ani raz, ak je podmienka nespĺnená už na začiatku. V prípade **do-while** (b) sa platnosť podmienky overuje až po vykonaní tela cyklu, a preto telo cyklu **do-while** je vykonané vždy aspoň raz.

Úloha 2-6

Určite výstup vykonania nasledujúceho programu.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, pocet = 0;
6
7      for (i = 20; pocet < 10; i++)
8      {
9          if (i % 3)
10             continue;
11             printf("%d\n", i);
12             pocet++;
13             if (i > 40)
14                 break;
15         }
16     return 0;
17 }
```

Úloha 2-7

Napište program, ktorý pre celé číslo N zadané na vstupe vypíše všetky nepárne čísla od 1 do N. Program by mal pracovať správne pre všetky prípustné hodnoty pre N. Pri overovaní správnosti riešenia experimentujte s rôznymi hodnotami pre N.

Úloha 2-8

Doplňte chýbajúce časti v nasledujúcom programe, ktorý načíta celé číslo N zo vstupu a na výstup vypíše hodnotu N! (faktoriál). Predpokladajte, že $N < 20$.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      _____ i, n, f = _____;
6      scanf("_____", &n);
7      for (i = _____; _____; i++)
8          f *= _____;
9
10     printf("_____\n", f);
11     return 0;
12 }
```

Úloha 2-9

Na vstupe je celé číslo N. Preskúmajte nasledujúci program, zistite načo slúži a určite výstup programu pre vstup: 12345

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 0, n;
6      scanf("%d", &n);
7      do
8      {
9          i = 10*i + n%10;
10     } while (n /= 10);
11
12     printf("%d\n", i);
13     return 0;
14 }
```

Úloha 2-10

Na vstupe je niekoľko (nevieme koľko) celých čísel oddelených medzerami, zaujíma nás, koľko z nich je párnych. Doplňte chýbajúce časti v nasledujúcom programe, ktorý na výstup vypíše počet párnych čísel na vstupe.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, j = █;
6
7      while (scanf("%d", █) == █)
8          █ += █;
9
10     printf("%d\n", █);
11     return 0;
12 }
```

Úloha 2-11

Na vstupe je číslo K a niekoľko (nevieme vopred koľko) ďalších celých čísel. Zaujíma nás, ktoré z týchto čísel sú násobky čísla K. Analyzujte nasledujúci program, ktorý zo vstupu načíta K a potom aj zostávajúce čísla, a na výstup vypíše tie, ktoré sú násobky čísla K. Predstavuje nasledujúci program správne riešenie tejto úlohy? Ak

áno, zdôvodnite prečo. Ak nie, vysvetlite v čom je chybný, navrhните ako chybu odstrániť a premyslite stratégiu ako týmto chybám predchádzať.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, k;
6      scanf("%d", &k);
7      while (scanf("%d", &i) == 1)
8          if (i % k == 0)
9              printf("%d\n", i);
10
11     return 0;
12 }

```

Vstup:

1	3
2	5 6 7 8 9 10

Výstup:

1	6
2	9
3	

Úloha 2-12

Daná je booleovská (logická) funkcia 3 premenných, označme ju **f**. Napíšte program, ktorý vypíše pravdivostnú tabuľku pre túto funkciu.

Uvažujme napr. funkciu **f(a,b,c) = (a+b+c > 1)**, výstup programu je uvedený vpravo.

1	a b c -> f(a,b,c)
2	-----
3	0 0 0 -> 0
4	0 0 1 -> 0
5	0 1 0 -> 0
6	0 1 1 -> 1
7	1 0 0 -> 0
8	1 0 1 -> 1
9	1 1 0 -> 1
10	1 1 1 -> 1

Úloha 2-13

Chceli by sme vytvoriť tzv. nekonečný cyklus, teda taký príkaz cyklu, ktorý v bežnom prípade neskončí, resp. program (v žiadnom prípade) nebude pokračovať na príkaz za takýmto cyklom. Analyzujte nasledujúce kódy a zistite, ktoré z nich predstavujú nekonečný cyklus. Skúste vymyslieť aj vlastné spôsoby. Premenná **i** je typu **unsigned int**. Bodkočiarka za cyklom predstavuje prázdne telo cyklu.

- for (i = 1; i > 0; i++);
- for (i = 1; i < i+1; i++);
- for (i = 1; i != 0; i++);
- for (i = 1; i = i; i++);
- { i=1; while(i++); }
- { i=1; while(++i > 0); }

Alternatívna úloha: Zamyslite sa, čo by sa zmenilo, ak by premenná **i** bola typu **int**.

Úloha 2-14

Analyzujte nasledujúci program. Čo vypíše na výstup, ak na vstupe sú čísla: 7 3?

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n, i, j;
6      while (scanf("%d", &n) == 1)
7          for (i = 0; i < n; i++)
8              {
9                  for (j = 0; j < n; j++)
10                     if (i == 0 || j == 0 || i == n-1 || j == n-1 || i == j)
11                         putchar('#');
12                     else
13                         putchar(i > j ? '.' : ' ');
14                 printf("\n");
15             }
16     return 0;
17 }
```

Úloha 2-15

Doplňte chýbajúce časti v nasledujúcom programe, ktorý vykreslí na obrazovku obrazec zo znakov #, bodka a medzera podľa uvedenej ukážky.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      _____;
6      while (scanf("%d", &n) == 1)
7          for (i = 0; i < _____; i++)
8              {
9                  for (j = 0; j < _____; j++)
10                     if (_____ || _____)
11                         putchar('#');
12                     else if (_____ || _____)
13                         putchar('o');
14                     else
15                         putchar(' ');
16                 printf("\n");
17             }
18     return 0;
19 }
```

Vstup:

1	3
---	---

Výstup:

1	ooo#...
2	ooo#...
3	ooo#...
4	#####
5	...#ooo
6	...#ooo
7	...#ooo
8	

Úloha 2-16

Na vstupe sú tri kladné celé čísla: veľkosť obrazca N, počet opakovaní pod seba R a za sebou S.

Napište program, ktorý pre čísla na vstupe vypíše na výstup základný vzor číselnej pyramídy veľkosti N opakujúci sa R-krát pod sebou a S-krát za sebou podľa uvedenej ukážky.

Najskôr vytvorte program, ktorý vypíše jeden základný vzor veľkosti N. Potom program rozšírite o opakovanie základného vzoru za sebou a potom aj o opakovanie pod seba.

Vstup:

1	3 3 2
---	-------

Výstup:

1	1	1
2	222	222
3	3333333333	
4	1	1
5	222	222
6	3333333333	
7	1	1
8	222	222
9	3333333333	
10		

Riešenia úloh (Kapitola 2)**Úloha 2-1 (riešenie)**

1	#include <stdio.h>
2	
3	int main(void)
4	{
5	int rok;
6	scanf("%d", &rok);
7	if (((rok%4 == 0) && (rok%100 != 0)) (rok%400 == 0))
8	printf("Rok %d je priestupny", rok);
9	else
10	printf("Rok %d nie je priestupny", rok);
11	return 0;
12	}

Úloha 2-2 (riešenie)

ASCII hodnoty znakov číslíc tvoria interval: '0' (48), '1' (49), ..., '9' (57), preto v podmienke môžeme použiť operácie >= a <=. Podobne pri znakoch: 'a' (97), ..., 'z' (122) a 'A' (65), ..., 'Z' (90). Doplnený program je nasledovný:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c;
6     scanf("%c", &c);
7     if ( c >= '0' && c <= '9' )
8         printf("cislica\n");
9     else if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
10        printf("pismeno\n");
11     else
12        printf("iny znak\n");
13     return 0;
14 }
```

Úloha 2-3 (riešenie)

Číslo zo vstupu načítame do premenných **x** a **y**. Číslo **x** vypíšeme práve vtedy, keď podmienka v riadku 7 je splnená, preto doplníme **x%2 == 0**. Podobne, číslo **y** vypíšeme práve vtedy, keď podmienka v riadku 12 je splnená: doplníme **y%2 == 0**. Zostáva nám doriešiť správne vypisovanie čiarky.

Čiarku vypíšeme práve vtedy, keď vypíšeme obe čísla, teda keď v riadku 14 platí okrem **y%2 == 0** aj **x%2 == 0**. V programe na doplnenie však máme ešte ďalšiu premennú (**ciarka**), ktorú by sme mohli na tento účel použiť. Do premennej **ciarka** si uložíme informáciu, či pred najbližšom výpisom čísla musíme vypísať ešte čiarku. Premenná **ciarka** je inicializovaná na 0 (čiarku nemusíme vypísať) a vždy po vypísaní čísla (riadky 10 a 17) nastavíme premennú **ciarka** na hodnotu 1 (čiarku musíme vypísať pred výpisom ďalšieho čísla). Podmienka 14 bude potom triviálna (**ciarka**).

Výsledný doplnený program vyzerá takto:

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int x, y, ciarka = 0;
6      scanf("%d %d", &x, &y);
7      if ( x % 2 == 0 )
8      {
9          printf("%d", x);
10         ciarka = 1;
11     }
12     if ( y % 2 == 0 )
13     {
14         if ( ciarka )
15             printf(",");
16         printf("%d", y);
17         ciarka = 1;
18     }
19     return 0;
20 }

```

Vysvetlenie: Načo je dobrá takáto implementácia pomocou dodatočnej premennej? Hlavná výhoda spočíva v tom, že to umožňuje rozhodnutie vypísať/nevypísať čiarku (riadok 14) urobiť bez skúmania konkrétnych hodnôt vstupných čísel. Stačí zistiť hodnotu (stav) premennej **ciarka**. V prípade, že by sme mali takto vypísať 3 alebo viac čísel, s využitím takejto stavovej premennej to môžeme spraviť bez väčších komplikácií v kóde. Naopak bez použitia stavovej premennej by takýto výpis pri troch alebo viac premenných bol už značne komplikovaný: vyskúšajte si to!

Úloha 2-4 (riešenie)

int i	3	140736829153164
int j	3	140736829153160
int k	2	140736829153156

Úpravy hodnôt sú vyznačené v komentároch na konci riadkov:

1	int i = 1, j = 1, k = 1;	
2	if (i++ && !j k++)	// i=2, k=2
3	j++;	// j=2
4	if (i++ j && !i k++)	// i=3
5	j++;	// j=3

V riadku 4 premenná **k** nezmení hodnotu, pretože hodnota **i** pred inkrementovaným je nenulová.

Úloha 2-5 (riešenie)

int i	-1	140736139359756
int j	3	140736139359752
int k	1	140736139359748

Úpravy hodnôt sú vyznačené v komentároch na konci riadkov:

1	int i = -1, j = 1, k = 1;	
2	if (i+2 && j++ && k--)	// j=2, k=0
3	j++;	// j=3
4	if (!i && j++ !j && i-- k++)	// k=1
5	j++;	

Úloha 2-6 (riešenie)

Hlavnou časťou program je cyklus **for**, všimnime si podmienku (**pocet < 10**), ktorá určuje či sa vykoná telo cyklu: telo cyklu (riadky 8-15) sa vykoná ak aktuálna hodnota v premennej **pocet** bude menšia ako 10.

Čo v programe predstavuje premenná **pocet**? Premennú **pocet** zvyšujeme tesne potom ako v riadku 11 vypíšeme hodnotu premennej **i** na výstup, teda premenná **pocet** symbolizuje počet čísel, ktoré sme už vypísali na výstup. Telo cyklu opakujeme dovtedy, kým sme na výstup vypísali zatiaľ menej čísel ako 10. Premennú **pocet** správne inicializujeme na hodnotu 0 hneď pri definícii v riadku 5, čo zodpovedá skutočnosti, že zatiaľ sme na výstup nevypisovali žiadne čísla.

Všimnime si, čo vypisujeme na výstup: vypisujeme hodnotu premennej **i**. Analyzujeme hodnoty, ktoré bude premenná **i** počas vykonávania príkazu **printf()** v riadku 11 nadobúdať. Tesne pred výpisom sa nachádza v riadkoch 9-10 podmienka **if**. Logický výraz **i%3** (zvyšok po delení hodnoty **i** číslom 3) sa vyhodnotí ako pravda (true) práve vtedy, ak je zvyšok nenulový a vtedy sa vykoná telo podmienky: príkaz **continue**, ktorý ukončí vykonávanie aktuálnej iterácie tela cyklu a pokračuje vo vykonávaní cyklu skokom na príkaz krok-u v riadku 7 (**i++**). Teda výpis na riadku 11 sa vykoná len pre také hodnoty **i**, ktoré sú násobky čísla 3.

Zostáva určiť, pre ktoré hodnoty **i** sa bude telo cyklu vykonávať. Inicializácia cyklu nastaví počiatočnú hodnotu **i** na 20, krok cyklu ju po jednom zvyšuje. Dokedy ju budeme zvyšovať? Odpoveď je uvedená na riadku 13, teda potom ako hodnotu **i** deliteľnú 3 vypíšeme skontrolujeme či sme vypísali hodnotu väčšiu ako 40, ak áno

príkazom **break** ukončíme vykonávanie cyklu. Cyklus teda ukončíme nielen pri nesplnení podmienky (**pocet < 10**) v riadku 7 ale aj pri prekročení hodnoty vypísaného čísla (**i > 40**) v riadku 14.

Program preto na vstup vypíše čísla: 21, 24, 27, 30, 33, 36, 39, 42, každé do samostatného riadku a skončí. Program vypísal 8 čísel a preto bol cyklus ukončený príkazom **break** v riadku 14. V prípade, že by sme podmienku v riadku 13 upravili na (**i > 50**), pokračovali by sme výpisom čísel 45 a 48, pričom by premenná **pocet** dosiahla hodnotu 10 a pri vyhodnotení podmienky cyklu (**pocet < 10**) by sme zistili, že neplatí, a teda cyklus sa ukončí.

Úloha 2-7 (riešenie)

Uvažujme nasledovný program ako riešenie tejto úlohy:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, n;
6      scanf("%d", &n);
7      for (i = 1; i <= n; i++)
8          if (i % 2)
9              printf("%d\n", i);
10     return 0;
11 }
```

Program načíta celé číslo **N** zo vstupu a v cykle (riadky 7-9) pre premennú **i** prechádza hodnoty **i = 1, 2, ..., N**. Ak je zvyšok po delení hodnoty **i** číslom 2 je nenulový (podmienka v riadku 8 sa vyhodnotí ako splnená), tak sa vypíše hodnota **i** na výstup. Pre kladné hodnoty **N** sa telo cyklu určite vykoná aspoň raz. Napr. pre **N = 10** sa vypíšu čísla **1, 3, 5, 7, 9** (každé na samostatnom riadku).

Čo však v prípade ak je celé číslo **N** na vstupe záporné? Napr. **N = -5**, program by mal vypísať nepárne čísla od **1** do **-5**. Môžeme teraz začať polemizovať o presnom významne slov „od“ a „do“ ale v širšom zmysle sa od čísla **1** postupných prechodom ku číslu **-5** po číselnej osi nachádzajú celé čísla **0, -1, -2, -3, -4, -5**. Z nich sú **-1, -3, -5** nepárne a program by ich mal vypísať. V našom programe však pre **N < 1** nie je splnená podmienka **i < N** ani raz, pretože premenná **i** je inicializovaná

na počiatočnú hodnotu **1**. Cyklus musíme preto upraviť tak, aby prechádzal čísla od **1** smerom k **N** bez ohľadu nato, či smer je doprava (krok je +1) alebo doľava (krok -1) na číselnej osi. Podmienku preto upravíme tak, aby sa telo cyklu opakovalo pokým premenná **i** nedosiahne hodnotu **N** (podmienka **i != N**). Podobne krok upravíme tak, že podľa vzájomného usporiadania **1** a **N** sa posúvame v tom správnom smere. Smer posunu si označíme **d**. Výsledný program vhodný aj pre záporné hodnoty **N**:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, n, d = 1;
6      scanf("%d", &n);
7      if (n < 1) d = -1;
8      for (i = 1; i != n+d; i+=d)
9          if (i % 2)
10             printf("%d\n", i);
11     return 0;
12 }
```

Premenná **d** reprezentuje smer posúvania. Inicializujeme na 1 (očakávame smer posunu +1). V riadku 7 v prípade, že načítané číslo **N** je menšie ako 1 tak zmeníme smer posunu na -1. Všimnime si upravený cyklus (riadok 8): príkaz kroku zodpovedá posunu hodnoty premennej **i** v smere posunu **d** (+1 alebo -1), a podmienka (určujúca či bude cyklus pokračovať alebo bude ukončený nie je (**i != N**) ako sme pôvodne zamýšľali preto, lebo cyklus by v prípade nepárneho **N** skončil predtým ako vypíše hodnotu **N** na výstup. Podmienka (**i != N+d**) zaručí, že cyklus sa pre nepárne **N** vykoná a po nasledujúcom posune (na číslo **N+d**) bude cyklus ukončený, pretože prestane platiť podmienka (**i != N+d**).

Úloha 2-8 (riešenie)

Pre číslo **N** je faktoriál **N!** definovaný podľa týchto dvoch pravidiel:

- ak **N = 0** platí: **N! = 1**,
- ak **N > 0** platí: **N! = N*(N-1)!**

Pre **N = 0** by teda mal program vypísať číslo 1 a pre **N > 0** by sme mali využiť hodnotu **(N-1)!** V riadku 10 vypisujeme hodnotu premennej **f** na výstup, z čoho môžeme

usudzovať, že premenná **f** v programe reprezentuje priebežne počítanú hodnotu faktoriálu. V riadku 8 hodnotu premennej **f** násobíme opakovane (v tele cyklu) nejakým neznámym výrazom. Čo doplníme v riadku 8? Vzhľadom na postup výpočtu faktoriálu musíme hodnotu premennej **f** v riadku 8 násobiť hodnotou premennej **i**.

Keďže hodnotu **f** upravujeme len v riadku 5 (inicializácia) a v riadku 8 (násobenie hodnotou), musíme hodnotu **f** inicializovať na hodnotu **1**, ktorá zodpovedá hodnote 0! Následne určíme hranice cyklu: pre hodnotu **N** potrebujeme **f** v poslednej iterácii cyklu vynásobiť **N**, teda podmienka bude **i <= N**. Zčať môžeme od hodnoty **i = 1**, resp. od **i = 2**, lebo násobenie **f** má zmysel až od hodnoty **i = 2**.

Zostáva nám určiť aký dátový typ použijeme pre reprezentáciu čísel. Hodnota 20! je 2 432 902 008 176 640 000, pričom najväčší bežne používaný celočíselný dátový typ (**unsigned long**) dokáže reprezentovať najviac číslo 18 446 744 073 709 551 615, čo je pre nás postačujúce.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      unsigned long i, n, f = 1;
6      scanf("%lu", &n);
7      for (i = 1; i <= n; i++)
8          f *= i;
9
10     printf("%lu\n", f);
11     return 0;
12 }
```

Praktický tip: Najväčšiu hodnotu, ktorú dokáže **unsigned long** reprezentovať zistíme nasledovným príkazom: `printf("%lu\n", (unsigned long)-1);` Zamyslite sa nad tým, prečo to funguje. Pri všetkých dátových typoch bez znamienka (**unsigned**) platí, že keď najväčšie číslo, ktoré dokážu reprezentovať zvýšime o 1, tak sa dostaneme naspäť na 0 (najmenšie číslo). V našom prípade to využijeme „v opačnom smere“, teda zoberieme najmenšie číslo, ktoré dokážu reprezentovať (0), odpočítame 1, čím sa dostaneme k najväčšiemu číslu.

Dôležité upozornenie: V prípade typov so znamienkom (**signed**) je takéto pretečenie (najväčšia hodnota + 1) alebo podtečenie (najmenšia hodnota - 1)

nebezpečné a spôsobí nedefinované správanie programu. Dôvod je ten, že presné správanie závisí od spôsobu implementácie bitovej reprezentácie premenných v kompilátore. V praxi môže (ale nemusí!) vykonávanie pokračovať bez vážnejšej ujmy, ale z formálneho hľadiska sa na výsledky programu pri pretečení/podtečení znamienkového (**signed**) typu už nedá spoľahnúť.

Úloha 2-9 (riešenie)

Po načítaní čísla v riadku 6 sa v programe vykonáva cyklus **do-while** (riadky 7-10). Potrebujeme zodpovedať na dve otázky:

1. **Koľkokrát sa bude cyklus opakovať?** Podmienka (riadok 10) obsahuje operátor \neq , ktorý v tomto prípade zodpovedá celočíselnému deleniu 10 a priradeniu výsledku do premennej **n**. V programe premennú **n** (okrem načítania v riadku 6) na inom mieste neupravujeme. Podmienka **n** \neq 10 je vyhodnotená ako splnená ak hodnota premennej **n** zostane po delení nenulová. Uvažujme teraz vstupné číslo **n** = 12345, výsledok **n** \neq 10 je 1234, čo zodpovedá odstráneniu poslednej cifry z čísla. Cyklus sa opakuje pokým podmienka je splnená, teda pokým hodnota premennej **n** je nenulová. Počet opakovaní cyklu preto zodpovedá počtu cifier čísla **n**.
2. **K čomu slúži výpočet v tele cyklu?** Uvažujme číslo **n** = 12345, priradenie v riadku 9, prečíta poslednú cifru **n** a upraví hodnotu premennej **i**. V prvej iterácii tela cyklu nastavíme poslednú cifru vstupného čísla **n** ako prvú cifru čísla **i**. V každej ďalšej iterácii priradenie **i** = 10*i posunie aktuálnu hodnotu v premennej **i** o jeden desiatkový rád doľava, teda poradie cifier zľava doprava zostane v premennej **i** zachované: napr. pre **i** = 54 je 10*i = 540. Pripočítaním poslednej cifry (**n**%10) k tejto hodnote potom prenesieme poslednú cifru z premennej **n** ako novú poslednú cifru premennej **i**.

Telo cyklu teda postupne od najnižších rádov presúva cifry zo vstupného čísla **n** do premennej **i**, čím ich otočí: pre vstupné číslo 12345 je výstup programu 54321.

Úloha 2-10 (riešenie)

V riadku 7 načítavame vstup a potrebujeme sa rozhodnúť, do ktorej z premenných (**i** alebo **j**) číslo zo vstupu načítame. Ak by sme ho načítali do premennej **j**, tak by inicializácia jej hodnoty v riadku 5 nemala zmysel a zostala by nám v programe neinicializovaná premenná **i**. Zrejme teda, načítavať budeme do premennej **i**. Cyklus **while** (riadky 7 až 8) by sa mal opakovať pre každé číslo, ktoré je na vstupe. Pripomeňme si, že funkcia **scanf()** vracia počet úspešne načítaných prvkov, teda v prípade, že sa zo vstupu podarilo načítať ďalšie číslo tak funkcia vráti 1.

Premenná **j** bude slúžiť ako počítadlo párnych čísel, teda najskôr premennú **j** inicializujeme v riadku 5 na hodnotu 0 (nenačítali sme párne čísla), v riadku 10 výslednú hodnotu premennej **j** vypíšeme a v riadku 8 do nej pripočítame 1 v prípade, že hodnota v premennej **i** je párne číslo. Potrebujeme teda ešte vymyslieť výraz, ktorý sa v prípade, že **i** je párne číslo vyhodnotí ako číslo 1 a ako 0 v prípade, že je nepárne.

Výsledné doplnenie programu:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, j = 0;
6
7      while (scanf("%d", &i) == 1)
8          j += 1-(i%2);
9
10     printf("%d\n", j);
11     return 0;
12 }
```

Úloha 2-11 (riešenie)

Uvedený program nesprávne pracuje v prípade ak má **k** hodnotu 0. Chyba sa pri vykonávaní vyskytne na riadku 8, v ktorom načítané číslo (**i**) delíme číslom **k** a ak je zvyšok po delení 0 (**i % k == 0**), tak vypíšeme **i** ako násobok čísla **k**.

Chybám tohto typu sa môžeme vyhnúť tak, že pred každým vykonaním operácie delenia (operácia **/** a aj zvyšok po delení **%**) overíme, že deliteľ je nenulový. Skúsme teda odstrániť chybu v riadku 8 nasledovným príkazom:

```
if (k != 0 && i % k == 0)
```

Touto úpravou sme odstránili prípadne delenie nulou, ktoré spôsobuje predčasné ukončenie vykonávania programu. Nedoriešili sme však ešte prípad ak k má hodnotu 0. Ktoré čísla sú potom násobky K ? Čísla s hodnotou 0 sú násobky $K = 0$. Správna podmienka v riadku 8, ktorá zahŕňa aj prípad pre $K = 0$, je teda takáto:

```
if ((k == 0 && i == 0) || (k != 0 && i % k == 0))
```

Úloha 2-12 (riešenie)

Booleovská funkcia ($f()$) môže byť akákoľvek, našou úlohou je do funkcie $f()$ postupne dosadiť všetky možné kombinácie hodnôt booleovských (0/1) premenných a prehľadne do tabuľky vypísať získané hodnoty.

Všetky možné kombinácie hodnôt troch booleovských premenných môžeme systematicky prejsť využitím troch vnorených cyklov. Výsledný program:

```

1  #include <stdio.h>
2
3  int f(int a, int b, int c)
4  {
5      return (a+b+c) > 1;
6  }
7  int main(void)
8  {
9      int a, b, c;
10     printf("a b c -> f(a,b,c)\n");
11     printf("-----\n");
12     for (a = 0; a <= 1; a++)
13         for (b = 0; b <= 1; b++)
14             for (c = 0; c <= 1; c++)
15                 printf("%d %d %d -> %d\n", a, b, c, f(a,b,c));
16     return 0;
17 }
```

Úloha 2-13 (riešenie)

Všetky uvedené výrazy na prvý pohľad vyzerajú ako nekonečné cykly. Skúste si ich ale spustiť a po nejakom čase (v závislosti od rýchlosti počítača) všetky z nich skončia.

Všetky prípady sú analogické. Pri inicializácii nastavíme hodnotu i na 1, pričom krok cyklu predstavuje zvýšenie hodnoty o 1, preto zdanlivo podmienka cyklu bude vždy splnená. Problém nastane pri obmedzení dátového typu. V prípade typu `unsigned int` je najväčšie reprezentovateľné číslo 4 294 967 295 a zvýšenie tejto

hodnoty o 1 spôsobí pretečenie, výsledok ktorého je hodnota 0, ktorá však poruší podmienku cyklu a teda cyklus skončí.

Praktický tip: Bežne sa ako cykly opakujúce sa donekonečna používajú príkazy `for(;;)` a `while(1)`, ktorých podmienky budú vždy vyhodnotené ako platné. Nemusí sa však jednať o chybu v programe. Ak sa takýto zdanlivo „nekonečný“ cyklus v programe nachádza, tak je to zvyčajne úmyselné. Takýto cyklus možno ukončiť volaním príkazu `break` alebo `return` z tela cyklu, a teda cyklus v konečnom dôsledku skončí, aj keď jeho podmienka je a bude vždy platná.

Alternatívna úloha: Situácia by bola dramaticky (!) iná ak by sme uvažovali túto úlohu pre premennú typu `int` resp. iného znamienkového (`signed`) typu ako napr. `char` alebo `long`. Ako sme už spomínali v riešení úlohy 2-8: v prípade znamienkových typov nie je totiž v jazyku C formálne presne definované čo sa stane pri pretečení a môže sa to líšiť v závislosti od spôsobu bitovej reprezentácie príslušného typu premennej v kompilátore. Vo vykonávanom programe to preto spôsobí nedefinované správanie, následkom čoho program ďalej nemusí pracovať spoľahlivo a výsledok môže byť prakticky akýkoľvek.

V praxi sa pretečenie premennej typu `int` prejaví tak, že zvýšením najväčšej reprezentovateľnej hodnoty 2147483647 o 1 dostaneme najmenšiu reprezentovateľnú hodnotu -2147483648, čo v prípadoch a), b) a f) spôsobí porušenie podmienky a ukončenie cyklu. V prípadoch c), d) a e) pokračujeme vo zvyšovaní aj pre zápornú hodnotu premennej `i`, avšak po konečnom počte krokov sa dostaneme až k hodnote 0, podmienka prestane platiť a cykly skončia.

Úloha 2-14 (riešenie)

Načítanie čísel zo vstupu sa vykonáva v riadku 6, každé číslo je spracované samostatne a výpočet pre rôzne čísla sa neovplyvňuje. Pre každé číslo na vstupe (`n`) v riadkoch 7-15 sa vykoná dvojitý vnorený cyklus. Vonkajší cyklus pre premennú `i` sa opakuje `n`-krát. Telo vonkajšieho cyklu obsahuje vnútorný cyklus, ktorý vypisuje znaky (#, bodka a medzera) a nakoniec (riadok 14) sa vypíše nový riadok.

Pre vstupné číslo n bude teda výstup obsahovať n riadkov. Na i -tom z riadkov vypíše vnorený cyklus (riadky 9-13) n znakov, pričom j -ty znak je určený vzhľadom na vyhodnotenie podmienok v riadkoch 10 a 13. Najskôr analyzujeme podmienku v riadku 10: ak je podmienka splnená tak sa na výstup vypíše znak # (mriežka). Podmienka obsahuje zložený logický výraz, ktorý sa vyhodnotí ako pravda (true) vtedy keď sme v prvom riadku výstupu ($i==0$) alebo sme v prvom stĺpci ($j==0$) alebo v poslednom riadku ($i==n-1$) alebo v poslednom stĺpci ($j==n-1$) alebo na hlavnej diagonále ($i==j$). Na výstup program vypíše štvorec (maticu) znakov: n riadkov, každý obsahujúci n znakov (stĺpcov), pričom na okraji a na hlavnej diagonále budú znaky #.

Zostávajúce znaky sa určia podľa ternárneho operátora v riadku 13: v prípade, že sme na riadku s poradovým číslom väčším ako poradové číslo stĺpca tak program vypíše . (bodku) inak vypíše medzeru. Logický výraz $i > j$ je splnený pre znaky pod hlavnou diagonálou, preto výsledný výstup bude vyzeráť takto:

1	#####
2	## #
3	#. #
4	#..#
5	#...#
6	#....##
7	#####
8	###
9	###
10	###

Zaujímavosťou je, že v prípade obrazca veľkosti 3 sa výstup javí len ako štvorec 3 x 3 znakov #. Dôvod je ten, že obrazec neobsahuje iné znaky ako tie, ktoré sú na okraji a na hlavnej diagonále.

Všimnime si ako sa vo výslednom obrazci prejavajú jednotlivé podmienky:

- Podmienky rovnosti ($i==0$, $i==n-1$ atď.) sú vo výslednom obrazci ako čiary. Operáciou logického súčtu (operátor $||$) môžeme tieto čiary v obrazci spájať. Napr. ak by sme chceli pridať čiaru pre vedľajšiu diagonálu, tak do zloženej podmienky (riadok 10) by sme pridali podmienku rovnosti $i==n-j-1$.
- Podmienky nerovnosti ($i > j$) sa vo výslednom obrazci prejavujú ako plochy znakov. Aj v tomto prípade môžeme operáciou logického súčtu takéto plochy

do obrazca pridať. Napr. pridaním podmienky $i < n/2$ do výrazu v riadku 13 by sme (bodkami) vyplnili aj plochu hornej polovice štvorca.

Úloha 2-15 (riešenie)

Najskôr musíme definovať premenné, ktoré sú v programe použité. V riadku 6 do premennej n načítavame cez formátovací reťazec `%d` celé číslo typu `int`. Ďalej v programe pracujeme s dvoma riadiacimi premennými cyklu (i a j), ktoré rozsahom (rádovo) zodpovedajú rozmeru hviezdy ($2*n+1$), preto všetky použité premenné môžeme definovať ako typ `int`. Program by mal vykresliť obrazec rozmerov $2*n+1$ riadkov a $2*n+1$ stĺpcov, preto počet opakovaní cyklov v riadkoch 7 a 9 zodpovedá tomuto rozmeru. Zostáva nám doplniť podmienky určujúce, ktorý znak sa vypíše v i -tom riadok a j -tom stĺpci obrazca.

Podmienka v riadku 10 určuje či sa vypíše znak `#`. Znak `#` vo výstupe tvoria spojenie dvoch čiar, preto v zloženej podmienke použijeme logické výrazy rovnosti (`==`): stredový riadok (`i==n`) a stredový stĺpec (`j==n`). Podmienka v riadku 12 potom určuje, kde sa vypíše znak `o`. Znak `o` vo výstupe tvoria dve plochy, preto použijeme logické výrazy nerovnosti: ľavý-horný kvadrant (`i<n && j<n`) a pravý dolný kvadrant (`i>n && j>n`).

Výsledný doplnený program vyzerá takto:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n, i, j;
6      while (scanf("%d", &n) == 1)
7          for (i = 0; i < 2*n+1; i++)
8              {
9                  for (j = 0; j < 2*n+1; j++)
10                     if (i == n || j == n)
11                         putchar('#');
12                     else if (i<n && j<n || i>n && j>n)
13                         putchar('o');
14                     else
15                         putchar('.');
16                     printf("\n");
17             }
18     return 0;
19 }
```

Úloha 2-16 (riešenie)

Pre danú veľkosť n je základný vzor obrazec veľkosti n riadkov a $2*n-1$ stĺpcov. Obrazec teda budeme vypisovať dvojitém vnoreným cyklom: vonkajší cyklus i pre riadky, a vnútorný cyklus j pre stĺpce. Musíme ešte navrhnuť podmienku, ktorá určí znak v i -tom riadku a j -tom stĺpci. Keďže v rámci riadku potrebujeme vypísať interval, musíme to zachytiť nerovnosťami: $j \geq n-i$ && $j \leq n-2+i$

Výsledný program pre vykreslenie základného vzoru je takýto:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n, i, j, k, r, s;
6      while (scanf("%d %d %d", &n, &r, &s) == 3)
7          // while (r--)
8          for (i = 1; i <= n; i++)
9          {
10             // for (k = 0; k < s; k++)
11             for (j = 0; j < 2*n-1; j++)
12                 if (j >= n-i && j <= n-2+i)
13                     printf("%d", i);
14                 else
15                     putchar(' ');
16             printf("\n");
17         }
18     return 0;
19 }
```

Program ešte rozšírime o vypísanie vzoru S-krát za sebou a R-krát pod seba:

Ak chceme **vykresliť základný vzor S-krát za sebou**, musíme v každom riadku cyklus S-krát zopakovať: zadefinujeme teda novú premennú k a cyklus v riadku 11 „obalíme“ novým vonkajším cyklom. Úprava je ako komentár v riadku 10.

Ak chceme **vykresliť základný vzor R-krát pod seba**, musíme celé spracovanie R-krát zopakovať: teda R-krát zopakujeme vypísanie základného vzoru S-krát za sebou. K tomu potrebujeme doterajšie spracovanie jedného vstupu (riadky 8-17) „obaliť“ novým vonkajším cyklom, ktorý sa zopakuje R-krát. Môžeme to spraviť napr. pridaním `while` cyklu ako je zapísané v riadku 7. Uvedený zdrojový kód po pridaní (odkomentovaní) príkazov na riadkoch 7 a 10 vypíše výstup podľa zadania úlohy.

Kapitola 3

Funkcie

Funkcia v jazyku C je pomenovaný blok príkazov s jasne definovaným rozhraním (vstupy, výstupy).

Napr. nasledujúca funkcia **min2()** pre vstupné hodnoty čísel **x** a **y** určí ako výstup minimum z hodnôt **x** a **y**, alebo funkcia **faktorial()** pre vstupné číslo **n** určí ako výstup hodnotu faktoriálu (**n!**):

```
1 int min2(int x, int y)
2 {
3     if (x < y)
4         return x;
5
6     return y;
7 }
```

```
1 int faktorial(int n)
2 {
3     int i, vysledok = 1;
4     for (i = 2; i <= n; i++)
5         vysledok *= i;
6     return vysledok;
7 }
```

Funkcie sú najdôležitejší nástroj na efektívne riešenie úloh v programovaní. Funkcie sú základné stavebné bloky umožňujúce zrozumiteľne realizovať rozsiahle a komplikované výpočty, ktoré sú ľahko udržiavateľné.

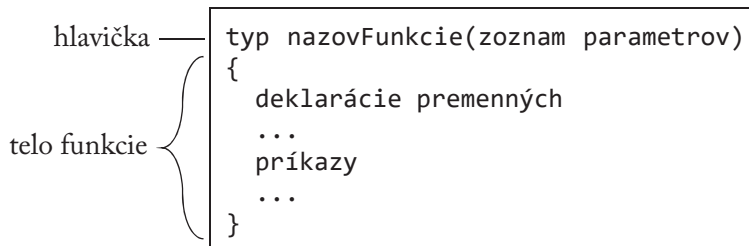
Rozsiahle výpočty sa pomocou funkcií rozdelia na menšie podproblémy. Komplikované výpočty sa pomocou funkcií rozdelia na jednoduchšie podproblémy. Vhodným použitím funkcií možno odstrániť opakujúci sa kód a testovať funkčnosť programu po menších celkoch, čím sa zlepšuje robustnosť a udržiavateľnosť vytvoreného zdrojového kódu.

Naopak, funkcie môžu byť aj „zlé“: keď sú príliš dlhé (obsahujú veľa príkazov) sú neprehľadné a málo zrozumiteľné, alebo keď nemajú jasne špecifikované rozhranie (vstupy a výstupy) môžu byť v programe nesprávne použité a môžu byť aj chybné, resp. spôsobiť chybné správanie celého programu, keď pre niektoré vstupy nesprávne určia výstup príp. majú neželané vedľajšie účinky, ktoré sa môžu prejaviť v programe až neskôr vo výpočte.

3.1 Jednoduché funkcie

Funkcia v jazyku C je pomenovaný blok príkazov, ktorý vykonáva určitú úlohu, s jasne definovaným rozhraním (vstupy, výstupy). Každý program v jazyku C sa skladá z jednej alebo viacerých funkcií: v každom programe musí byť aspoň jedna (hlavná) funkcia **main()**, ktorou sa začína výpočet v programe.

Definícia každej funkcie v zdrojovom kóde obsahuje nasledovné časti:



Všetko čo je pred prvou otváracou zloženou zátvorkou (**{**) nazývame **hlavička funkcie**, ktorá pri definícii funkcie nie je (!) ukončená bodkočiarkou. Všetko čo je medzi prvou otváracou a poslednou ukončovacou zloženou zátvorkou (**}**) nazývame **telo funkcie**. Zoznam parametrov tvoria čiarkami oddelené deklarácie parametrov (argumentov), ak funkcia nemá parametre uvádza sa typ **void**. Typ uvedený pred názvom funkcie je typ návratovej hodnoty, ktorú funkcia po ukončení vráti. Ak funkcia nevracia výsledok ako návratovú hodnotu uvedie sa typ **void**. V rámci tela funkcie môžeme deklarovať tzv. lokálne premenné, ktoré v pamäti existujú len počas behu funkcie, teda môžeme ich používať len v rámci príkazov tela funkcie.

Napr. v nasledujúcom programe je použitá funkcia **obvod_obdlznika()**, ktorá pre vstupné hodnoty dĺžok strán obdĺžnika (typ **double**) vráti hodnotu obvodu (typ **double**) ako návratovú hodnotu:

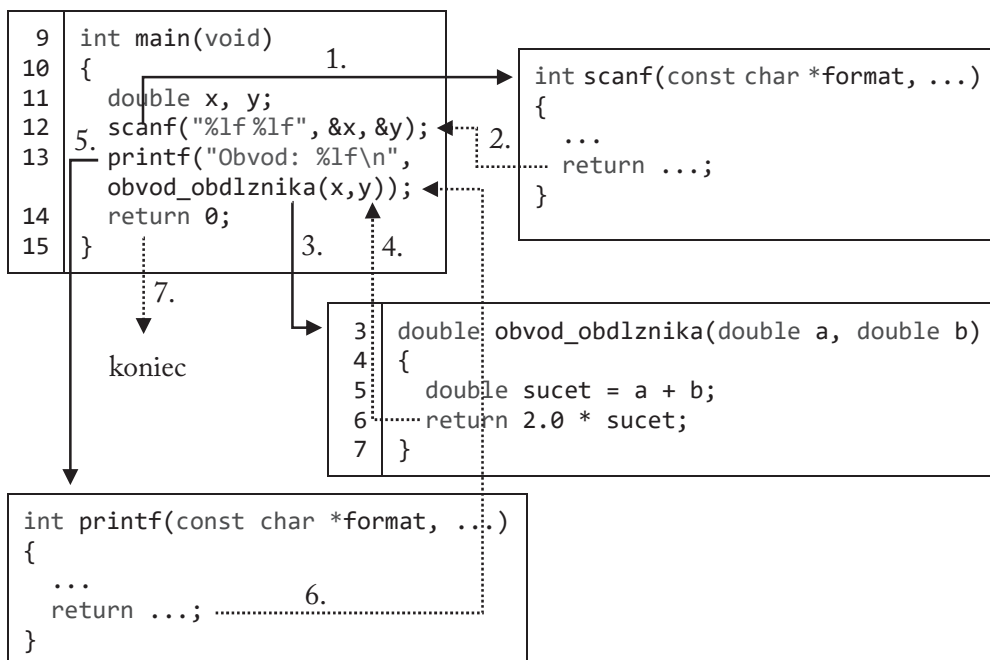
1	#include<stdio.h>
2	
3	double obvod_obdlznika(double a, double b)
4	{
5	double sucet = a + b;
6	return 2.0 * sucet;
7	}
8	

```

9  int main(void)
10 {
11     double x, y;
12     scanf("%lf %lf", &x, &y);
13     printf("Obvod: %lf\n", obvod_obdlznika(x,y));
14     return 0;
15 }

```

Riadenia výpočtu sa začne vykonávaním funkcie **main()**: v riadku 11 deklarujeme premenné pre desatinné čísla **x** a **y** (typ **double**), v riadku 12 načítame zo vstupu hodnoty do premenných **x** a **y**, v riadku 13 vypíšeme hodnotu obvodu obdĺžnika so stranami **x** a **y**. Jednotlivé kroky odovzdávania riadenia programu (control flow) do funkcií sú znázornené na nasledujúcom obrázku:



Ako prebehne vykonanie riadku 13? Program narazí na volanie funkcie: príkaz volania funkcie je tvorený identifikátorom (názvom) funkcie (**printf**) a zoznamom argumentov v okrúhlych zátvorkách. Príkaz volania funkcie sa vykoná tak, že sa najskôr vyhodnotia argumenty funkcie a potom sa odovzdá riadenie funkcii a funkcia sa vykoná: v pamäti sa počas vykonávania funkcie najskôr vyhradí miesto pre parametre funkcie (podobne ako pre lokálne premenné), ktoré nadobudnú

hodnoty, ktoré im boli odovzdané vo volaní. Potom sa pokračuje vo vykonávaní príkazov tela funkcie. Keď skončí vykonávanie tela funkcie, tak sa riadenie programu vráti späť na miesto, kde bola funkcia zavolaná.

V tomto prípade je potrebné (predtým ako odovzdáme riadenie funkcii **printf()** a vypíšeme výsledok na obrazovku) najskôr zavolať funkciu **obvod_obdlznika()**, ktorej sú odovzdané aktuálne hodnoty premenných **x** a **y**. Funkcia vypočíta výsledok (typ **double**), ktorý potom vstúpi ako vstupný argument do funkcie **printf()**, ktorá ho vypíše na obrazovku. Riadenie výpočtu vo funkcii **obvod_obdlznika()** prebieha postupným vykonávaním príkazov: v riadku 5 vyhradíme v pamäti miesto pre premennú **sucet** (typ **double**), ktorú inicializujeme hodnotou súčtu vstupných parametrov **x** a **y**, v riadku 6 ukončíme vykonávanie funkcie príkazom **return**, pomocou ktorého funkcia vráti hodnotu výrazu (**2.0 * sucet**) ako návratovú hodnotu.

Vyhodnocovanie argumentov funkcie

Informácie môže funkcia z programu získať dvoma spôsobmi: z globálnych (definovaných mimo funkcie) premenných alebo cez (vstupné) argumenty. Použitie globálnych premenných zvyšuje závislosť (tzv. previazanosť) funkcie od globálneho stavu programu a obmedzuje jej prípadnú budúcu znovupoužiteľnosť v iných situáciách. Je vhodné, keď funkcia pracuje len so vstupnými argumentami (parametrami) a výsledok vráti ako návratovú hodnotu, resp. vo výstupných argumentoch.

V programovacom jazyku C sú argumenty (parametre) do funkcie **odovzdané hodnotou**. To znamená, že pri každom volaní funkcie sa najskôr vyhodnotia výrazy zodpovedajúce argumentom, potom sa získané hodnoty nakopírujú do nových premenných (podľa definície argumentov v hlavičke funkcie), s ktorými pracuje telo funkcie. Akékoľvek zmeny týchto nových premenných v rámci vykonávaného tela funkcie teda nie sú prenesené do hodnôt premenných vo volajúcej funkcii.

V nasledujúcom programe zostane hodnota premennej **i** vo funkcii **main()** nezmenená, napriek tomu, že funkcia **nastav()** ju zamýšľa nastaviť na hodnotu 10.

```
1  #include<stdio.h>
2
3  void nastav(int x)
4  {
5      x = 10;
6  }
7
8  int main(void)
9  {
10     int i = 3;
11     printf("Povodna hodnota: %d\n", i); // 3
12     nastav(i);
13     printf("Upravena hodnota: %d\n", i); // 3
14     return 0;
15 }
```

Niektoré programovacie jazyky (napr. C++) umožňujú aj tzv. volanie odkazom (referenciou), ktorá umožňuje argumenty v rámci funkcie upravovať, pričom vykonané úpravy sú vykonávané priamo v premennej, ktorú sme do funkcie pri volaní odovzdali. V jazyku C toto nie je priamo takto možné, ale vyžaduje nepriamy spôsob použitím smerníku.

Ak chceme aby funkcia **nastav()** upravila hodnotu premennej **i** vo funkcii **main()** musíme jej odovzdať informáciu, kde sa premenná **i** v pamäti nachádza, na základe ktorej funkcia **nastav()** upraví príslušné pamäťové miesto a úprava sa tak prejaví v upravenej hodnote premennej **i**. Umiestnenie premennej **i** v pamäti zodpovedá adrese **&i**, a teda upravený program, v ktorom do funkcie **nastav()** odovzdáme hodnotu adresy premennej je takýto:

```
1  #include<stdio.h>
2
3  void nastav(int *x)
4  {
5      *x = 10;
6  }
7
8  int main(void)
9  {
10     int i = 3;
11     printf("Povodna hodnota: %d\n", i); // 3
12     nastav(&i);
13     printf("Upravena hodnota: %d\n", i); // 10
14     return 0;
15 }
```

V literatúre sa môžete stretnúť s pomenovaním tohto prístupu ako: volanie „odkazom“ (v úvodzovkách). Dôležité si však uvedomiť, že v jazyku C sú argumenty odovzdávané vždy len hodnotou a v tomto prípade odovzdávame hodnotu smerníku (odkazu) na typ hodnoty, ktorý chceme „vrátiť“ cez argument.

Predchádzajúci prístup je nutné použiť aj v prípade ak potrebujeme, aby funkcia vrátila dva alebo viac výstupov. Napr. nasledujúca funkcia **kruh()** pre vstupný argument **polomer** určí **obvod** aj **obsah** kruhu, pričom na vrátenie oboch z nich nie je možné použiť (jednu) návratovú hodnotu, preto funkcii prostredníctvom argumentu odovzdáme (hodnoty) smerníkov na desatinné čísla (typ **double***), do ktorých funkcia výsledky zapíše. Niekedy sa takýto argument označuje ako výstupný argument – lebo prostredníctvom neho funkcia zapisuje výstup (výsledok).

```
1 void kruh(double polomer, double *obvod, double *obsah)
2 {
3     *obvod = 2 * 3.14159 * polomer; // 2*PI*r
4     *obsah = 3.14159 * polomer * polomer; // PI*r^2
5 }
6
7 int main(void)
8 {
9     double r, o, s;
10    scanf("%lf", &r);
11    kruh(r, &o, &s);
12    printf("Obvod kruhu: %lf\n", o);
13    printf("Obsah kruhu: %lf\n", s);
14    return 0;
15 }
```

Typický príklad je výmena dvoch hodnôt. Uvažujme napr. nasledovnú funkciu **vymen()**, ktorá pomocou pomocnej premennej **tmp** vymení hodnoty dvoch celých čísel (argumenty funkcie sú smerníky na pamäť, ktorá obsahuje hodnoty, ktoré chceme vymeniť):

```
1 void vymen(int *x, int *y)
2 {
3     int tmp = *x;
4     *x = *y;
5     *y = tmp;
6 }
7
8
```

```
9  int main(void)
10 {
11     int i = 5, j = 10;
12     printf("%d %d\n", i, j); // 5 10
13     vymen(&i, &j);
14     printf("%d %d\n", i, j); // 10 5
15
16     // chybne pouzitie:
17     // vymen(i, j); alebo vymen(*i, *j);
18     return 0;
19 }
```

Upozornenie: V prípade funkcií s viac argumentami, poradie vyhodnocovania je v štandarde jazyka ANSI C nešpecifikované: **argumenty môžu byť vyhodnotené v ľubovoľnom poradí**. Poradie vyhodnocovania závisí od kompilátora a môže závisieť aj od spôsobu optimalizácie programu pri kompilácii.

Príklad nesprávneho volania funkcie je uvedený v nasledujúcom programe, ktorý v rámci volania funkcie **fn()** v hlavnej funkcii **main()** v argumentoch upravuje hodnotu premennej **i**, čím spôsobí nedefinované správanie programu. Hodnoty argumentov, s ktorými je funkcia **fn()** zavolaná, sú prekvapujúce a prakticky nepoužiteľné:

```
int fn(int a, int b)
{
    printf("%d %d\n", a, b); // 2 3
}

int main(void)
{
    int i = 0;
    fn(i++, i+=2);
    return 0;
}
```

Nielenže výraz v druhom argumente (**i+=2**) bol vyhodnotený pred prvým (**i++**) lebo **a=2**, ale efekt výrazu v prvom argumente (**i++**) sa prejavil aj v hodnote druhého argumentu lebo **b=3**.

Poučenie: Argumenty, ktoré dávame do funkcií by mali obsahovať len výrazy, ktoré hodnoty čítajú. Akékoľvek úpravy hodnôt premenných treba vykonať predtým ako ich použijeme v argumentoch pri volaní funkcií.

Poradie funkcií v zdrojovom kóde

Všetky funkcie v zdrojovom kóde programu v jazyku C sú definované na rovnakej úrovni – funkciu v jazyku C nie je možné definovať v rámci tela inej funkcie. Dôležité však je aj poradie, v akom sú funkcie v zdrojovom kóde definované (definícia funkcie sa skladá z hlavičky a tela funkcie). Každú funkciu, ktorú v programe používame (voláme), by sme mali v rámci zdrojového kódu aj niekde definovať. Ak chceme funkciu použiť v zdrojovom kóde skôr ako je definovaná alebo je definovaná v inom súbore musíme vopred uviesť tzv. prototyp funkcie. Prototyp funkcie je forma deklarácie (teda uvedenie typu – hlavičky) funkcie bez toho, aby sme uviedli implementáciu (telo funkcie). V tomto kontexte sa nepoužíva pojem deklarácia funkcie preto, lebo z historických dôvodov deklarácia funkcie uvádza len názov funkcie a typ návratovej hodnoty bez uvedenia typov parametrov. Bez toho je deklarácia prakticky nepoužiteľná, preto štandard ANSI C umožňuje použiť prototyp funkcie, čo je istým spôsobom rozšírená deklarácia funkcie.

Úloha 3-1

Napíšte funkciu `min3()`, ktorá určí minimum z troch celých čísel (typu `int`). Preskúmajte nasledujúci zdrojový kód a zistite, prečo je chybný. Navrhnite vlastné riešenie, spravte ho čo najefektívnejšie a pokúste sa aj zdôvodniť, prečo je efektívne.

```
1 int min3(int a, int b, int c)
2 {
3     if (a < b && a < c)
4         return a;
5     if (b < a && b < c)
6         return b;
7     return c;
8 }
```

Úloha 3-2

Napíšte funkciu `mocnina()`, ktorá pre reálne číslo `x` (typ `double`) a celé číslo `n` (typ `int`) vypočíta hodnotu mocniny x^n . Napr. pre `x = 1.1`, a `n = 5` funkcia vráti 1.61051.

Úloha 3-3

Napište funkciu **dalsie_prvocislo()**, ktorá pre celé číslo **x** (typ **int**) nájde najbližšie prvočíslo väčšie ako **x**. Napr. pre **x = 20** funkcia vráti 23.

Úloha 3-4

Doplňte chýbajúce časti funkcie **kvadraticka_rovnica()**, ktorá pre koeficienty **a**, **b**, **c** (typ **double**) nájde korene kvadratickej rovnice $ax^2 + bx + c = 0$. Predpokladajte, že **a** $\neq 0$. Funkcia nevráti oba korene rovnaké.

```
1  kvadraticka_rovnica( )
2  {
3      double d = ;
4      if (d < 0.0)
5          return ;
6      d = sqrt(d);
7      *x1 = ;
8      if ( )
9          return ;
10     *x2 = ;
11     return ;
12 }
```

Doplňujúca úloha: Ako by ste upravili funkciu, aby správne pracovala aj pre **a = 0**?

Úloha 3-5

Analyzujte nasledujúcu funkciu **priesečník_priamok()**, ktorá určí priesečník priamok $[x_1, y_1] - [x_2, y_2]$ a $[x_3, y_3] - [x_4, y_4]$. Funkcia vráti 1 ak majú priamky jeden priesečník, inak vráti 0. Je v implementácii chyba? Napr. pre priamky $[0, 0] - [1, 0]$ a $[3, 5] - [3, 4]$ funkcia vráti (správny) priesečník $[3, 0]$.

```
1  void rovnica(double x1, double y1, double x2, double y2,
2              double *a, double *b, double *c)
3  {
4      *a = y1 - y2;
5      *b = x2 - x1;
6      *c = x1*y2 - y1*x2;
7  }
8
9
```

```
10 int priesečník_přiamok(double x1, double y1, double x2, double y2,  
11                       double x3, double y3, double x4, double y4,  
12                       double *x, double *y)  
13 {  
14     double a1, b1, c1, a2, b2, c2;  
15     rovnica(x1, x2, y1, y2, &a1, &b1, &c1);  
16     rovnica(x3, x4, y3, y4, &a2, &b2, &c2);  
17  
18     double d = a1*b2 - b1*a2;  
19     if (fabs(d) < 0.000000001)  
20         return 0;  
21     *x = c1*a2 - a1*c2;  
22     *y = b1*c2 - c1*b2;  
23     return 1;  
24 }
```

Úloha 3-6

Určite výstup, ktorý vypíše nasledujúci program.

```
1  #include <stdio.h>  
2  
3  int d(int x) { return 3 * x; }  
4  int c(int x) { return d(x) + 1; }  
5  int b(int x) { return 2 * c(d(x)) - 3; }  
6  int a(int x) { return 2 * b(c(x)) + 1; }  
7  
8  int main(void)  
9  {  
10     printf("%d\n", a(0));  
11     return 0;  
12 }
```

3.2 Rekurzívne funkcie

Programovací jazyk C prostredníctvom zásobníku volaní umožňuje každej funkcii v rámci svojho tela zavolať akúkoľvek inú funkciu. Je teda úplne bežné, že funkcia priamo alebo nepriamo zavolá funkciu s tým istým názvom, zvyčajne však s inými parametrami (hodnotami argumentov). Takéto funkcie nazývame rekurzívne funkcie a princíp riešenia úloh využívajúci rekurzívne funkcie nazývame rekurzia.

Uvažujme nasledujúci príklad (rekurzívnej) funkcie **nekonečno()**:

```
1 #include <stdio.h>
2
3 void nekonecno()
4 {
5     printf("Nikdy neskoncim!?\n");
6     nekonecno();
7 }
8 int main(void)
9 {
10    nekonecno(0);
11    return 0;
12 }
```

Funkcia trochu naivne vypisuje na výstup správu **Nikdy neskoncim!?**

Otázka na zamyslenie: Naozaj funkcia nikdy neskončí?

Odpoveď: Každé ďalšie zavolanie funkcie zvýši hĺbku vnorenia a vyžaduje pamäť na zásobníku volaní. Zásobník má obmedzenú veľkosť (závislú od kompilátora alebo od prostredia operačného systému), a preto v konečnom čase (zvyčajne do pár sekúnd) tento program skončí.

Ak chceme rekurziu efektívne použiť pri riešení problémov, musíme jasne definovať **terminálnu (ukončovaciú) podmienku** – teda určiť situáciu, pri ktorej rekurzívna funkcia nebude pokračovať v ďalšom vnáraní.

Rekurzívna funkcia zvyčajne prostredníctvom argumentov (parametrov) špecifikuje riešený problém. Riešený problém, ktorý je natoľko jednoduchý, že ho už nemá zmysel ďalej deliť na menšie podproblémy, nazývame **základný prípad**. Rekurzívna funkcia výsledok problému pre základný prípad vypočíta priamo bez ďalšieho volania funkcií. Základné prípady riešeného problému teda zodpovedajú terminálnej podmienke v implementácii rekurzívnej funkcie.

V prípade, že rekurzívna funkcia dostane vo vstupných argumentoch zadaný problém, ktorý nie je základný prípad, zvyčajne problém zjednoduší vykonaním jedného kroku vo výpočte a zostávajúci výpočet ponechá na rekurzívne volanie so zníženou hodnotou argumentu, čím sa v ďalšom volaní priblíži k splneniu terminálnej podmienky, a teda v konečnom počte krokov dosiahne terminálnu podmienku a rekurzívne vnáranie sa zastaví, pričom pri následnom vynáraní sa postupne určí výsledná hodnota. Príklady rekurzie, ktoré sú uvedené azda vo všetkých

knížkách o programovaní, sú matematické funkcie, ktoré majú už priamo rekurzívnu definíciu: výpočet hodnoty faktoriálu ($n!$) alebo výpočet n -tého Fibonacciho čísla. Paradoxne, tieto prípady nie sú najlepšie ukážky použitia rekurzie v programovaní, pretože obe úlohy majú výrazne efektívnejšie iteratívne riešenie použitím cyklov.

Sila rekurzie spočíva v jednoduchosti vyjadrenia riešeného problému pomocou menších problémov (rovnakého typu), teda jedná sa o celkový princíp riešenia (zložitých) problémov. Nevýhody rekurzie spočívajú najmä v nárokoch na pamäť, ktorá je potrebná pre uchovanie potrebných informácií (hodnôt argumentov, lokálnych premenných, a pod.) každého jedného volania funkcie. V prípade rekurzie je zvyčajne tých volaní (rekurzívnych) funkcií viac ako pri použití iterácie a preto sú pamäťové nároky na zásobník výrazne vyššie. Pri použití rekurzie pri riešení problémov preto treba vždy veľmi dôsledne vyhodnotiť, či pamäťové nároky na zásobník nebudú (pre požadovaný rozsah vstupov) príliš veľké, čo by mohlo celý výpočet v porovnaní s iteratívnym riešením značne spomaliť.

Uvažujme teda zaužívanú rekurzívnu implementáciu matematickej funkcie ($n!$) **faktorial()**. Implementáciu sme rozšírili o výpis adresy argumentu a lokálnej premennej ako aj priebežných výsledkov (návratových hodnôt pred tým ako ich funkcia vráti):

```
1  #include <stdio.h>
2
3  int faktorial(int n)
4  {
5      int v;
6      printf("f(n=%d): &n = %p, &v = %p\n", n, (void*)&n, (void*)&v);
7      if (n < 2)
8      {
9          printf("  return 1 ... f(n=%d)\n", n);
10         return 1;
11     }
12     v = n * faktorial(n - 1);
13     printf("  return %d ... f(n=%d)\n", v, n);
14     return v;
15 }
16
17 int main(void)
18 {
19     printf("5! = %d\n", faktorial(5));
20     return 0;
21 }
```

Výstup programu (pre prehľadnosť sme doplnili odsadenie podľa úrovne vnorenia):

1	f(n=5): &n = 0x7ffdfacfc0c, &v = 0x7ffdfacfc14
2	f(n=4): &n = 0x7ffdfacfcadc, &v = 0x7ffdfacfcac4
3	f(n=3): &n = 0x7ffdfacfc0c, &v = 0x7ffdfacfcab4
4	f(n=2): &n = 0x7ffdfacfc7c, &v = 0x7ffdfacfc84
5	f(n=1): &n = 0x7ffdfacfc4c, &v = 0x7ffdfacfc54
6	return 1 ... f(n=1)
7	return 2 ... f(n=2)
8	return 6 ... f(n=3)
9	return 24 ... f(n=4)
10	return 120 ... f(n=5)
11	5! = 120

Pre výpočet hodnoty faktoriálu pre vstupnú hodnotu **n** sa funkcia **n**-krát vnorí, na najhlbšej úrovni vnorenia (pre **n = 1**) je výsledok 1 a potom sa vynára, pri každom vnorení funkcia prenášobí číslom **n** hodnotu získanú z vnoreného volania (pre **n-1**).

Vo výpise si môžeme všimnúť, že argument **n** a priebežný výsledok **v** majú vždy iné adresy (umiestnenie v pamäti), čo znamená, že pre každú volanú (rekurzívnu) funkciu program vytvorí (na zásobníku) nové argumenty a nové lokálne premenné. Všimnime si, že hodnotu smerníku vypíše **%p** v hexadecimálnom zápise (v tvare **0x**).

Nevýhodou rekurzívneho riešenia je pamäť potrebná pre udržiavanie stavu vnorení: pri najhlbšom vnorení je potrebné uchovávať naraz v pamäti informácie o **n** vnorených volaniach funkcií.

V prípade iteratívneho riešenia postačuje pre ľubovoľné **n** len jedna premenná s výsledkom a jedna riadiaca premenná cyklu. V prípade rekurzívneho výpočtu faktoriálu však v praxi na pamäťový nedostatok (spôsobený rekurzívnym vnáraním funkcií) nenarazíme, pretože výsledná hodnota faktoriálu rýchlo rastie a skôr narazíme (už pri malých hodnotách **n**) na obmedzenie dátového typu **int** (32 bit) alebo **long** (64 bit).

Úloha 3-7

Dané je celé číslo **x** (typ **int**). Nasledujúce funkcie použitím cyklu určia ciferný súčet čísla **x**. Napíšte rekurzívnu funkciu **ciferny_sucet()**, ktorá vypočíta ten istý výsledok, nepoužívajte pritom cykly ani globálne alebo statické premenné.

```
1 int ciferny_sucet(int x)
2 {
3     int vysledok = 0;
4     while (x > 0)
5     {
6         vysledok += x % 10;
7         x /= 10;
8     }
9     return vysledok;
10 }
```

```
1 int ciferny_sucet(int x)
2 {
3     int vysledok;
4     for (vysledok = 0; x > 0; x /= 10)
5         vysledok += x % 10;
6     return vysledok;
7 }
```

Úloha 3-8

Dané je nezáporné celé číslo **x** (typ **unsigned int**) do 1 000 000. Nasledujúca funkcia použitím cyklu vypíše číslo **x** v binárnom zápise. Napíšte rekurzívnu funkciu, ktorá vypočíta ten istý výsledok, nepoužívajte pritom cykly ani globálne alebo statické premenné.

```
1 void print_bin(unsigned int x)
2 {
3     long long i = 1, vysledok = 0;
4     for (; x > 0; x /= 2, i *= 10)
5         vysledok = vysledok + (x % 2) * i;
6     printf("%llu\n", vysledok);
7 }
```

Úloha 3-9

Dané je celé číslo **x** (typ **int**). Nasledujúca funkcia použitím cyklu určí, či je číslo **x** palindróm. Napíšte rekurzívnu funkciu, ktorá vypočíta ten istý výsledok, nepoužívajte pritom cykly ani globálne alebo statické premenné.

```
1 int palindrom(int x)
2 {
3     int i, otocene = 0;
4     for (i = x; i > 0; i /= 10)
5         otocene = 10*otocene + (i % 10);
6     return otocene == x;
7 }
```

Úloha 3-10

Dané sú celé čísla **k** a **n** (typ `int`). Napíšte rekurzívnu funkciu, ktorá na výstup vypíše variácie s opakovaním **k**-tej triedy z **n** prvkov podľa uvedenej ukážky. Môžete používať cykly, globálne aj statické premenné. Nasledujúca funkcia použitím troch vnorených cyklov vypíše variácie s opakovaním 3-triedy z **n** prvkov (**k** = 3), uvedený postup sa však nedá bez použitia rekurzie zovšeobecniť pre premenlivú hodnotu **k**.

1	<code>void print_var3(int n)</code>		
2	<code>{</code>		
3	<code> int i, j, k;</code>		
4	<code> for (i = 1; i <= n; i++)</code>		
5	<code> for (j = 1; j <= n; j++)</code>		
6	<code> for (k = 1; k <= n; k++)</code>		
7	<code> printf("%d%d%d\n", i, j, k);</code>		
8	<code>}</code>		

	Výstup <code>print_var3(2):</code>
1	111
2	112
3	121
4	122
5	211
6	212
7	221
8	222

Doplňujúca úloha: Pokúste sa túto úlohu vyriešiť:

1. Bez použitia globálnych alebo statických premenných, cykly môžete použiť.
2. Bez použitia cyklov, globálnych alebo statických premenných.

3.3 Rozsah platnosti identifikátorov

Identifikátor je programátorom určené pomenovanie nejakého prvku v programe, najčastejšie označuje nejakú premennú alebo funkciu. Vhodne zvolené názvy identifikátorov môžu program sprehľadniť a naopak nezrozumiteľné identifikátory môžu zdrojový kód rýchlo zneprehľadniť.

Identifikátor v prípade premenných reprezentuje previazanie názvu (identifikátora) s pamäťou premennej, ku ktorej pristupujeme, keď v zdrojovom kóde použijeme príslušný identifikátor. Nové identifikátory zavádzame v blokoch, ktoré nám umožňujú definovať nové premenné. Rozsah platnosti identifikátora vymedzuje oblasť zdrojového kódu, v ktorej je identifikátor platný (viditeľný) a môžeme ho použiť na prístup k premennej, s ktorou je previazaný.

Blok je oblasť zdrojového kódu ohraničená zloženými zátvorkami, ktorá môže obsahovať definície premenných nasledované príkazmi alebo inými blokmi.

Bloky môžeme ľubovoľne vnárať, pričom ak vnorený blok začne v nejakom inom bloku, musí v ňom aj skončiť. Premenné, ktoré definujeme v bloku sú platné v rámci tohto aj vnorených blokov, ale nie mimo nich. Premenné sa snažíme nazývať najlepšie ako vieme, najčastejšie podľa účelu bloku, v ktorom ich používame. Môže sa nám ale ľahko stať, že rovnaký názov identifikátora použijeme v rozličných blokoch. Pravidlá pre rozsah platnosti identifikátorov potom určujú, ktorý z identifikátorov je použitý, a teda s ktorou hodnotou v programe pracujeme.

Základné pravidlo je, že v prípade rovnakého názvu je identifikátor z vonkajšieho bloku prekrytý identifikátorom z aktuálneho (vnoreného) bloku. Uvažujme nasledovný program s viacerými vnorenými blokmi. Preskúmajme, čo program vypíše na výstup.

```
1  #include <stdio.h>
2
3  int n = 5;
4
5  int main(void)
6  {
7      int i = 0;
8      printf("%d %d\n", i, n); // 0 5
9      {
10         int i = 2;
11         i++; n++;
12         printf("%d %d\n", i, n); // 3 6
13     }
14     printf("%d %d\n", i, n); // 0 6
15     {
16         int n = 7;
17         i++; n++;
18         printf("%d %d\n", i, n); // 1 8
19     }
20     printf("%d %d\n", i, n); // 1 6
21     return 0;
22 }
```

V riadku 8 program vypíše hodnotu lokálnej premennej **i** (definovanú v riadku 7) a globálnej premennej **n**. V prvom vnorenom bloku (riadky 9-13) definícia novej premennej **i** (riadok 10) prekryje identifikátor lokálnej premennej **i**, ktorá je definovaná v riadku 7. Zvýšenie hodnoty **i** v riadku 11 a výpis preto prebehne s touto novou premennou **i**. Podobne v druhom vnorenom bloku (riadku 15-19): definícia

novej premennej **n** prekryje identifikátor globálnej premennej **n** a operácie s identifikátorom **n** v tomto bloku pracujú s touto novou premennou. Globálna premenná **n** je v programe teda zvýšená len raz v riadku 11. Lokálna premenná **i** definovaná v riadku 7 je tiež zvýšená len raz v riadku 17. Konkrétne hodnoty vypísané na výstup, sú uvedené v komentári za príslušným riadkom.

Pamäťové triedy

Každá premenná má okrem dátového typu definovanú pamäťovú triedu, ktorá určuje oblasť pamäte, v ktorej je premennej pridelená pamäť, ako aj dĺžku trvania pridelenia pamäte, čím ovplyvňuje aj rozsah platnosti identifikátora premennej, tzn. v akej oblasti zdrojového kódu možno identifikátor použiť.

Premenným definovaným v rámci bloku program automaticky prideliť pamäť (na zásobníku): pri vstupe do bloku je pamäť pridelená a pri ukončení vykonávania bloku je pamäť uvoľnená. Hodnoty premenných definovaných v bloku sa po ukončení bloku stanú nedostupné, platí to rovnako pre vnorené bloky v rámci funkcií ako aj celé funkčné bloky. V prípade, že chceme, aby bola premenná dostupná aj v nejakom inom konkrétnom bloku, je potrebné ju definovať v takom bloku, ktorý je nadradený všetkým blokom, v ktorých chceme aby bola dostupná.

Ak chceme premennú sprístupniť pre viacero funkcií je potrebné ju definovať mimo funkcie, takéto premenné nazývame externé alebo globálne, majú pamäťovú triedu **extern**. Sú viditeľné v celom programe a pamäť im zostane pridelená počas celého behu programu, teda ich hodnoty sú zachované bez ohľadu na vykonávaný blok kódu. Kľúčové slovo **extern** možno použiť pri definícii premennej v rámci bloku, čím programu oznámime, že uvedený identifikátor zodpovedá nejakej externej premennej. Program identifikátor previaže s pamäťou zodpovedajúcou príslušnej externej premennej. Externé premenné sa tiež používajú na zdieľanie stavu pamäte v situáciách ak zdrojový kód je tvorený viacerými súbormi.

Ak chceme obmedziť rozsah platnosti externých premenných použijeme pamäťovú triedu **static**, tzv. statické premenné. Statické premenné môžu byť definované v rámci bloku alebo ako globálne (externé) premenné. V prípade definície v rámci bloku sú viditeľné len v rámci tohto bloku a program im pridelí pamäť

pri prvom vstupe do bloku. Statické premenné sú automaticky inicializované na 0 alebo ich môžeme v zdrojovom kóde inicializovať inou konštantnou hodnotou.

Praktický tip: Typický príklad použitia statickej premennej je použitie vnoreného bloku so statickou premennou pri hľadaní chýb v programe.

Predstavme si, že niekde hlboko v programe máme chybu. Pridaním bloku so statickým počítadlom môžeme na ľubovoľnom mieste v programe počítať poradie, koľkokrát program vstúpil do predmetného bloku a vypísať pri tom potrebné informácie (hodnoty iných premenných). Potom, keď objavíme pri koľkom vykonávaní bloku vznikla chyba, môžeme si výpočet tesne predtým odkrokovať. Po opravení chyby, pridaný blok so statickým počítadlom bez ujmy priamo odstránime. Využitím bloku so statickým počítadlom nenarušíme výpočet neželanými úpravami iných premenných bez ohľadu na názov počítadla.

V prípade externých premenných, ktoré definujeme ako statické: externé statické premenné sú dostupné len v súbore, v ktorom sú definované (nie je možné ich použiť v inom súbore) a len v tej časti kódu, ktorá nasleduje za definíciou premennej (teda nie sú dostupné v rámci definovaného súboru skôr ako boli definované). Statické externé premenné sú vhodné ako forma obmedzenia rozsahu platnosti pre premennú, ktorú používa viacero funkcií v rámci jedného zdrojového súboru.

Poslednou možnosťou použitia pamäťovej triedy **static** je pri definovaní funkcií. Statické funkcie sú viditeľné len v rámci súboru, v ktorom boli definované.

Upozornenie: Programovací jazyk C formálne obsahuje ešte dve pamäťové triedy, ktoré nepoužívajte! Pamäťová trieda **auto** je pridelovaná automaticky premenným, ktoré nemajú špecifikovanú inú triedu (lokálnym premenným a argumentom funkcií). Kľúčové slovo **auto** však medzičasom nadobudlo nový význam a v jazyku C++ znamená automaticky odvodený typ premennej, čo vzhľadom na časté kombinovanie programov v C a C++ môže spôsobiť neželané komplikácie.

Zostávajúca pamäťová trieda **register** označuje premennú, ktorú programátor odporúča umiestniť do registrov procesora, pri optimalizácii toto však kompilátor dokáže určiť lepšie.

Typové kvalifikátory

Každá premenná má povinne uvedený názov (identifikátor) a dátový typ, ktorý možno ďalej spresniť použitím typových kvalifikátorov. Typové kvalifikátory upresňujú ako možno s premennou v programe pracovať. Štandard ANSI C umožňuje použiť dva typové kvalifikátory: **const** a **volatile**.

Pridaním typového kvalifikátora **const** k premennej vyjadrujeme, že prostredníctvom tejto premennej nie je možné po prvotnej inicializácii ďalej meniť jej hodnotu. Zvolené kľúčové slovo, resp. označenie tohto typového kvalifikátora (**const**) je zdrojom mnohých chýb a nedorozumení. Najčastejšie použitie **const** je pre smerník v argumente funkcie, čím oznamujeme kompilátoru, že hodnotu nie je možné využitím tohto smerníku zmeniť. Zmeniť ju však môžeme iným spôsobom, nejde totiž o konštantu, dokonca je úplne bežné, že jej hodnoty sú v programe zmenené:

```
1 void fn(void)
2 {
3     int a = 10;
4     const int *p = &a;
5     printf("%d\n", *p); // 10
6     *p = 20; // chyba!!
7     printf("%d\n", *p); // 10
8     a = 20;
9     printf("%d\n", *p); // 20
10 }
```

Použitie typového kvalifikátora **const** pri smerníkoch umožňuje definovať:

- smerník na **const** typ (napr. **const int *p**), cez ktorý nie je možné hodnotu, na ktorú ukazuje upraviť, ale samotné smerník môžeme zmeniť a bude ukazovať na inú premennú.
- **const** smerník na bežný typ (napr. **int * const q**), ktorý nemôžeme zmeniť, ale môžeme upraviť hodnotu, na ktorú ukazuje (***q = 40**).
- **const** smerník na **const** typ (napr. **const int * const r = &a;**), cez ktorý nemôžeme zmeniť ani hodnotu, na ktorú ukazuje, ani nemôžeme zmeniť kam ukazuje.

Typový kvalifikátor **volatile** označujeme „nestále“ premenné, ktorých hodnota sa môže neočakávane meniť. Dôvod zmeny hodnoty takejto premennej je zvyčajne mimo rámec práve bežiacieho programu: hodnota sa mohla upraviť pri obsluhu prerušenia (angl. interrupt) v operačnom systéme, hodnotu mohol zmeniť asynchrónny výpočet v inom vlákne, resp. premenná môže zodpovedať stavu hardvérového zariadenia (napr. LED, FPGA čipu). Zvyčajné použitie tohto typového kvalifikátora je pre vývoj na tzv. embedované platformy (napr. Raspberry Pi, IOT zariadenia) a iné špecializované harvdérové zariadenia.

Typový kvalifikátor **volatile** možno kombinovať s kvalifikátorom **const**, čím označujeme premenné, ktorých hodnota sa môže neočakávane meniť, pričom z programu ich nemôžeme upravovať. Prakticky predstavuje typový kvalifikátor **volatile** predovšetkým usmernenie pre kompilátor, že akékoľvek operácie s danou premennou nemôžu byť optimalizované a vždy je potrebné pracovať s aktuálnou hodnotou v pamäti.

Nasledujú príklady definícií premenných s typovým kvalifikátorom **volatile** pre prístup k harvdérovým registrom:

1	// stav LED-ky
2	unsigned char volatile *p_led = (unsigned char *)0x1234;
3	
4	// hardverovy cas
5	extern const volatile int hw_clock;

V riadku 2 je uvedená definícia smerníku na **unsigned char**, cez ktorý môžeme pristupovať k harvdérovému registru na adrese 0x1234, v ktorom je umiestnený stav LED. Úpravou hodnoty tohto registru zmeníme reálny stav diódy v harvdérovom zariadení, pričom stav sa môže kedykoľvek zmeniť aj mimo vykonávaného programu (**volatile**). V riadku 5 je externá deklarácia premennej, ktorú môžeme len čítať (**const**) a ktorej hodnota môže byť kedykoľvek zmenená harvdérom (**volatile**).

V tejto knihe sa s premennými označenými typovým kvalifikátorom **volatile** nestretneme. Pri vývoji programov pre embedované platformy sa odporúča premenné označovať týmto kvalifikátorom, aby sa predišlo zbytočným ťažko odhaliteľným chybám.

Úloha 3-11

Programátor napísal trochu neprehľadný program. Určite výstup nasledujúceho programu:

```
1  #include <stdio.h>
2
3  int cislo = 3;
4
5  int fn(int cislo, int k)
6  {
7      cislo += k;
8      printf("%d\n", cislo);
9      {
10         int cislo = 2;
11         cislo += k;
12         printf("%d\n", cislo);
13     }
14     {
15         int k = 2;
16         cislo += k;
17         printf("%d\n", cislo);
18     }
19     return k;
20 }
21
22 int main(void)
23 {
24     cislo *= fn(1, 2);
25     printf("%d\n", cislo);
26     return 0;
27 }
```

Úloha 3-12

Uvažujte nasledujúcu implementáciu programu online herne a hráča. V súbore **online_herna.h** je hlavičkový súbor pre funkcie knižnice online herne, ktoré sú implementované v súbore **online_herna.c**. V súbore **hrac.c** je implementácia hráča herne, ktorý každý deň vsadí 10 % svojho účtu (majetku v herni) na udalosť s predpokladanou výhrou 1,1 * vklad.

Po 100 dňoch pravidelného stávkovania je výsledok pre hráča tragický:

Hrac: 214.0 EUR
Stat(5%) = 253.6 EUR
Stat(20%) = 106.5 EUR
Zisk herne : 425.9 EUR

Najviac z jeho peňazí dostane herňa (425,9 EUR), ďalšiu významnú časť si zoberie štát na povinnom odvode (5 %) a dani z príjmu (20 %): $253,6 + 106,5 = 360,1$ EUR, a hráčovi zostane 214,0 EUR. Pokračovať v stávkovaní situáciu hráča ďalej len zhoršuje.

Daná implementácia knižnice **online_herna.c**, umožňuje hráčovi vyhrať čo najčastejšie s tým, že si určité malé percento obratu zoberie herňa. Hráč teda veľmi často vyhráva aj keď menšie sumy, čo v ňom postupne buduje závislosť od hrania. Dôležité pre herňu je to, že hráč nikdy nepresiahne svoj počiatočný vklad. Nejakú takto zvyčajne funguje softvér online herní.

Programátor v knižnici **online_herna.c** ale spravil vážnu chybu. Pokúste sa upraviť zdrojový kód hráča (**hrac.c**) tak, aby ste začali častejšie vyhrávať.

```
1 // online_herna.h
2 double stavka(double vklad, double kurz);
3 double odvod_statu();
4 double dan();
5 double zisk();
```

```
1 // hrac.c
2 #include <stdio.h>
3 #include "online_herna.h"
4
5 double hrac = 1000.0; // vložil 1000 EUR
6
7 void skus(double suma)
8 {
9     hrac -= suma;
10    double vyhra = stavka(suma, 1.1);
11    if (vyhra > 0.0)
12        hrac += vyhra;
13 }
14
15 int main(void)
16 {
17     int i;
18     for (i = 0; i < 100; i++)
19         skus(hrac / 10.0);
20     printf("Hrac: %.11f EUR\n", hrac);
21     printf("Stat(5%): %.11f EUR\n", odvod_statu());
22     printf("Stat(20%): %.11f EUR\n", dan());
23     printf("Zisk herne: %.11f EUR\n", zisk());
24     return 0;
25 }
```

```
1 // online_herna.c
2 double ucet, odvod, na_vyhry;
3
4 double stavka(double vklad, double kurz)
5 {
6     odvod += 0.05 * vklad;
7     ucet += 0.95 * vklad;
8     na_vyhry += 0.85 * vklad;
9
10    double vyhra = vklad * kurz;
11    if (vyhra > na_vyhry)
12        return 0.0;
13
14    ucet -= vyhra;
15    na_vyhry -= vyhra;
16    return vyhra;
17 }
18
19 double odvod_statu()
20 {
21     return odvod;
22 }
23
24 double dan()
25 {
26     return (ucet > 0.0) ? 0.2 * ucet : 0.0;
27 }
28
29 double zisk()
30 {
31     return (ucet > 0.0) ? 0.8 * ucet : ucet;
32 }
```

Úloha 3-13

Uvažujme nasledujúci program, v dvoch zdrojových súboroch **hlavny.c** a **dalsi.c**. Hlavná funkcia **main()** 5-krát volá funkcie **prva()**, **druha()** a **tretia()** a nakoniec vypíše hodnotu premennej **pocet**.

Koľko rôznych hodnôt môže program vypísať ak môžeme do chýbajúcich miest doplniť do každého najviac jedno slovo (postupnosť znakov bez medzier)?

Doplňujúca úloha: Je možné program doplniť tak, aby sa dostal na riadok **dalsi.c:18** a vypísal **BINGO**?

1	// hlavny.c	1	// dalsi.c
2	#include <stdio.h>	2	#include <stdio.h>
3		3	
4	void druha();	4	void druha()
5	void tretia(int pocet);	5	{
6	void prva()	6	int pocet;
7	{	7	pocet++;
8	int pocet;	8	}
9	pocet = 0;	9	
10	}	10	void tretia(int pocet)
11	int pocet;	11	{
12		12	{
13	int main(void)	13	int pocet;
14	{	14	pocet += 2;
15	int n = 5;	15	}
16	while (n --> 0)	16	
17	{	17	if (pocet == 11)
18	prva();	18	printf("BINGO\n", pocet);
19	druha();	19	
20	tretia(pocet);	20	{
21	pocet++;	21	int pocet;
22	}	22	pocet = 3;
23	printf("%d\n", pocet);	23	}
24	return 0;	24	}
25	}		

3.4 Smerník na funkciu

Funkcia je v programe uložená v pamäti podobne ako premenné, pričom uložené údaje sú inštrukcie, ktoré po zavolaní funkcia vykonáva. Jazyk C však programátorovi neumožňuje vo vykonávanom programe „hodnotu funkcie“ (jej zdrojový kód) upravovať. Nie je možné vo vykonávanom programe vytvoriť novú funkciu, uložiť príkazy do premennej, nie je možné ani priamo poslať samotný kód funkcie ako argument do funkcie, resp. vrátiť kód funkcie ako návratovú hodnotu. Niektoré iné programovacie jazyky, ktoré podporujú tzv. funkcionálne programovanie tieto možnosti poskytujú.

Programovací jazyk C umožňuje používať len smerník na funkciu – teda samotná implementácia (príkazy) funkcie je síce uložená v pamäti ale program ju pri vykonávaní nemôže upravovať ani inak s ňou manipulovať. V programe je možné len odovzdať smerník na túto (vopred danú) implementáciu a teda možno vytvoriť akési všeobecné príkazy alebo funkcie, ktoré sú implementované bez ohľadu

na konkrétnu implementáciu. Môžeme takto vytvoriť „všeobecné“ funkcie, ktoré pri vykonávaní zavolajú pomocné funkcie prostredníctvom príslušného smerníku na funkciu, ktorý dostali ako argument. Túto všeobecnú implementáciu potom možno použiť s rôznymi implementáciami pomocných funkcií predpísaného typu. Využitie tohto prístupu znižuje previazanosť medzi rôznymi časťami kódu.

Využitím smerníka na funkciu je napr. možné v jazyku C implementovať zaužívané návrhové vzory Strategy (stratégia) a Observer (pozorovateľ), resp. posielanie správ vzorom Publish-subscribe.

Uvažujme nasledujúcu všeobecnú funkciu **tabulka()**, ktorá pre danú aritmetickú operáciu vypíše výsledkovú tabuľku veľkosti do **n**, pričom čísla sú zarovnané na **k** miest:

1	#include <stdio.h>	
2		
3	int sucet(int a, int b)	
4	{ return a + b; }	
5	int sucin(int a, int b)	
6	{ return a * b; }	
7		
8	void tabulka(int n,	
9	int(*op)(int, int))	
10	{	
11	int i, j;	
12	for (i = 0; i <= n; i++)	
13	{	
14	for (j = 0; j <= n; j++)	
15	printf("%2d ", op(i, j));	
16	printf("\n");	
17	}	
18	}	
19		
20	int main(void)	
21	{	
22	tabulka(5, sucet);	
23	tabulka(7, sucin);	
24	return 0;	
25	}	

	Výstup:
1	0 1 2 3 4 5
2	1 2 3 4 5 6
3	2 3 4 5 6 7
4	3 4 5 6 7 8
5	4 5 6 7 8 9
6	5 6 7 8 9 10
7	0 0 0 0 0 0 0 0
8	0 1 2 3 4 5 6 7
9	0 2 4 6 8 10 12 14
10	0 3 6 9 12 15 18 21
11	0 4 8 12 16 20 24 28
12	0 5 10 15 20 25 30 35
13	0 6 12 18 24 30 36 42
14	0 7 14 21 28 35 42 49
15	

Funkcia vo vnorenom cykle prejde všetky dvojice čísel pre **i = 0, ..., n** a **j = 0, ..., n**: do **i**-teho riadku vypíše výsledok vykonania operácie **op** (typu **int(*) (int, int)**) s číslami **i** a **j** (**i+j** pre **sucet**, resp. **i*j** pre **sucin**). Funkciu môžeme volať s ľubovoľnou funkciou **op**, ktorá spĺňa typové ohraňenie (funkcia musí vracať **int**,

a vstupné argumenty musia byť dva, oba typu `int`). Môžeme ju teda napr. ľahko rozšíriť o `rozdiel()`: `int rozdiel(int a, int b) { return a - b; }`

Použitie smerníku na funkciu zjednodušuje použitie všeobecných algoritmov. Napr. algoritmus implementovaný vo funkcii `bisekcia()` na numerické hľadanie koreňa spojitkej funkcie na intervale metódou pólania intervalu (tzv. bisekcie) je uvedený v nasledujúcom programe:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  double f1(double x) { return 5 * x*x + x - 1; }
5  double f2(double x) { return 3 * x*x*x - 2 * x*x + x - 5; }
6  double f3(double x) { return 2 * x*x*x*x - 5*x*x*x + x - 1; }
7
8  double bisekcia(double(*fx)(double), double a, double b)
9  {
10     double mid = (a + b) / 2.0;
11     if (fabs(fx(mid)) < 0.000001 || fabs(b - a) < 0.000001)
12         return mid;
13     if (fx(a) * fx(mid) < 0.0)
14         return bisekcia(fx, a, mid);
15     return bisekcia(fx, mid, b);
16 }
17 int main(void)
18 {
19     printf("Koren: |f(%lf)| < 0.000001\n", bisekcia(f1, -100, 100));
20     printf("Koren: |f(%lf)| < 0.000001\n", bisekcia(f2, -100, 100));
21     printf("Koren: |f(%lf)| < 0.000001\n", bisekcia(f3, -100, 100));
22     return 0;
23 }
```

Algoritmus nájde pre spojitú funkciu reálnej premennej, ktorá na danom intervale má koreň, jej koreň. Všeobecný algoritmus je ľahko použiteľný pre ľubovoľnú (vyhovujúcu) funkciu zadaného typu (`double(*) (double)`), ktorú odovzdáme funkcii ako argument. Výstup programu je nasledovný (dodajme, že pri reálnom použití tohto algoritmu je ešte potrebné skontrolovať funkčnú hodnotu v nájdenom „koreni“ pre prípad, že funkcia na danom intervale koreň nemá, v tomto prípade nájdené hodnoty sú korene):

```

1  Koren: |f(-0.558257)| < 0.000001
2  Koren: |f(1.342781)| < 0.000001
3  Koren: |f(-0.639134)| < 0.000001
```

V bežných programoch sa asi najčastejšie používajú smerníky na funkciu pri usporiadaní hodnôt prvkov. Pri usporadúvaní N prvkov sa implementuje len pomocná porovnávacía funkcia na porovnanie hodnôt dvoch prvkov, čím si programátor môže implementovať akékoľvek želané usporiadanie: podľa ľubovoľného atribútu prvku ako aj ľubovoľný smer: od najmenšieho po najväčšie alebo opačne. Pomocná funkcia na porovnanie dvoch prvkov potom vstupuje do všeobecného algoritmu usporadúvania pre N prvkov, ktorý je implementovaný nezávisle od cieľového usporiadania prvkov.

Ukážka tohto postupu je uvedená v nasledujúcej kapitole v časti 4.1.

Úloha 3-14

Napíšte definície smerníkov na nasledujúce typy funkcií:

- Smerník na funkciu **min3()** (Úloha 3-1), ktorej vstupné argumenty sú tri celé čísla (typ **int**) a výstupom je ich minimum,
- Smerník na funkciu **priemer()**, ktorej vstupné argumenty sú dve celé čísla (typ **int**) a výstupom je priemer týchto čísel (desatinné číslo),
- Smerník na funkciu **vymen()**, ktorá vymení hodnoty dvoch premenných **int**,
- Smerník na funkciu **kvadraticka_rovnica()**, ktorá pre koeficienty a , b , c (typ **double**) určí korene kvadratickej rovnice $ax^2 + bx + c = 0$,
- Smerník na funkciu **vypocitaj()**, ktorá vykoná binárnu operáciu (funkcia nad celými číslami) nad dvomi vstupnými celými číslami. Napr. pre operáciu **int sucet(int,int)** je výsledok volania funkcie **vypocitaj(sucet, 3, 5)** hodnota 8, ale pre **int sucin(int,int)** je výsledok volania funkcie **vypocitaj(sucin, 3, 5)** hodnota 15.

Úloha 3-15

Doplňte chýbajúce časti programu na výpočet hodnoty funkcií a ich derivácií v danom bode. V programe sú použité funkcie **f1()**, **f2()** a **deriv()**. Parametrom funkcie **f1()**, resp. **f2()** je reálne číslo x , návratovou hodnotou je funkčná hodnota funkcie **f1()** resp. **f2()** v bode x .

Funkcie sú tvaru: $f_1(x) = \frac{e^x}{|x|+1}$ a $f_2(x) = x * \sin x$. Funkcia **deriv()** vypočíta numerickú deriváciu funkcie **f()** v bode **x** podľa vzťahu: $f'(x) = \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$, kde ε je presnosť výpočtu. V programe zvolte vhodnú hodnotu ε . Program načíta reálne čísla **r1** a **r2** (typ **double**) a vypíše tabuľku hodnôt funkcií **f1()** a **f2()** a ich derivácií v intervale **<r1, r2>** s krokom **STEP**.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define STEP 0.2
5
6  double deriv(_____, _____)
7  {
8      return _____;
9  }
10 double f1(double x)
11 {
12     return _____;
13 }
14 double f2(double x)
15 {
16     return _____;
17 }
18 int main()
19 {
20     double r1, r2, x;
21     scanf("%lf %lf", &r1, &r2);
22     printf("  x    f1(x)  f1'(x)  f2(x)  f2'(x)\n");
23     for (x = r1; x < r2 + 0.000000001; x += STEP)
24         printf("%5.21f  %5.21f  %5.21f  %5.21f  %5.21f\n",
25             x, _____, _____, _____, _____);
26     return 0;
27 }
```

Úloha 3-16

Doplňte chýbajúce časti programu tak, aby hlavná funkcia **main()** na výstup vypísala: čísla od 0 do 100 (riadok 56), párne čísla spomedzi čísel od 0 do 100 (riadok 57) a čísla obsahujúce cifru 3 spomedzi čísel od 0 do 100 (riadok 58).

Typ smerníku na funkciu, ktorý je v programe použitý (typ **input**), pomenujte pomocou príkazu **typedef** v riadku 3.

1	#include <stdio.h>	31	int obsahuje3(int *cislo)
2		32	{
3	typedef _____;	33	int i, _____;
4		34	while (vsetko(_____))
5	int vsetko(int *cislo)	35	for (_____)
6	{	36	if (k % 10 == 3)
7	_____ int i = 0;	37	{
8		38	_____;
9	if (i <= 100)	39	return _____;
10	{	40	}
11	*cislo = _____;	41	return _____;
12	return 1;	42	}
13	}	43	
14	i = 0;	44	void print(input fn)
15	return 0;	45	{
16	}	46	if (_____)
17		47	return;
18	int parne(_____)	48	int i;
19	{	49	while (_____)
20	int i;	50	printf("%d ", _____);
21	while (vsetko(_____))	51	printf("\n");
22	if (_____)	52	}
23	{	53	
24	*cislo = i;	54	int main(void)
25	return _____;	55	{
26	}	56	// vsetky od 0 do 100
27		57	print(vsetko);
28	return _____;	58	// parne od 0 do 100
29	}	59	print(parne);
30		60	// obsahuje 3
		61	print(obsahuje3);
		62	return 0;
		63	}

Funkcia **vsetko()** slúži ako zdroj čísel pre ostatné funkcie: každé nasledujúce úspešné volanie funkcie **vsetko()** by malo vrátiť návratovú hodnotu 1 a naplniť číslo v argumente ďalším číslom **i** (postupne hodnotami od 0 do 100), pričom nasledujúce volanie po dosiahnutí čísla 100 vráti návratovú hodnotu 0, ktorá signalizuje vyčerpanie všetkých čísel, a inicializuje číslo **i** opäť na počiatočnú hodnotu 0.

Doplňujúca otázka: Ako by sme upravili funkciu **obsahuje3()** ak by sme chceli, aby volanie **print(obsahuje3)** vypísalo len párne čísla, ktoré obsahujú cifru 3?

Doplňujúca úloha: Napíšte funkciu **prvocisla()** ako jednoduchý iterátor, ktorý vyfiltruje prvočísla z funkcie **vsetko()**. Volanie **print(prvocisla)** vypíše postupne všetky prvočísla do 100.

Riešenia úloh (Kapitola 3)

Úloha 3-1 (riešenie)

Uvedený zdrojový kód nie je správne riešenie úlohy preto, že v podmienkach neuvažuje rovnosť čísel. Napr. v prípade, že čísla **a** a **b** sú obe najmenšie, ale číslo **c** je väčšie, tak podmienky nebudú splnené a funkcia vráti hodnotu **c** (ktorá nie je najmenšia), napr. pre čísla **a=3, b=3, c=7**. Opravený kód vyzerá takto, ak **a** je menšie ale rovné ako **b** a **c**, vráti **a**, podobne pre **b**, inak musí byť najmenšie **c**:

```
1 int min3(int a, int b, int c)
2 {
3     if (a <= b && a <= c)
4         return a;
5     if (b <= a && b <= c)
6         return b;
7     return c;
8 }
```

Je táto funkcia efektívna? Efektívnosť môžeme merať rôznymi spôsobmi, najčastejšie ako náročnosť vykonania pre počítač, čo môžeme merať počtom inštrukcií. V tomto programe použijeme až 4 porovnania: ak je najmenšie číslo **a** tak program vykoná 2 porovnania, ale v opačnom prípade (ak sú najmenšie **b** alebo **c**) vykoná 4 porovnania.

Skúsme teraz alternatívny postup, vhodnejší ak by sme určovali minimum pre ľubovoľný (vopred neznámy) počet čísel. Postupne budeme spracúvať čísla a v premennej **a** si budeme priebežne uchovávať najmenšie zo spracovaných čísel:

```
1 int min3(int a, int b, int c)
2 {
3     if (a > b)
4         a = b;
5     if (a > c)
6         a = c;
7     return a;
8 }
```

Na začiatok (riadok 2) uvažujeme len jedno číslo **a**, ktoré je zároveň aj minimum. Uvažujme, teraz ďalšie číslo **b** (riadky 3-4), ak doterajšie minimum (**a**) je väčšie ako nové číslo (**b**), uložíme si nové minimum (**b**). Pokračujeme s ďalším číslom (**c**) podobne (riadky 5-6), nakoniec funkcia vráti hodnotu minima (**a**). Tento postup použije síce

menej porovnaní (len 2) ale viac priradení. Priradenie je „lacnejšia“ operácia, v zmysle, že počítač zvyčajne potrebujeme menej zdrojov na jej vykonanie, vyžaduje však, istým spôsobom, pomocnú pamäť.

Vráťme sa ešte k predchádzajúcemu riešeniu s 4 porovnaniami a skúsme odstrániť prebytočné porovnania. V riadku 5 v podmienke **(b <= a && b <= c)** vykonávame už opakované porovnanie **a** a **b**. Rozdeľme teda zložené podmienky do jednoduchých a pokúsme sa už raz zistený výsledok porovnania využiť, dostávame tento kód:

```
1 int min3(int a, int b, int c)
2 {
3     if (a <= b)
4     {
5         // a je kandidat
6         if (a <= c)
7             return a;
8         return c;
9     }
10    else
11    {
12        // b je kandidat
13        if (b <= c)
14            return b;
15        return c;
16    }
17    return c;
18 }
```

V každom prípade, či je výsledok (minimum) **a**, **b** alebo **c** vykonáme práve 2 porovnania. Menej ako použitím 2 porovnaní to určite nepôjde, pretože s jedným porovnaním nedokážeme rozlíšiť vzájomnú polohu troch čísel. Na počet vykonaných inštrukcií je to najefektívnejší postup.

Analizujme ešte postup, ktorý využíva určenie minima z dvoch čísel (**min2**):

```
1 #define min2(a,b) (((a)<(b))?(a):(b))
2
3 int min3(int a, int b, int c)
4 {
5     return min2(min2(a, b), c);
6     // a < ( b < c ? b : c ) ? a : ( b < c ? b : c )
7 }
```

Na výpočet **min2** môžeme použiť makro (riadok 1), ktoré sa pred kompiláciou rozvinie (riadok 6), alebo aj funkciu, princíp zostane rovnaký: v priemere urobíme viac (2,666...) porovnaní: 3 porovnaní ak výsledok je **a** alebo **b**, 2 porovnaní ak výsledok je **c**.

V praxi na modernom procesore s využitím moderného kompilátora bude najefektívnejší postup s čo najjednoduchšími podmienkami: postup využívajúci dve priradenia s dvoma porovnaniami, pretože hlboké vetvenia pri ostatných postupoch spôsobujú väčšiu záťaž pri tzv. predpovedi vetvení, ktoré si vyžaduje pri vykonávaní od procesora najväčšiu námahu.

Úloha 3-2 (riešenie)

Prvý krok pri návrhu funkcie je určiť hlavičku: vstupné a výstupné argumenty. Vstupné argumenty určíme zo zadania: reálne číslo **x** (typ **double**) a celé číslo **n** (typ **int**), výstupom bude hodnota mocniny **xⁿ** (typ **double**). Hlavička teda bude takáto (riadok 1). Vo funkcii si deklarujeme pomocnú premennú pre výsledok (**v**) a v cykle jej hodnotu **n**-krát vynásobíme hodnotou **x**. Počiatočná hodnota premennej **v** bude 1 (neutrálny prvok operácie násobenia):

1	double xn(double x, int n)
2	{
3	double v = 1.0;
4	int i;
5	for (i = 0; i < n; i++)
6	v = v*x;
7	return v;
8	}

Týmto postupom v každej iterácii cyklu zvýšime mocninu výsledku o 1: z **xⁱ⁻¹** na **xⁱ**.

Existuje aj **pokročilejší postup**, ktorý efektívnejšie (rýchlejšie) zvyšuje mocninu: v jednom kroku ju zdvojnásobí. Na zdvojnásobenie mocniny musíme hodnotu **v** vynásobiť aktuálnou hodnotou **v**, a teda napr. z hodnoty **v=x⁸** jedným násobením (**v=v*v**) vznikne hodnota **x¹⁶**. Týmto spôsobom by sme vedeli rýchlo určiť hodnoty **xⁿ** pre exponent **n**, ktorý je mocninou čísla 2, teda **n = 1, 2, 4, 8, 16, 32, 64, 128, 256, ...** Algoritmus pre výpočet **xⁿ** pre všeobecné hodnoty **n** musí ešte hodnoty

mocnín pre vhodné exponenty mocnín dvojky vzájomne vynásobiť, aby sme dostali výsledok pre ľubovoľné n , napr. pre x^{19} vynásobíme: $x^1 * x^2 * x^{16} = x^{19}$, kde 1, 2 a 16 sú mocniny dvojky, pričom hodnoty x^1 , x^2 a x^{16} už vieme efektívne určiť predchádzajúcim postupom. Na efektívny výpočet x^n pre ľubovoľné n treba vynásobiť tie hodnoty x^i , ktoré sa nachádzajú v binárnom zápise čísla n ($19 = 1 + 2 + 16 = 10011_2$). Výsledný program:

```

1 double xn(double x, int n)
2 {
3     double v = 1.0, s = x;
4     while (n > 0)
5     {
6         if (n % 1 == 0)
7             v *= s;
8         s *= s;
9         n /= 2;
10    }
11    return v;
12 }
```

Na určenie hodnoty x^n pre veľké čísla n tento program potrebuje vykonať rádovo $\log n$ operácií, čo je rádovo menej ako predchádzajúci postup s využitím lineárneho cyklu, ktorý potreboval rádovo n operácií.

Úloha 3-3 (riešenie)

Najskôr musíme určiť hlavičku funkcie: vstupné a výstupné argumenty. Vstupné argumenty vyplývajú zo zadania, výstup bude hodnota ďalšieho prvočísla (typ `int`), resp. prvočíslo nemôže byť záporné a mohli by sme použiť typ `unsigned int`, ale ak na vstupe je `int`, tak to zachováme. Hlavička teda bude takáto:

```
int dalsie_prvocislo(int n)
```

prvočísla sú medzi celými číslami pomerne husto, a preto ako vhodný postup bude pre hodnoty postupne od $n+1$, ... zisťovať, či číslo je prvočíslo. Ak je, máme výsledok. Pre zistenie či je nejaké číslo prvočíslo môžeme použiť funkciu `je_prvocislo()`, ktorú sme si predstavili v kapitole 0. Pre vstupné číslo x , funkcia `je_prvocislo()` postupne vyskúša celé čísla $i=2, \dots, \sqrt{n}$ (vrátane) či niektoré delí n bezo zvyšku. Ak

áno, **x** je zložené číslo a nie je prvočíslo, inak (ak sa nenájde deliteľ **i** čísla **x**), tak **x** je prvočíslo.

Program by mohol (podľa vyššie uvedeného postupu) vyzeráť takto:

```
1 int je_prvocislo(int x)
2 {
3     int i;
4     for (i = 2; i*i <= x; i++)
5         if (x % i == 0)
6             return 0;
7     return 1;
8 }
9
10 int dalsie_prvocislo(int n)
11 {
12     while (!je_prvocislo(++n));
13     return n;
14 }
```

V tomto programe je však chyba. Pokúste sa ju nájsť!

Akým postupom sa dá nájsť prípadná chyba? Musíme preskúmať, či vytvorené funkcie pre každý možný vstup vrátia správny výsledok. Skúsime napr. pre vstup **n=20**, **dalsie_prvocislo(20)**: postupne sa zavolajú funkcie **je_prvocislo(21)**, **je_prvocislo(22)** a **je_prvocislo(23)**. Posledné uvedené volanie vráti 1 a teda cyklus (riadok 12) sa ukončí a vráti sa **n** (23). Vyzerá to dobre, kde teda môže byť chyba? Aké ešte iné hodnoty môžu do funkcie vstúpiť?

Možné hodnoty na vstupe sú ohraničené dátovými typmi, v tomto prípade **n** je typu **int**, preto na vstupe môže prísť napr. aj záporné číslo; všimnime si, že takýto vstup (záporné číslo) plne vyhovuje zadaniu. Čo ale vráti naša funkcia pre záporné číslo? Napr. **n = -20**. Zavoláme **je_prvocislo(-19)**, ktoré vráti 1 a teda výsledok **dalsie_prvocislo(-20)** bude -19. Podobný problém nastane aj pre **n = 0**, výsledok bude 1, lebo pre číslo **x = 1** nenájde v cykle (riadky 4-6) deliteľov. Keďže prvočísla sú definované ako kladné celé čísla tak tento problém môžeme vyriešiť pri určení návratovej hodnoty (riadok 7), pridáme podmienku (**x >= 2**): **return x >= 2;**

Úloha 3-4 (riešenie)

Použijeme bežne zaužívaný postup výpočtu kvadratickej rovnice využitím vzorca:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hodnotu pod odmocninou ($b^2 - 4ac$) nazývame diskriminant, podľa ktorého vieme rozlíšiť typ riešení rovnice. Vstupné argumenty sú určené zadáním úlohy. Výstupné argumenty zodpovedajú výsledkom riešenia rovnice: celkovo môžu byť až dva korene rovnice, preto výstupné argumenty musia byť tri: hodnota prvého koreňa (**x1**, typ **double**), hodnota druhého koreňa (**x2**, typ **double**) a informácia o počte platných riešení (ako návratová hodnota, typ **int**, hodnoty 0, 1 alebo 2).

V zdrojovom kóde je diskriminant označený premennou **d**. V riadku 3 určíme hodnotu diskriminantu. V prípade, že je diskriminant záporný (**d < 0.0**), tak nie je možné odmocninu vypočítať a rovnica nemá riešenie (návratová hodnota 0). Ak je nezáporný, vypočítame odmocninu (matematické funkcia **sqrt()**) a určíme riešenia. Riešenia sú dve v prípade ak je diskriminant kladný (nenulový). Preto podmienkou v riadku 8 zistíme, či je diskriminant nulový a ak áno, končíme s jedným riešením (návratová hodnota 1). Pri overení či je diskriminant nulový je vzhľadom na presnosť výpočtov dobré namiesto toho overiť, či je príliš malý (už vieme že nie je záporný). Ak je dostatočne veľký (väčší ako ϵ) môžeme určiť aj druhé riešenie (návratová hodnota 2). Zvyšok programu je priamo doplnený podľa vzorca na riešenie kvadratickej rovnice:

```
1  int kvadraticka_rovnica(double a, double b, double c,  
                             double *x1, double *x2)  
2  {  
3      double d = b*b - 4*a*c ;  
4      if (d < 0.0)  
5          return 0;  
6      d = sqrt(d);  
7      *x1 = (-b + d) / (2*a) ;  
8      if (d < 0.00000001)  
9          return 1;  
10     *x2 = (-b - d) / (2*a) ;  
11     return 2;  
12 }
```

Riešenie dopĺňujúcej úlohy: pre hodnotu $a = 0$ nie je možné realizovať výpočet vzťahom pre výpočet kvadratickej rovnice, lebo $bx+c = 0$ už nie je kvadratická rovnica. Dostávame lineárnu rovnicu, ktorú riešime bežnou úpravou: $x = \frac{-c}{b}$ (výpočet je možný pre nenulové b). Ak je b nulové, tak dostávame rovnicu $c = 0$, ktorá má pre koeficient c s hodnotou 0 nekonečne veľa riešení (x) a pre koeficient $c \neq 0$ nemá riešenie. Do výslednej funkcie preto doplníme nasledujúce príkazy:

```

1  if (fabs(a) < 0.000000001) // b*x + c = 0
2  {
3      if (fabs(b) < 0.000000001)
4      {
5          if (fabs(c) < 0.000000001)
6              return -1; // nekonecne vela
7              return 0; // nema riesenie
8      }
9      *x1 = -c / b;
10     return 1;
11 }

```

Úloha 3-5 (riešenie)

Pri výpočte použijeme zaužívaný postup využívajúci reprezentáciu priamok cez všeobecné rovnice priamky tvaru $ax + by + c = 0$. Funkcia `rovnica()` na výpočet rovnice priamky je správna, ale pri jej volaní vo funkcii `priesečník_priamok()` sú argumenty uvedené v chybnom poradí (x_1, x_2, y_1, y_2), správne poradie je (x_1, y_1, x_2, y_2). Ďalšie chyby sú vo výpočte súradníc, ktoré sú vymenené a nie sú normalizované hodnotou d .

```

9  int priesečník_priamok(double x1, double y1, double x2, double y2,
10                        double x3, double y3, double x4, double y4,
11                        double *x, double *y)
12 {
13     double a1, b1, c1, a2, b2, c2;
14     rovnica(x1, y1, x2, y2, &a1, &b1, &c1);
15     rovnica(x3, y3, x4, y4, &a2, &b2, &c2);
16
17     double d = a1*b2 - b1*a2;
18     if (fabs(d) < 0.000000001)
19         return 0;
20     *x = (b1*c2 - c1*b2) / d;
21     *y = (c1*a2 - a1*c2) / d;
22     return 1;
23 }

```

Úloha 3-6 (riešenie)

Volanie **a(0)** musí vyhodnotiť výraz v riadku 6: najskôr vyhodnotí návratovú hodnotu z **c(0)**. Hodnotu **c(0)** určíme z riadku 4 ako **d(0)+1**, keďže **d(0)** je $3*0 = 0$, tak **c(0)** = $3*0+1 = 1$. V riadku 6 následne dosadíme hodnotu **c(0)** ako vstupný argument do funkcie **b()**. Vyhodnotíme **b(1)** v riadku 5: najskôr určíme **d(1)** = $3*1 = 3$, dosadíme do **c()**: **c(3)** = **d(3)** + 1 = $3*3 + 1 = 10$. Preto **b(1)** = $2*10-3 = 17$. Nakoniec dopočítame hodnotu **a(0)** = $2*b(1)+1 = 2*17+1 = 35$. Program vypíše hodnotu 35.

Úloha 3-7 (riešenie)

Pri návrhu každej rekurzívnej funkcie musíme navrhnúť nielen vstupné a výstupné argumenty ale aj **rekurzívny krok výpočtu**, ktorý určí výsledok funkcie pre hodnoty vstupných argumentov na základe rekurzívneho volania ako aj **terminálnu podmienku**, resp. **základný prípad**, pri ktorom je výsledná hodnota určená bez použitia rekurzívneho volania.

V tomto prípade si analýzou iteratívnych programov na určenie ciferného súčtu, ktoré sú uvedené v zadaní, môžeme všimnúť hraničné podmienky: výpočet ciferného súčtu **x** nepokračuje ak hodnota **x** = 0, kedy je výsledok 0. V ostatných prípadoch (**x** > 0), v cykle postupne pripočítavame poslednú cifru k výsledku a pokračujeme so zmenšeným číslom **x / 10**.

Zodpovedajúci rekurzívny program je takýto:

```
1 int ciferny_sucet(int x)
2 {
3     if (x == 0)
4         return 0;
5     return x % 10 + ciferny_sucet(x / 10);
6 }
```

Návratová hodnota volania **ciferny_sucet(x / 10)** predstavuje ciferný súčet cifier čísla **x** bez poslednej cifry. K tomuto výsledku pripočítame (riadok 5) hodnotu poslednej cifry (**x % 10**), čím určíme celkový výsledok pre hodnotu **x**. Rekurzívne volania budú pokračovať pre znižujúce sa hodnoty čísla **x**, a preto na zabránenie nekonečného rekurzívneho vnorenia musíme zabezpečiť, aby sme pre nejaké malé hodnoty **x** výsledok vo funkcii určili bez rekurzívneho volania. Základný prípad

(riadky 3 až 4) v tomto prípade môže byť $x = 0$ (výsledok 0) alebo aj jednociferné čísla ($x \leq 9$) a výsledok x .

Úloha 3-8 (riešenie)

Analyzujeme iteratívne riešenie: v cykle vytvára hodnotu binárneho čísla **vysledok**, ktoré nakoniec vypíše. Výsledné binárne číslo je v programe reprezentované ako celé číslo (typu **long long**), ktorého hodnotu formálne počítame v desiatkovej sústave, preto pre zápis binárnej cifry 1 na i -tom ráde musíme k číslu pripočítať 10^i . Obmedzenie vstupnej hodnoty x do 1 000 000, zodpovedá použitému dátovému typ, pretože zápis väčších čísel v binárnej sústave si vyžaduje výsledok, ktorý by už dátový typ **long long** nedokázal reprezentovať. Program postupne delí vstupné číslo x dvomi, pričom zvyšok po delení dvomi priebežne ukladá na začiatok výsledného čísla, v jednom kroku pridá najvyšší rád čísla.

Rekurzívna funkcia pre vstupnú hodnotu x teda musí vypísať zvyšok po delení čísla x dvomi až potom ako vypíše binárnu reprezentáciu zostávajúceho čísla ($x/2$). Základný prípad, ktorý na riešenie nevyžaduje rekurzívne volanie, je jednociferné binárne číslo x , čiže ak $x = 0$ alebo $x = 1$.

Výsledná rekurzívna funkcia potom vyzerá takto:

1	void print_bin(int x)
2	{
3	if (x > 1)
4	print_bin(x / 2);
5	printf("%d", x % 2);
6	}

Rekurzívne vnorenie prebieha len v prípade ak x nie je jednociferné binárne číslo ($x > 1$). Potom ako vypíše binárnu reprezentáciu vyšších rádov ($x/2$) dopíše aj poslednú cifru ($x\%2$) binárneho zápisu x . Všimnime si, že toto rekurzívne riešenie nemá obmedzenie na vstupné číslo x do 1 000 000. Je to spôsobené tým, že zásobník volaní funguje ako akási pamäť poslednej cifry ($x\%2$) predtým ako ju vypíšeme (pretože rekurzívne vnorenie ($x/2$) prebieha tesne pred výpisom **printf()**). Z formálneho hľadiska teda rekurzívne riešenie vyžaduje viac pomocnej pamäte, čím však získame všeobecnejšie riešenie, ktorého kód je jednoduchší a prehľadnejší.

Úloha 3-9 (riešenie)

Iteratívny postup pracuje podobne ako pri iteratívnom prevode čísla na binárny zápis s využitím pomocnej premennej (Úloha 3-8): do pomocnej premennej **otocene** si funkcia vypočíta otočené číslo **x**, ktoré nakoniec porovná s hodnotou **x**. Ak sa rovnajú (otočené **x** má rovnakú hodnotu ako **x**) tak **x** je palindróm, inak nie je. Doterajší prístup k vytvoreniu rekurzívneho riešenia, v ktorom sme používali zásobník volaní ako istú formu pamäte musíme rozšíriť o dodatočný argument, ktorý bude predstavovať výslednú (otočenú) hodnotu. Pri otočení čísla postupujeme v prípade rekurzívneho riešenia rovnakým spôsobom ako v iteratívnom prípade (funkcia zo zadania): v jednom kroku poslednú cifru (**x%10**) pripojíme k výslednému otočenému číslu sprava a pokračujeme pre **x** bez poslednej cifry (**x/10**). Požadovaná funkcia **palindrom()**, teda najprv využitím pomocnej funkcie **otoc()** určí otočenú hodnotu **x**, ktorú potom porovná s pôvodnou hodnotou **x**:

```
1 int otoc(int x, int otocene)
2 {
3     if (x == 0)
4         return otocene;
5     return otoc(x / 10, 10 * otocene + (x % 10));
6 }
7
8 int palindrom(int x)
9 {
10     return otoc(x, 0) == x;
11 }
```

Alternatívne riešenie (bez pomocného argumentu): Vyššie uvedená implementácia je priamy prepis iteratívneho riešenia, ktoré využije dodatočný argument funkcie ako pamäť pre výpočet, pričom na konci výpočtu vráti hodnotu tohto argumentu ako výstup (návratovú hodnotu).

Dalo by sa to implementovať aj bez použitia dodatočného argumentu? Zamyslime sa nad tým, čo by sa stalo, keby sme vo funkcii **otoc()** nepoužili druhý argument. Vo všetkých funkciách by sme museli výpočet realizovať len so znalosťou vstupnej hodnoty (**x**), bez znalosti priebežného stavu výpočtu v pomocnom argumente. V prípade rekurzcie teda musíme v jednom kroku rozhodnúť či posledná

cifra x zodpovedá prvej cifre x . V ďalšom kroku totiž výpočet pokračuje pre $x/10$, čo je číslo x bez poslednej cifry. Poslednú cifru vieme určiť ľahko, problém je s určením prvej cifry, ktorá sa nedá zistiť jednoduchou operáciou.

Navrhujeme preto pomocnú (rekurzívnu) funkciu **prva_cifra()**, ktorá určí prvú cifru v čísle x . Ako bude funkcia pracovať? Ak je x jednociferné, výsledok je priamo hodnota x . Ak je x viacciferné ($x \geq 10$), tak výsledok je taký istý ako výsledok pre x bez poslednej cifry ($x/10$). Na základe tejto funkcie navrhujeme funkciu **palindrom()**, ktorá skontroluje či prvá a posledná cifra sú rovnaké, ak áno pokračuje výpočet s číslom x bez prvej a poslednej cifry:

```
1 int prva_cifra(int x)
2 {
3     if (x < 10)
4         return x;
5     return prva_cifra(x / 10);
6 }
7 int palindrom(int x)
8 {
9     if (x < 10)
10        return 1;
11    if (x % 10 != prva_cifra(x))
12        return 0;
13    x =           ;
14    return palindrom(x / 10);
15 }
```

Nedoriešená otázka zostáva, ako upraviť hodnotu x , aby sme z neho odstránili prvú a poslednú cifru. Odstrániť poslednú cifru je triviálne: $x = x - x\%10$; Aj keď poznáme hodnotu prvej cifry (napr. cifra 3 v čísle 3214), stále nepoznáme celkovú hodnotu, ktorú treba od premennej x pre odstránenie prvej cifry odpočítať (v príklade to je 3000). Potrebujeme k tomu poznať hodnotu najvyššieho rádu (v príklade 1000).

Navrhujeme rekurzívnu funkciu **najvyssi_rad()**, ktorá pracuje analogicky ako funkcia **prva_cifra()**:

```
1 int najvyssi_rad(int x)
2 {
3     if (x < 10)
4         return 1;
5     return 10 * najvyssi_rad(x / 10);
6 }
```


Úprava čísla x v riadku 14 (vo funkcii **palindrom()**) je potom nasledovná:

14	<code>x -= prva_cifra(x) * najvyssi_rad(x);</code>
----	--

Teraz dokonca v riešení ani nepotrebujeme funkciu **prva_cifra()**, lebo na určenie prvej cifry postačuje poznať hodnotu rádu prvej cifry, ktorú potom určíme ako $x / \text{najvyssi_rad}(x)$. Zjednodušená funkcia **palindrom()** bez použitia funkcie **prva_cifra()** vyzerá takto:

8	<code>int palindrom(int x)</code>
9	<code>{</code>
10	<code>if (x < 10)</code>
11	<code>return 1;</code>
12	<code>if (x % 10 != x / najvyssi_rad(x))</code>
13	<code>return 0;</code>
14	<code>x -= x / najvyssi_rad(x) * najvyssi_rad(x);</code>
15	<code>return palindrom(x / 10);</code>
16	<code>}</code>

Úloha 3-10 (riešenie)

Iteratívne riešenie tejto úlohy pre všeobecné hodnoty k nie je možné vytvoriť tak jednoducho ako boli použité vnorené cykly pre $k=3$ v ukážke v zadaní, pretože nie je možné do kódu (jednej funkcie) napísať premenlivý počet vnorených cyklov bez ohraničenia na hodnotu k .

Premenlivú hĺbku vnorenia je možné realizovať rekurzívne tak, že na každej úrovni rekurzívne bude cyklus, ktorý postupne pre hodnoty $1, \dots, n$ zavolá funkciu rekurzívne pre ďalšiu úroveň vnorenia. Rekurzívne volania nebudú pokračovať, ak sme už dosiahli vnorenie v požadovanej hĺbke k . Požadovanú hĺbku vnorenia budeme reprezentovať argumentom k , zdrojový kód vyzerá takto:

1	<code>int x;</code>
2	
3	<code>void print_var(int n, int k)</code>
4	<code>{</code>
5	<code>if (k == 0)</code>
6	<code>{</code>
7	<code>printf("%d\n", x);</code>
8	<code>return;</code>
9	<code>}</code>

```
10  int p;  
11  for (p = 1; p <= n; p++)  
12  {  
13      x = 10 * x + p;  
14      print_var(n, k - 1);  
15      x /= 10;  
16  }  
17 }
```

Všimnime si riadok 14, ktorý sa v cykle zavolá **n** krát: teda pri každom volaní rekurzívnej funkcie sa **n** krát vnoríme hlbšie (číslo úrovne **k-1**), až kým dosiahneme poslednú úroveň (riadok 5) **k = 0**, v ktorej vypíšeme hodnotu globálnej premennej **x** a ďalej sa nevňarame (ale vrátime späť do predchádzajúcej úrovne **k = 1**). Globálna premenná **x** teda musí postupne nadobudnúť všetky možné hodnoty variácií s opakováním **k**-tej triedy, napr. pre **n = 2** a **k = 3** to je: 111, 112, 121, 122, 211, 212, 221 a 222.

Hodnotu globálnej premennej **x** upravujeme v riadku 13, v ktorom pridáme do **x** cifru **p** sprava, a v riadku 15, v ktorom po vynorení z rekurzcie odstránime poslednú cifru **x**, ktorú sme tam pridali v riadku 13. Dôležité pozorovanie je, že efekt riadkov 13 a 15 je spolu neutrálny: príkaz v riadku 15 odstráni úpravu vykonanú v riadku 13 a vráti hodnotu **x** do stavu pred vykonaním riadku 13. Na základe toho vykonávaný cyklus v riadku 9 pre **p = 1, ..., n** vyskúša pridať postupne každú z cifier do výsledného čísla sprava a vnorí sa – čím pre konkrétne dosadenie cifry **p** na **k**-tej úrovni dosadí (a aj vypíše na výstup) všetky možné dosadenia zostávajúcich cifier. Postupným rekurzívnym vnáraním do úrovni **k-1, ..., 1** sa pre každú zo zostávajúcich cifier dosadia všetky možnosti cifier, pričom po kompletnom dosadení **k** cifier sa (v úrovni rekurzcie pre **k = 0**) hodnota **x** vypíše.

Riešenie doplňujúcej úlohy (1):

Bez použitia globálnych alebo statických premenných). Globálnu premennú **x** môžeme odstrániť tak, že do funkcie pridáme pomocný argument, v ktorom si budeme hodnotu priebežne prenášať, upravené riešenie je nasledujúce:

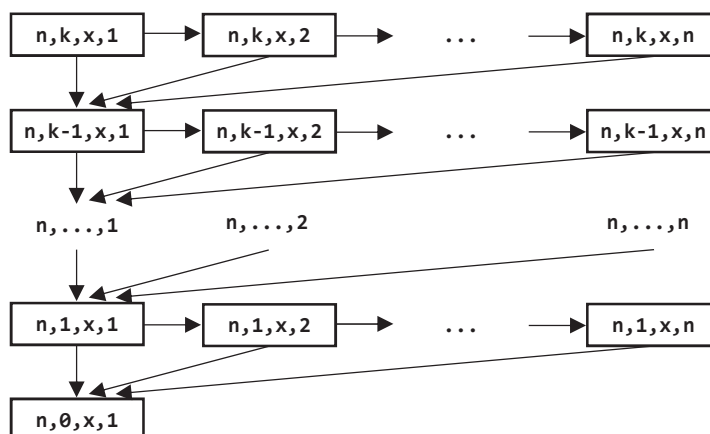
```
1 void var(int n, int k, int x)
2 {
3     if (k == 0)
4     {
5         printf("%d\n", x);
6         return;
7     }
8     int p;
9     for (p = 1; p <= n; p++)
10         var(n, k - 1, 10 * x + p);
11 }
12
13 void print_var(int n, int k)
14 {
15     var(n, k, 0);
16 }
```

Vzhľadom k doplneným argumentom sme pre zachovanie hlavičky pôvodnej funkcie museli výpočet presunúť do pomocnej funkcie **var()**. Alternatívne nahradenie globálnej premennej **x** by mohlo byť realizované tak, že by sme prenášali len smerník na premennú **x**, ktorá by bola lokálna premenná vo funkcii **print_var()**, čiže pri každom volaní rekurzie by sme upravovali tú istú hodnotu. Vyššie uvedené riešenie však zjednodušuje (odstraňuje) potrebu úpravy **x** do pôvodného stavu po vynorení a je preto prehľadnejšie a pre programátora jednoduchšie na implementáciu.

Riešenie doplňujúcej úlohy (2):

Bez použitia cyklov, globálnych alebo statických premenných). Ako môžeme z riešenia odstrániť použitie cyklu? Na každej úrovni rekurzie (pre konkrétne hodnoty **n** a **k**) musíme realizovať **n** krát vnorenie do hlbšej úrovne pre **p = 1, ..., n**. Namiesto cyklu to môžeme realizovať dodatočným argumentom **p**, ktorý bude reprezentovať koľkú iteráciu „cyklu pre **p**“ práve realizujeme.

Rekurzívne volanie pre znižujúce sa **k** si môžeme predstaviť ako vnáranie smerom dolu, rekurzívne volanie pre zvyšujúcu sa hodnotu **p** si môžeme predstaviť ako vnáranie smerom doprava, vzájomná súhra oboch vnáraní v jednej funkcii je znázornená na nasledujúcom obrázku (úpravy **x** neuvádzame):



Výsledný program je takýto. Pre každé vnáranie potrebujeme vo funkcii **var()** implementovať základný prípad, resp. terminálnu podmienku, pri ktorej vnáranie dosiahlo koniec a ďalej nepokračuje. Pre prvé vnáranie pre znižujúce sa **k** je základný prípad realizovaný v riadkoch 3 až 6 (ak **k** = 0). Pre druhé vnáranie pre zvyšujúce sa **p** je podmienka v riadku 8.

```

1 void var(int n, int k, int x, int p)
2 {
3     if (k == 0)
4     {
5         printf("%d\n", x);
6         return;
7     }
8     if (p <= n)
9     {
10        var(n, k - 1, 10 * x + p, 1); // dole (dalsia cifra)
11        var(n, k, x, p + 1); // doprava (cyklus p)
12    }
13 }
14
15 void print_var(int n, int k)
16 {
17     var(n, k, 0, 1);
18 }

```

Vnorenie v riadku 11 zodpovedá pôvodnému cyklu pre **p** – prechod na ďalšiu hodnotu **k**-tej cifry. Vnorenie v riadku 10 zodpovedá dosadeniu hodnoty **p** sprava do **x** a následné vnorenie do úrovne **k-1** pre dosadenie zostávajúcich cifier alebo výpis **x** (pre **k** = 0).

Úloha 3-11 (riešenie)

Neprehľadnosť programu spočíva v prekryvaní názvu globálnej premennej **cislo** (riadok 3) a lokálnej premennej **cislo** jednak ako argument (riadok 5) funkcie **fn()** a tiež ako lokálna premenná deklarovaná v bloku (riadok 10). Deklarovaním premennej rovnakého názvu sa stráca viditeľnosť premennej a nadobúda platnosť nová premenná, pôvodná premenná v pamäti zostáva, len nie je možné k nej pristupovať pomocou rovnakého identifikátoru (ktorý teraz ukazuje na novo zadeklarovanú premennú).

Efekt príkazu je vyznačený v komentári za príslušným riadkom:

```
1  #include <stdio.h>
2
3  int cislo = 3;
4
5  int fn(int cislo, int k)
6  {
7      cislo += k;           // cislo(parameter) = 3
8      printf("%d\n", cislo); // cislo(parameter)
9      {
10         int cislo = 2;    // cislo(vnorene) = 2
11         cislo += k;       // cislo(vnorene) = 4
12         printf("%d\n", cislo); // cislo(vnorene)
13     }
14     {
15         int k = 2;        // k(vnorene) = 2
16         cislo += k;       // cislo(parameter) = 5
17         printf("%d\n", cislo); // cislo(parameter)
18     }
19     return k;
20 }
21
22 int main(void)
23 {
24     cislo *= fn(1, 2);     // cislo(globalne) = 6
25     printf("%d\n", cislo); // cislo(globalne)
26     return 0;
27 }
```

Úloha 3-12 (riešenie)

Analýzou zdrojového kódu **online_herna.c** môžeme zistiť, kedy program herne umožní hráčovi vyhrať. Hráčovi znemožní vyhrať vtedy (riadok 11), keď suma možnej výhry (**vyhra**) je väčšia ako rozpočet, ktorý je vyčlenený na výhry (premenná

na_vyhry). Kľúčové je teda upraviť túto podmienku, aby umožnil výhru napriek tomu, že herňa nemá dostatok prostriedkov. Zdrojový kód herne meniť nemôžeme, takže môžeme len ovplyvniť hodnoty premenných, ktoré vystupujú v logickom výraze.

Premenná **na_vyhry** je globálna premenná v súbore **online_herna.c**, môžeme ju teda používať aj v ostatných súboroch. V súbore **hrac.c** predtým ako odovzdané riadenie funkcii **stavka()** v riadku 10 potrebujeme hodnotu premennej **na_vyhry** upraviť, aby výhra v herni vo funkcii **stavka()** prešla:

```

7 void skus(double suma)
8 {
9     hrac -= suma;
+   extern double na_vyhry;
+   na_vyhry += suma * 1.1 + 1.0;
10    double vyhra = stavka(suma, 1.1);
11    if (vyhra > 0.0)
12        hrac += vyhra;
13 }
```

Doplníme externú deklaráciu premennej **na_vyhry** a pripočítame k nej potrebnú hodnotu, tak aby podmienka (**online_herna.c:11**) vo funkcii **stavka()** umožnila vyplatiť výhru.

Po 100 dňoch je teraz výsledok pre hráča viac potešujúci:

```

Hrac: 2704.8 EUR
Stat(5%) = 852.4 EUR
Stat(20%) = 0.0 EUR
Zisk herne : -2557.2 EUR
```

Ako mohol programátor knižnice pre online herňu zabrániť tejto chybe?

Riešením je obmedziť rozsah platnosti premennej len v rámci modulu **online_herne** použitím statických premenných takto:

```

1 // online_herna.c
2 static double ucet, odvod, na_vyhry;
```

Úloha 3-13 (riešenie)

Porozmýšľajme najskôr akú najvyššiu hodnotu premennej **pocet** je možné dosiahnuť. Premenná **pocet** sa zvyšuje v riadkoch: **hlavny.c:20**, **dalsi.c:7** a **dalsi.c:14**, ktoré sa zopakujú najviac 5-krát, takže najvyššia hodnota, ktorú program vypíše, môže

byť 20. Ako by sa dala hodnota 20 dosiahnuť? Každé spomínané zvýšenie by muselo zvýšiť globálnu premennú **pocet**. Docieliť to môžeme tak, že do riadkov **dalsi.c:7** a **dalsi.c:14** doplníme **extern**. Ostatné chýbajúce miesta vyplníme ako **static**.

Akú môžeme dosiahnuť najnižšiu hodnotu premennej **pocet** pri výpise? Musíme obmedziť zvyšovanie globálnej premennej **pocet**, takže do riadkov **dalsi.c:7** a **dalsi.c:14** doplníme **static**. Zvyšovaniu v hlavnej funkcii v riadku 21 sa nezbavíme, môžeme však využiť vynulovanie hodnoty v riadku **hlavny.c:9** tým, že v riadku **hlavny.c:8** doplníme **extern**, čím sa identifikátor **pocet** vo funkcii **prva()** previaže s globálnou premennou **pocet** a pri každom zavolaní ju nastaví na hodnotu 0. V poslednej (piatej) iterácii nakoniec túto hodnotu (0) zvýšime ešte o 1 a premenná **pocet** dosiahne konečnú hodnotu 1, ktorú program aj vypíše.

Aké ďalšie hodnoty medzi najnižšou (1) a najvyššou (20) dokážeme vypísať. Začnime opäť od najvyšších. Hodnotu 20 sme dosiahli doplnením do chýbajúcich miest **hlavny.c:8**, **dalsi.c:6**, **dalsi.c:13** a **dalsi.c:21** postupne: **static**, **extern**, **extern**, **static**. V prípade, že zmeníme prvý **extern** (v riadku **dalsi.c:6**) na **static**, tak dosiahneme hodnotu 15, keď naopak zmeníme len druhý **extern** (v riadku **dalsi.c:13**) na **static** dostaneme 10, keď doplníme do oboch **static**, dosiahneme 5. Pozrime sa teraz nato ako sme získali hodnotu 1 a pokúsme sa získať ďalšie hodnoty 2, 3 a 4. Hodnotu 15 sme dosiahli doplnením do chýbajúcich miest **hlavny.c:8**, **dalsi.c:6**, **dalsi.c:13** a **dalsi.c:21** postupne: **extern**, **static**, **static**, **static**. Keď v riadku **dalsi.c:6** doplníme **extern**, v poslednej iterácii sa zvýši hodnota **pocet** aj v tejto funkcii a výsledok bude 2, keď namiesto toho doplníme **extern** v riadku **dalsi.c:13**, v poslednej iterácii sa hodnota **pocet** zvýši o 2 a výsledná hodnota bude 3. Ak doplníme oba na **extern**, dosiahneme výslednú hodnotu 4. Nakoniec, hodnotu 6 získame tým, že využijeme nastavenie hodnoty **pocet** na 5 v riadku **dalsi.c:21**. kam doplníme **extern** a ďalším zvýšením v hlavnej funkcii dosiahneme konečnú hodnotu 6. Program teda môže vypísať niektorú z hodnôt: 1, 2, 3, 4, 5, 6, 10, 15 alebo 20.

Všimnime si ešte jednu zvláštnosť programu **hlavny.c** v riadku 16: cyklus, v ktorom premennú **n** znižujeme po jednom až klesne na hodnotu 0, je zapísaný ako `while (n --> 0)`, čo je trochu nezvyklá vizuálna „hračka“. Nezvyklosť je v tom,

že zvyčajne sa takéto znižovanie zapisuje ako `while (n-- > 0)`, ale vďaka tomu, že medzera nie je pri interpretácii tohto kódu kompilátorom významná, môžeme to prepísať na „-->“ čím vznikne spomínaná vizuálna zvláštnosť, ktorú si môžeme predstavovať ako šípku zľava doprava, teda ako keby sa pôvodná hodnota `n` postupne znižovala (šípkou doprava) na hodnotu 0.

Riešenie dopĺňajúcej úlohy: Analyzujeme, aké hodnoty nadobúda lokálna premenná **pocet** na riadku **dalsi.c:17**. Pre všetky doterajšie doplnenia, ktoré sme rozoberali v pôvodnej úlohe, lokálna premenná **pocet** v riadku **dalsi.c:17** nadobudne inú hodnotu ako 11. Blízke hodnoty, ktoré v tom riadku dosiahne, sú 9, 10, 12 alebo 13, ale hodnota 11 tam chýba. Teraz však nie je potrebné upraviť globálnu hodnotu (premennej **pocet**) ako v pôvodnej úlohe, ale potrebujeme docieľiť úpravu lokálnej premennej. Blok **dalsi.c:12-15** zdanlivo obmedzuje zvýšenie premennej **pocet** o 2 len na premennú v tomto bloku, v dôsledku čoho nevieme hodnotu parametru funkcie **pocet** ovplyvniť. Skúste sa zamyslieť nad možnosťou, ako doplniť program v riadku **dalsi.c:13** tak, aby sa zvýšenie v riadku **dalsi.c:14** prejavilo ako zmena hodnoty lokálneho parametra **pocet**. Dá sa to dosiahnuť jedine tak, ak by parameter **pocet** nebol v tomto bloku deklarovaný, čo môžeme dosiahnuť trošku netradičným doplnením znakov `//` (komentár), čím vlastne odstránime z kódu deklaráciu premennej **pocet** v bloku a teda zvýšenie sa premietne do lokálneho parametru. Program vypíše **BINGO** ak doplníme chýbajúce miesta takto: **static**, **extern**, `//` a **static**.

Úloha 3-14 (riešenie)

- a) `int(*min3)(int, int, int)`
- b) `double (*priemer)(int, int)`
- c) `void (*vymen)(int*, int*)`
- d) `int(*kvadraticka_rovnica)(double,double,double,double*,double*)`
- e) `int(*vypocitaj)(int (*)(int,int), int, int)`

Úloha 3-15 (riešenie)

Najskôr doplníme výpočet hodnoty funkcií **f1()** v riadku 13 a **f2()** v riadku 18 podľa zadania. Použijeme matematické funkcie z knižnice **math.h**: **exp(x)** pre e^x , **fabs(x)** pre $|x|$ a **sin(x)** pre $\sin x$.

Ďalej môžeme doplniť volania funkcií **f1()**, **f2()** a **deriv()** pri výpise v riadku 28: **f1(x)**, **deriv(f1, x)**, **f2(x)** a **deriv(f2, x)**. V prípade derivácie chceme, aby fungovala všeobecne pre ľubovoľnú funkciu jednej premennej nad reálnymi číslami (**double -> double**): smerník na typ funkcie teda je **double (*f)(double)**, podľa čoho doplníme argumenty funkcie **deriv()**: funkcia **f**, a bod **x** (typ **double**) v ktorom chceme deriváciu vypočítať. Nakoniec doplníme výpočet numerickej derivácie podľa vzťahu v zadaní, zvolíme $\varepsilon = 0,001$: **f(x+0.001) - f(x)) / 0.001**.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define STEP 0.2
5
6  double deriv( double (*f)(double), double x )
7  {
8      return (f(x+0.001) - f(x)) / 0.001 ;
9  }
10
11 double f1(double x)
12 {
13     return exp(x) / (fabs(x) + 1) ;
14 }
15
16 double f2(double x)
17 {
18     return x * sin(x) ;
19 }
20
21 int main()
22 {
23     double r1, r2, x;
24     scanf("%lf %lf", &r1, &r2);
25     printf("    x    f1(x)  f1'(x)  f2(x)  f2'(x)\n");
26     for (x = r1; x < r2 + 0.000000001; x += STEP)
27         printf("%5.2lf %5.2lf %5.2lf %5.2lf %5.2lf\n",
28             x, f1(x), deriv(f1,x), f2(x), deriv(f2,x));
29     return 0;
30 }
```

Úloha 3-16 (riešenie)

Prvý argument funkcie **print()** je typu **input**, čo je zatiaľ nedefinovaný typ, a preto ho musíme definovať využitím **typedef** v riadku 3. Typ funkcie **vsetko()**, **parne()**

a **obsahuje3()**, ktoré očakáva funkcia **print()** ako argument, je **int(*input)(int*)**. Podľa toho môžeme doplniť do hlavičky funkcie **parne()** v riadku 18 argument **int *cislo**.

Najskôr doplníme implementáciu funkcie **vsetko()**. Funkcia musí pri opakovaných volaniach naplniť do argumentu postupne zvyšujúce sa číslo, preto obsah premennej **i** musí zostať zachovaný medzi volaniami (použijeme **static**) a hodnotu premennej **i** musíme pri každom volaní zvýšiť o 1 (v riadku 11 doplníme **i++**). Dokončíme implementáciu funkcie **print()**, tak aby volanie **print(vsetko)** vypísalo čísla od 0 do 100. Funkcia **print()** vypisuje čísla v cykle v riadkoch 49-50. V riadku 49 teda musí číslo niekam (ponúka sa premenná **i**) načítať volaním funkcie **fn**, ktorú dostala ako smerník v argumente: použijeme načítanie **fn(&i)**, v riadku 50 vypíšeme načítanú hodnotu **i**.

V ďalšom kroku doplníme implementáciu funkcie **parne()**. Funkcia číta čísla z funkcie **vsetko()** rovnako ako vo funkcii **print()** v riadku 49, kde sme ale použili **fn** (všeobecný) smerník na funkciu. V riadku 21 voláme konkrétne funkciu **vsetko()**, doplníme: **&i**. Potom v riadku 22 overíme, či je načítané číslo **i** párne číslo (**i%2 == 0**), ak áno, naplníme ho do argumentu **cislo** (doplníme ***cislo = i**) a vrátime 1. Po dočítaní všetkých čísel z funkcie **vsetko()** vrátime 0.

Nakoniec doplníme implementáciu funkcie **obsahuje3()**. Riadky 34, 38, 39 a 41 doplníme analogicky ako v prípade funkcie **parne()** postupne: **&i**, ***cislo = i**, **1** a **0**. Všimneme si, že funkcia používa lokálnu premennú **k**, ktorú v riadku 33 doplníme do deklarácie. Zostáva určiť príkaz cyklu (riadok 35), ktorý s využitím podmienky (riadok 36) určíme či načítané číslo **i** vyhovuje zmyslu funkcie **obsahuje3()** a vrátime ho cez výstupný argument **cislo** alebo nie. V cykle potrebujeme prejsť postupne každú cifru čísla **i**: urobíme to cyklom pre **k = i** tak, že z čísla **k** budeme postupne odstraňovať poslednú cifru (**k /= 10**) až pokiaľ sa v čísle **k** nachádzajú nejaké cifry (**k > 0**). Túto úvahu prepíšeme do cyklu.

Výsledný doplnený program je uvedený na ďalšej strane.

Riešenie doplňujúcej otázky:

Stačí upraviť riadok 34 a čítať z funkcie **parne()**: **while (parne(&i))**.

1	#include <stdio.h>	31	int obsahuje3(int *cislo)
2		32	{
3	typedef int(*input)(int*);	33	int i, k;
4		34	while (vsetko(&i))
5	int vsetko(int *cislo)	35	for (k=i; k>0; k /= 10)
6	{	36	if (k % 10 == 3)
7	static int i = 0;	37	{
8		38	*cislo = i;
9	if (i < 100)	39	return 1;
10	{	40	}
11	*cislo = i++;	41	return 0;
12	return 1;	42	}
13	}	43	
14	i = 0;	44	void print(input fn)
15		45	{
16	return 0;	46	if (!fn)
17	}	47	return;
18		48	int I;
19	int parne(int *cislo)	49	while (fn(&i))
20	{	50	printf("%d ", i);
21	int i;	51	printf("\n");
22	while (vsetko(&i))	52	}
23	if (i % 2 == 0)	53	
24	{	54	int main(void)
25	*cislo = i;	55	{
26	return 1;	56	// vsetky od 0 do 99
27	}	57	print(vsetko);
28		58	// parne od 0 do 99
29	return 0;	59	print(parne);
30	}	60	// obsahuje 3
		61	print(obsahuje3);
		62	return 0;
		63	}

Riešenie doplnujúcej úlohy: Funkciu **prvocisla()** implementujeme podľa vzoru ostatných funkcií. Funkcia bude v cykle načítavať čísla z funkcie **vsetko()** a pre každé načítané číslo **i** overí, či je prvočíslo.

Ako overíme či číslo **i** je prvočíslo? Tak, že preskúmame či nie je prvočíslo: číslo **i** nie je prvočíslo vtedy, ak má deliteľa v intervale **<2,i)**, inak je prvočíslo. V prípade, že sme deliteľa čísla **i** v danom intervale nenašli, načítané číslo **i** je prvočíslo a vrátime ho cez výstupný argument **cislo** a funkcia vráti návratovú hodnotu 1, inak pokračujeme načítaním ďalšieho čísla z funkcie **vsetko()**. Keď vyčerpáme všetky čísla z funkcie **vsetko()** vrátime návratovú hodnotu 0.

Výsledný program vyzerá takto:

```
1 int prvocisla(int *cislo)
2 {
3     int i, k;
4     while (vsetko(&i))
5     {
6         int nasli = 1;
7         for (k = 2; k*k <= i; k++)
8             if (i % k == 0)
9             {
10                 nasli = 0;
11                 break;
12             }
13         if (nasli && i > 1)
14         {
15             *cislo = i;
16             return 1;
17         }
18     }
19     return 0;
20 }
```

Na ďalšie zamyslenie: Ako by sme upravili program, aby sme filtre (funkcie **parne()**, **obsahuje3()** a **prvocisla()**, prípadne aj iné) bez zmeny ich implementácie mohli v hlavnej funkcii **main()** jednoducho ľubovoľne kombinovať? Napr. chceli by sme všetky párne prvočísla alebo všetky prvočísla obsahujúce 3.

Pomôcka: Filtre treba navrhnuť tak, že zdroj údajov nebude napevno zapísaný v kóde funkcie, ale bude ako argument funkcie. Výsledný program však nie je jednoduché správne v jazyku C naprogramovať... Skúste si to!

Kapitola 4

Polia a reťazce

Programy sú zaujímavejšie, keď dokážu spracovať veľké množstvo údajov. Doteraz sme v programoch pracovali len s jednoduchými premennými, s použitím ktorých vieme prakticky pracovať len s malým počtom premenných. Pri práci s väčším počtom premenných začína byť program rýchlo neprehľadný.

Uvažujme napr. nasledovný program, ktorý načíta 5 celých čísel a vypočíta ich súčet:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x1, x2, x3, x4, x5;
6     scanf("%d %d %d %d %d", &x1, &x2, &x3, &x4, &x5);
7     printf("%d\n", x1 + x2 + x3 + x4 + x5);
8     return 0;
9 }
```

Program môžeme zjednodušiť použitím cyklu nasledovne:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, x, sucet = 0;
6     for (i = 0; i < 5; i++)
7     {
8         scanf("%d", &x);
9         sucet += x;
10    }
11    printf("%d\n", sucet);
12    return 0;
13 }
```

Otázka na zamyslenie: Ako by sme tieto dva programy upravili ak by sme nakoniec chceli ešte načítané čísla (jednotlivo) vypísať?

Odpoveď: V prvom programe by sme len pridali ďalší výpis hodnôt premenných. V druhom programe si načítané čísla nepamätáme, a teda na konci programu ich už nie je možné vypísať. V programovacom jazyku potrebujeme nejaký mechanizmus ako si väčší počet premenných uložíme v pamäti tak, aby sme s nimi mohli aj neskôr efektívne pracovať.

4.1 Jednorozmerné polia

Na prácu s väčším počtom premenných rovnakého typu sa v programovacom jazyku C používa odvodený typ tzv. pole (array). Pole v programe definujeme ako obyčajnú premennú s pridaním hranatých zátvoriek (`[]`) za názov premennej. Do týchto zátvoriek pri definícii uvádzame (konštantnú) veľkosť poľa. Pri použití poľa prístupujeme k jednotlivým prvkom poľa pomocou zátvoriek `[]`.

Upravený program na výpočet súčtu 5 celých čísel s využitím poľa vyzerá takto:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, x[5], sucet = 0;
6      for (i = 0; i < 5; i++)
7      {
8          scanf("%d", &x[i]);
9          sucet += x[i];
10     }
11     printf("%d\n", sucet);
12     return 0;
13 }
```

V pamäti dostane pole pridelený súvislý blok pamäte. Prvky sú v poli indexované od 0, napr. pole `int p[100]` dostane 400 bytov pamäte (ak veľkosť jedného `int` je 4 byte) a prvý prvok je `p[0]`. Nasledujúci obrázok znázorňuje pole `p[100]`, ktoré je v pamäti umiestnené na adrese 264:

264	265	266	267	268	269	270	271	...	660	661	662	663
<code>p[0]</code>				<code>p[1]</code>				...	<code>p[99]</code>			

Prvky polí, ktoré sú definované ako globálne (pamäťová trieda **extern**) alebo statické (pamäťová trieda **static**), sú automaticky pred prvým použitím inicializované na hodnotu 0. Okrem automatickej inicializácie možno všetky (globálne, statické aj lokálne) polia inicializovať priradením niekoľko prvých hodnôt, ako napr. v príkaze:

```
int p[100] = {1,2,3};
```

ktorý definuje nové (lokálne) pole **p** a priradí prvkom hodnoty: **p[0]=1**, **p[1]=2**, **p[2]=3**, zvyšné prvky (**p[3]**, ..., **p[99]**) sa inicializujú na hodnoty 0. Prvky polí, ktoré sú definované v rámci blokov ako lokálne premenné, nie sú automaticky inicializované na 0. Lokálne polia môžeme inicializovať nasledujúcim jednoduchým príkazom, napr. pre pole **p** veľkosti 10: **int p[10] = {0};**

V prípade použitia takejto inicializácie poľa je možné v definícii vynechať hodnotu veľkosti poľa, ktorá je potom automaticky odvodená podľa počtu použitých prvkov pri inicializácii.

Samotná hodnota premennej **p** je adresa na začiatok poľa. Prostredníctvom nej môžeme (podobne ako použitím smerníkov) pristupovať k jednotlivým prvkom poľa takto:

264	265	266	267	268	269	270	271	...	660	661	662	663
*p, resp. *(p + 0)				*(p + 1)			...	*(p + 99)				

Premenná **p** sa nespráva rovnako ako bežný smerník: hodnotu premennej poľa **p** nie je možné upravovať (správa sa podobne ako **const** smerník). Druhý významnejší rozdiel je ten, že pomocou vstavaného operátora **sizeof** možno zistiť celkové množstvo pridelenej pamäte pre pole, čo použitím obyčajného smerníku nie je možné:

1	int p[100], *ptr = &p[0];
2	printf("%d %d\n", sizeof(p), sizeof(ptr)); // 400 4

Túto vlastnosť premenná typu pole stratí, keď ho odovzdáme do funkcie ako argument, ako je uvedené v nasledujúcom programe:

1	void fn(int pole[])
2	{
3	printf("%d\n", sizeof(pole)); // 4
4	}

```

5
6 int main(void)
7 {
8     int x[100], *p = &x[0];
9     fn(x);
10    printf("%d %d\n", sizeof(p), sizeof(ptr)); // 400 4
11    return 0;
12 }

```

V rámci funkcie **fn()** neexistuje spôsob ako zistiť veľkosť poľa **p** z jeho identifikátora, preto musíme túto veľkosť odovzdať v ďalšom argumente. Teda, keď odovzdávame nejaké pole ako argument do funkcie uvádzame vždy dva argumenty: smerník na adresu začiatku poľa (zázpisy **int pole[]** a **int *pole** sú ekvivalentné) a veľkosť poľa (zvyčajne typu **int**, resp. **unsigned int**).

Preskúmajte nasledujúce implementácie funkcií, ktoré ako argument dostanú pole **p** a jeho veľkosť **n**. Uvádzame iteratívnu (s použitím cyklu) aj rekurzívnu implementácia funkcie **sucet()**, ktorá vypočíta súčet **n** čísel v poli:

```

1 int sucet(int *p, int n)
2 {
3     int i, vysledok = 0;
4     for (i = 0; i < n; i++)
5         vysledok += p[i];
6     return vysledok;
7 }

```

```

1 int sucet(int *p, int n)
2 {
3     if (n == 0)
4         return 0;
5     return p[0] + sucet(p+1, n-1);
6 }

```

Funkciu **sucet()** môžeme v programe použiť napr. na súčet 10 prvkov poľa:

```

1 int x[1000] = { 1,2,3,4,5 };
2 printf("%d\n", sucet(x, 10));

```

Všimnime si, že pole **x** má veľkosť 1000 celých čísel, pričom sme inicializovali len prvých päť a ostatné sú automaticky inicializované na hodnotu 0. Potom sme zavolali funkciu **sucet()** ako keby pre pole dĺžky 10.

Funkcie nám týmto spôsobom umožňujú pracovať nad ľubovoľným podintervalom poľa:

$\text{sucet}(x, 10) =$	$x[0] + x[1] + \dots + x[9] = 15$
$\text{sucet}(x + 3, 5) =$	$x[3] + x[4] + \dots + x[7] = 9$

<code>sucet(&x[2], 10) ==</code>	<code>x[2] + x[4] + ... + x[11] = 12</code>
<code>sucet(&x[100], 1000) ==</code>	chybný prístup do pamäte!

Štandardná knižnica **stdlib.h** obsahuje aj funkcie na usporiadanie prvkov v poli (**qsort()**) a vyhľadanie prvku v usporiadanom poli (**bsearch()**). Použitie týchto funkcií v programe predstavuje asi najbežnejšie využitie smerníku na funkciu: pomocná funkcia na porovnanie prvkov je odovzdaná (pri volaní funkcie (**qsort()**) alebo **bsearch()**) ako smerník na funkciu predpísaného typu.

Funkcia **qsort()** umožňuje efektívne (použitím optimalizovaného algoritmu QuickSort) usporiadať prvky v poli podľa zadefinovanej funkcie pre porovnanie. Uvažujme nasledovný program, v ktorom usporiadame prvky v poli vzostupne (od najmenšieho po najväčšie) a potom aj zostupne (od najväčšieho po najmenšie) len použitím inej porovnávacej funkcie.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int cmp_asc(const void *va, const void *vb)
5  {
6      const int *ia = va, *ib = vb;
7      return (*ia) - (*ib);
8  }
9
10 int cmp_desc(const void *va, const void *vb)
11 {
12     const int *ia = va, *ib = vb;
13     return (*ib) - (*ia);
14 }
15
16 int main(void)
17 {
18     int x[] = { 4,1,5,2,3 };
19     qsort(x, 5, sizeof(int), cmp_desc); // 5, 4, 3, 2, 1
20     qsort(x, 5, sizeof(int), cmp_asc);  // 1, 2, 3, 4, 5
21     return 0;
22 }
```

Alokácia pamäte

Programovací jazyk C podporuje tri spôsoby alokácie (pridelovania) pamäte: statickú alokáciu, automatickú alokáciu a dynamickú alokáciu.

Statická alokácia je pridelenie pamäte globálnym alebo statickým (**static**) premenným na začiatku behu programu. Každá takáto premenná dostane pridelenú súvislú oblasť pamäti (v tzv. dátovej oblasti programu) počas celého behu programu.

Automatická alokácia priebežne prideluje pamäť na zásobníku volaní (call stack) lokálnym premenným a argumentom funkcií (tzv. automatickým premenným). Pamäť je pridelená pri vstupe do bloku a uvoľnená po skončení vykonávania bloku. Tento spôsob alokácie sa používa aj pri volaniach funkcií, pričom najväčšiu záťaž pre zásobník spôsobujú hlboké rekurzívne volania funkcií.

Dynamická alokácia umožňuje prostredníctvom funkcií v štandardnej knižnici jazyka C alokovať požadované množstvo pamäte za behu programu. Ku dynamicky alokovanej pamäti pristupujeme prostredníctvom smerníka.

Formálne nie je možné mať v jazyku C „dynamickú“ premennú, teda takú, ktorá priamo reprezentuje v kóde dynamickú pamäť. S dynamickou pamäťou pracujeme v zdrojovom kóde nepriamo cez smerník(y). Samotný smerník je staticky alebo automaticky alokovaný a obsahuje hodnotu (adresu pamäte), kde začína pridelená oblasť pamäte, ktorá bola dynamicky alokovaná.

Dynamická alokácia prideluje pamäť z tzv. haldy (heap), ktorá umožňuje čerpať prakticky celý rozsah operačnej pamäte počítača.

Premennú pre dynamicky alokované pole definujeme ako smerník na príslušný typ prvkov. Pre alokáciu použijeme niektorú z dostupných funkcií **malloc()**, **calloc()** alebo **realloc()** v štandardnej knižnici **stdlib.h**, na uvoľnenie používame funkciu **free()**.

Nasledujúci program alokuje dve polia veľkosti 100 a vypíše hodnoty prvých troch prvkov. V riadku 7 sme do smerníka **x** naplnili adresu pamäte, ktorú vrátila funkcia **calloc()** ako odpoveď na požiadavku na alokáciu 100 prvkov každý o veľkosti **sizeof(int)** bytov. V riadku 8 sme do **y** priradili návratovú hodnotu z **malloc()**, do ktorého sme zadali rovnakú celkovú veľkosť poľa: **100*sizeof(int)**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int i, *x, *y;
7     x = calloc(100, sizeof(int));
8     y = malloc(100 * sizeof(int));
9     for (i = 0; i < 3; i++)
10         printf("x[%d]=%d y[%d]=%d\n", i, x[i], i, y[i]);
11     return 0;
12 }
```

Výstup programu:

```
1 x[0]=0 y[0]=-842150451
2 x[1]=0 y[1]=-842150451
3 x[2]=0 y[2]=-842150451
```

Rozdiel v alokácii funkciou **calloc()** a **malloc()** spočíva v tom, že funkcia **calloc()** prvky inicializuje na hodnoty 0. Funkcia **malloc()** prvky neinicializuje, v praxi teda volanie **malloc()** prebehne o niečo rýchlejšie (za cenu neinicializovanej pamäte), pričom potom treba pridelenú pamäť v programe explicitne inicializovať.

Praktický tip: Občas je v zdrojovom kóde pri priradení alokovaných smerníkov do premenných uvedené explicitné pretypovanie typu na smerník príslušného typu: napr. pretypovanie na **int*** v nasledujúcom príkaze:

```
y = (int*)malloc(100 * sizeof(int));
```

Použitie pretypovania je v tomto prípade však nadbytočné, pretože alokačná funkcia vracia smerník všeobecného typu **void***, ktorý je pri priradení automaticky pretypovaný na požadovaný typ. Explicitné pretypovanie dokonca môže spôsobiť problémy ak zdrojový kód po prvotnom napísaní začneme upravovať a omylom uvedieme v pretypovaní nesprávny typ: kompilátor začne hlásiť upozornenia.

Funkcia **realloc()** alokuje požadovanú veľkosť bloku s tým, že pamäť inicializuje hodnotami zo starého bloku pamäte **x**. Napr. pre blok **x** veľkosť **100*sizeof(int)** volanie **x = realloc(x, 50*sizeof(int))** priradí novú pamäť pre 50 prvkov typu **int** do smerníku **x**, pričom obsah prvých 50 prvkov zo starého bloku **x** prekopíruje

do novej pamäte. Starý blok pamäte **x** je potom uvoľnený a preto zostávajúce hodnoty **x[50]**, ..., **x[99]** sa stanú nedostupné. Ak by volanie blok zväčšovalo, tak funkcia **realloc()** novú pamäť neinicializuje.

V uvedenom programe nie je použitá funkcia **free()** z viacerých dôvodov:

Prvý dôvod: výsledný kód je kratší a prehľadnejší, čo však zdanlivo môže byť aj dôsledkom neporiadnosti autora kódu. Druhý dôvod, ktorý vysvetľuje, prečo prvý dôvod nepramení z neporiadnosti: v takomto krátkom programe to nie je potrebné, pretože po ukončení procesu (vykonávania programu) operačný systém automaticky uvoľní všetku pamäť, ktorú proces alokoval.

Tretí (najväčší) dôvod je ten, že uvoľňovať pamäť na konci behu programu môže byť nevhodné až nebezpečné: v zložitých programoch sa často stáva, že vykonané alokácie pamäte sú natoľko komplikované, že zrealizovať dôsledné uvoľnenie pamäte tesne pred ukončením nielen zbytočne spomaľuje ukončenie programu ale môže byť zdrojom chýb, kvôli, ktorým program „nešťastne“ spadne až vtedy, keď vykonáva poslednú (a ako sa ukazuje zbytočnú) úlohu.

Uvoľňovanie pamäte má najväčší zmysel pri priebežnej správe pamäte, keď vieme, že niektorý alokovaný blok už nie je potrebný a môže sa nám v blízkej budúcnosti zísť, vtedy ho treba uvoľniť (volaním funkcie **free()**). Priebežné uvoľňovanie pamäte má tiež význam pri použití nástrojov na analýzu pamäti (napr. Valgrind), ktoré sú schopné nájsť úniky pamäte, tzv. memory leak.

Otázka na zamyslenie: Ako priradíme (všetky) hodnoty prvkov poľa **a** do poľa **b**?

Odpoveď: Pripomeňme, že do premennej typu pole, či už staticky alebo automaticky alokovanej, nie je možné priamo priradovať (**b = a** je chybný zápis), a v prípade dynamickej alokácie priradenie adresy do smerníka (**b = a**) nevykoná priradenie hodnôt prvkom poľa. Jediná možnosť je prvky poľa priradiť po jednom, resp. nejakým spôsobom prekopírovať obsah pamäte pridelenej poľu **a** do pamäte pridelenej pre pole **b**. Napr. funkciou na kopírovanie pamäte **memcpy()**.

Podobný prístup je použitý v nasledujúcej funkcii **memdup()**, ktorá zduplikuje obsah (pamäte) poľa do nového poľa: teda alokuje nové pole potrebnej veľkosti, do ktorého naplní obsah pôvodného poľa, a adresu na začiatok nového poľa vráti ako

návratovú hodnotu. Typ **size_t** použitý v argumente je v jazyku C zvyčajne len iné pomenovanie pre **unsigned int**, resp. **unsigned long**.

Nasleduje implementácia funkcie **memdup()**:

```
1 void *memdup(void *ptr, size_t size)
2 {
3     void *result = malloc(size);
4     if (result)
5         memcpy(result, ptr, size);
6     return result;
7 }
```

Využitie smerníkov pre prístup k pamäti prostredníctvom zátvoriek **[]** (ako k poľu) má ešte jednu veľmi zaujímavú výhodu: použitím smerníka možno upraviť hranice „poľa“. Napr. príkaz:

```
y = malloc(100 * sizeof(int)), y -= 5;
```

spôsobí, že prostredníctvom smerníka **y** možno pristupovať k prvkom alokovaného poľa ako k prvkom **y[5]**, **y[6]**, ..., **y[104]**.

Úloha 4-1

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**). Napíšte funkciu **druhe_najvacsie()**, ktorá vráti index druhého najväčšieho čísla v poli **a**. Preskúmajte nasledujúci zdrojový kód a zistite, či je chybný.

Napr. pre vstupné pole **a[] = { 3, 1, 8, 6, 10, -4, 5 }** funkcia vráti (správny) index: **2** (**a[2] = 8**). Ak je uvedený kód chybný, navrhните vlastné správne riešenie a zdôvodnite, prečo je správne.

```
1 int druhe_najvacsie(int *a, int n)
2 {
3     int i, max = 0, max2;
4     for (i = 1; i < n; i++)
5     {
6         if (a[max] <= a[i])
7         {
8             max2 = max;
9             max = i;
10        }
11    }
12    return max2;
13 }
```

Úloha 4-2

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**). Napíšte zdrojový kód funkcie **najvacsie_zaporne()**, ktorá využitím cyklu nájde v poli **a** najväčšie záporné číslo.

Napr. pre vstupné pole **a[] = { 1,10,-3,-4,-1,5 }** funkcia vráti index: **4** (**a[4] = -1**). Ak sa v poli **a** nenachádza záporné číslo, funkcia vráti -1. Okrem iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

Úloha 4-3

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**). Doplňte chýbajúce časti funkcie **najcastejsi()**, ktorá vráti hodnotu najčastejšie sa vyskytujúceho čísla v poli **a**. Ak je takých čísel viac, funkcia môže vrátiť ľubovoľné z nich. Potom upravte funkciu tak, aby ako návratovú hodnotu vracala počet výskytov najčastejšie sa vyskytujúceho čísla, pričom hodnotu čísla vráti prostredníctvom výstupného argumentu. Okrem iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

```
1  _____ najcastejsi(_____, int n)
2  {
3      int i, j, k, pocet=0, _____;
4      for (i = 0; _____; i++)
5      {
6          for (j = k = 0; j < n; j++)
7              if (_____)
8                  k++;
9          if (pocet < _____)
10         {
11             pocet = _____;
12             vysledok = _____;
13         }
14     }
15     return vysledok;
16 }
```

Úloha 4-4

Dvaja hráči sa hrajú s kameňkami. Hra začína tak, že na kôpku nahnú niekoľko kamienkov. Hráči striedavo z kôpky odoberajú kamienky, hráč na ťahu môže odobrať

z kôpky 1, 2 alebo 3 kamienky. Ten, kto zoberie posledné kamienky vyhráva. Dané je kladné celé číslo **n** (typ **int**) – počet kamienkov na kôpke.

Napíšte funkciu **vyhra()**, ktorá pre číslo **n** vráti 1 ak začínajúci hráč môže hru s **n** kamienkami na kôpke vyhrať, inak vráti 0 (vyhrá hráč, ktorý nezačína). Hodnota **n** nepresiahne 100 000. Vo svojom riešení využite pole **stav[]**, v ktorom si pre každý počet kamienkov **i** na kôpke (**i = 0, ..., n**) určíte, či hráč na ťahu (s kôpkou obsahujúcou **i** kamienkov) vyhrá alebo nie. Hodnoty **stav[i]** pre väčšie **i** môžete určiť z hodnôt **stav[j]** pre **j < i**. Okrem iteratívneho riešenia s využitím poľa navrhnete aj rekurzívne riešenie bez použitia cyklov.

Doplňujúca úloha: Modifikujte riešenie tak, aby pracovalo pre iné povolené ťahy (počty kamienkov, ktoré je dovolené z kôpky odobrať v jednom ťahu): napr. 3, 5, 7 alebo 9 kamienkov.

Úloha 4-5

Napíšte zdrojový kód funkcie **vypis_histogram()**, ktorá dostane pole celých čísel reprezentujúcich známky, ktoré študenti získali v predmete a na výstup vypíše histogram získaných známok a dosiahnuté počty do zátvorky podľa ukážky nižšie. Každá známka vo vstupnom poli **znamky[]** je vo výslednom histograme reprezentovaná jednou hviezdikou.

Vstup: `znamky[] = { 3,1,2,3,5,2,1,2,2,1,2,3,2,3,5,2,3,1,2,2,3,5 }`

Výstup:

1	1: **** (4)
2	2: ********** (9)
3	3: ***** (6)
4	4: (0)
5	5: *** (3)

Úloha 4-6

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**) usporiadaných od najmenšieho po najväčšie a celé číslo **cislo**, ktoré do poľa vkladáme. Doplňte chýbajúce časti funkcie **vloz()**, ktorá vloží prvok **cislo** do poľa **a[]**, výsledné pole musí zostať usporiadané. Pole **a[]** je dostatočné veľké aj pre ďalšie číslo.

Okrem uvedeného iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

```
1 void vloz(int cislo, int *a,         )
2 {
3     int i;
4     for (i =         ; i >= 0; i--)
5     {
6         if (        )
7             break;
8         a[        ] = a[i];
9     }
10    a[        ] = cislo;
11    (*n)++;
12 }
```

Úloha 4-7

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**). Napíšte funkciu **otoc_pole()**, ktorá obráti prvky v poli naopak: posledný bude prvý, predposledný bude druhý atď. Napr. pre vstupné pole { 1,2,3,4,5 } je obsah poľa po otočení { 5,4,3,2,1 }. Analyzujte nasledujúce riešenie, ktoré využitím funkcie **vymen()**, vymení každý prvok poľa s jeho zrkadlovým obrazom. Ak je uvedený kód chybný, navrhните vlastné správne riešenie a zdôvodните, prečo je správne. Okrem uvedeného iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

```
1 void vymen(int *i, int *j)
2 {
3     int k = *i;
4     *i = *j;
5     *j = k;
6 }
7
8 void otoc_pole(int *a, int n)
9 {
10    int i;
11    for (i = 0; i < n; i++)
12        vymen(&a[i], &a[n - i - 1]);
13 }
```


Úloha 4-8

Dané sú tri polia: pole **a[]** obsahujúce **na** celých čísel (typ **int**) usporiadaných od najmenšieho po najväčšie, pole **b[]** obsahujúce **nb** celých čísel (typ **int**) usporiadaných od najmenšieho po najväčšie a pole **c[]** dĺžky **na+nb** prvkov typu **int** bez vyplnených hodnôt. Doplňte chýbajúce časti funkcie **spoj()**, ktorá prvky vstupných polí **a[]** a **b[]** spojí do výstupného poľa **c[]**, ktoré po vykonaní funkcie bude obsahovať **na+nb** čísel z polí **a[]** a **b[]**, ktoré budú usporiadané od najmenšieho po najväčšie.

Vstup: **a[] = { -2,3,4,4,6 }, b[] = { -4,-1,2,3,5,6,7 }**

Výstup: **c[] = { -4,-2,-1,2,3,3,4,4,5,6,6,7 }**

```
1 void spoj( )
2 {
3     int ia, ib, ic;
4     for (ia = ib = ic = 0; ia < na && ib < nb; )
5     {
6         if ( )
7             c[ ] = a[ ];
8         else
9             c[ ] = b[ ];
10    }
11    while ( )
12        c[ ] = a[ ];
13    while ( )
14        c[ ] = b[ ];
15 }
```

Doplňujúce úlohy: Riešenie upravte tak, aby:

1. funkcia **spoj()** používala len 1 cyklus,
2. funkcia **spoj()** používala len 1 cyklus a 1 podmienku,
3. funkcia **spoj()** nepoužívala cyklus,
4. funkcia **spoj()** nepoužívala cyklus a používala len 1 podmienku.

4.2 Reťazce

Reťazec v programovaní je postupnosť znakov. Napr. Ahoj svet! je postupnosť znakov A, h, o, j, (medzera), s, v, e, t, ! V zdrojovom kóde je potrebné znaky a reťazce nejakým spôsobom oddeliť od ostatných príkazov. V jazyku C zapisujeme jednotlivé znaky do apostrofov (napr. znak 'x') a reťazce do úvodzoviek ("xyz"), teda reťazec "Ahoj svet!" je postupnosť znakov 'A', 'h', 'o', 'j', ' ', 's', 'v', 'e', 't' a '!'.

V pamäti je reťazec reprezentovaný ako pole prvkov typu `char`, pričom posledný znak reťazca je `'\0'`, čo je „znakový zápis“ pre hodnotu 0. Znak je v pamäti reprezentovaný číselným kódom, ktorý v zmysle použitého kódovania, zodpovedá nejakému znaku. Pri práci s reťazcami sa v programe s „vizuálnou reprezentáciou“ znakov nepracuje: vizuál znakov možno vidieť len pri výpise, resp. zobrazení-interpretácii číselného kódu ako znaku (v nejakej znakovnej sade). Jazyk C na interpretáciu znakových kódov používa štandardné ASCII kódovanie.

Reťazec `char str[50] = "Ahoj svet!"` je v pamäti reprezentovaný takto:

	0	1	2	3	4	5	6	7	8	9	10	...	49
reťazec	'A'	'h'	'o'	'j'	' '	's'	'v'	'e'	't'	'!'	'\0'	??	??
ASCII kód	65	104	111	106	32	115	118	101	116	33	0	??	??

Všimnime si ďalší rozdiel medzi poľom znakov a reťazcom: pole znakov `str` má veľkosť 50, pričom reťazec, ktorý je v ňom uložený, má dĺžku len 10 znakov. Prirodzene je každý reťazec uložený v nejakom poli, a teda v prípade reťazcov rozlišujeme (aktuálnu) dĺžku a maximálnu dĺžku. Najviac môžeme v tomto poli `str` reprezentovať reťazec obsahujúci 49 znakov (posledný 50. prvok poľa použijeme pre ukončovací znak `'\0'`).

Reťazce v zdrojovom kóde v úvodzovkách sú reťazcové konštanty. Sú to staticky alokované polia znakov obsahujúce presný počet znakov nasledovaných ukončovacím kódom. V prípade, že v programe používame rovnakú reťazcovú konštantu na rôznych miestach, všetky výskyty tejto reťazcovej konštanty budú zodpovedať rovnakej adrese v pamäti, ktorá bola tejto konštante pri začiatku behu programu statickou alokáciou pridelená. Samotné reťazcové konštanty sú polia a teda

môžeme s nimi pracovať podobne ako so smerníkmi, napr. `"Ahoj svet!"[5]` označuje písmeno `'s'`.

Premennú typu „reťazec“ môžeme inicializovať piatimi rôznymi spôsobmi:

```
1 int main(void)
2 {
3     char *a = "Ahoj svet!";
4     char b[] = "Ahoj svet!";
5     char c[] = { 'A','h','o','j',' ','s','v','e','t','!','\0' };
6     char d[100] = "Ahoj svet!";
7     char e[100] = { 'A','h','o','j',' ','s','v','e','t','!'};
8     printf("%d\n", sizeof(a)); // 4
9     printf("%d\n", sizeof(b)); // 11
10    printf("%d\n", sizeof(c)); // 11
11    printf("%d\n", sizeof(d)); // 100
12    printf("%d\n", sizeof(e)); // 100
13    return 0;
14 }
```

V riadku 3 je staticky alokovaná reťazcová konštanta príslušnej dĺžky (pole znakov dĺžky 11 naplnené hodnotami znakov), automaticky alokovaný smerník **a** na znak, pričom do tohto smerníka je priradená adresa reťazcovej konstanty. Teda v skutočnosti je potrebná pamäť aj pre smerník (**a**) aj pre reťazec (**"Ahoj svet!"**).

V riadku 4 program automaticky alokuje pole potrebnej dĺžky: 11 znakov, obsahujúce znaky reťazca ako aj ukončovací znak. Definícia v riadku 5 je len iný zápis pre definíciu v riadku 4. V riadku 6 je automaticky alokované pole dĺžky 100, pričom prvých 11 znakov sa naplní podľa znakov uvedeného reťazca, podobne v riadku 7. Všimnime si, že v definícii v riadku 5 musí byť v poli znakov uvedený aj ukončovací kód 0, lebo inak by bez uvedenia rozsahu poľa program alokoval len 10 znakov a pole znakov by nebol platný reťazec.

Reťazce vypisujeme na výstup funkciou **printf()** použitím formátu **%s**, načítavame funkciou **scanf()** použitím formátu **%s**. V nasledujúcom programe, načítame reťazec a znak zo vstupu a nájdeme jeho výskyt v reťazci pomocou funkcie **najdi()**, uvádzame aj iteratívnu (pomocou cyklu) aj rekurzívnu implementáciu:

```

1 int najdi(char *s, char c)
2 {
3     int i;
4     for (i = 0; s[i]; i++)
5         if (s[i] == c)
6             return i;
7     return -1;
8 }

```

```

1 int najdi(char *s, char c)
2 {
3     int i;
4     if (*s == 0)
5         return -1;
6     if (*s == c)
7         return 0;
8     if ((i = najdi(s+1, c)) < 0)
9         return -1;
10    return i + 1;
11 }

```

```

12 int main(void)
13 {
14     char buf[1000], znak[2];
15     int pos;
16     scanf("%s %s", buf, znak);
17     if ((pos = najdi(buf, *znak)) >= 0)
18         printf("Znak %c je na pozicii %d v %s\n", *znak, pos, buf);
19     else
20         printf("Znak %c sa v %s nenachadza\n", *znak, buf);
21     return 0;
22 }

```

Štandardná knižnica **string.h** obsahuje funkcie určené pre prácu s reťazcami.

Funkcia **strcmp()** porovná dva reťazce znak po znaku: vráti hodnotu < 0 ak je prvý menší ako druhý, > 0 ak je prvý väčší ako druhý, vráti 0 ak sú rovnaké. Porovnanie **s1="Ahoj svet!"** a **s2="Ahoj Sveto."** (**strcmp(s1, s2)**) vráti > 0 , pretože malé písmeno **'s'** má ASCII kód (115) väčší ako veľké písmeno **'S'** (83):

	0	1	2	3	4	5	6	7	8	9	10	...	49
reťazec s1	'A'	'h'	'o'	'j'	' '	's'	'v'	'e'	't'	'!'	'\0'	??	??
ASCII kód	65	104	111	106	32	115	118	101	116	33	0	??	??
	=	=	=	=	=	>							
ASCII kód	65	104	111	106	32	83	118	101	116	33	46	0	??
reťazec s2	'A'	'h'	'o'	'j'	' '	'S'	'v'	'e'	't'	'o'	'.'	'\0'	??

Funkcia **strlen()** slúži na zistenie dĺžky reťazca: **strlen("Ahoj!") == 5**.

Funkcia **strchr()** vyhľadá znak v reťazci podobne (výsledok vracia ako smerník do reťazca) ako funkcia **najdi()** v predchádzajúcom program. Funkcia

strstr() vyhľadá výskyt reťazca v inom reťazci. Funkcia **strcpy()** na kopírovanie znakov z jedného reťazca do druhého. Funkcia **strcat()** na pripojenie znakov reťazca na koniec druhého reťazca. Funkcia **strtok()** na rozdeľovanie reťazca podľa rozdeľovača (reťazec) na menšie časti.

Celkovo pri použití reťazcových funkcií je absolútne nevyhnutné zaručiť, že spracované reťazce sú ukončené 0 a majú dostatočnú maximálnu dĺžku na vykonanie operácie (napr. na nakopírovanie alebo pripojenie znakov z iného reťazca). Nedostatočná dĺžka polí (reprezentujúcich reťazce, ktoré vstupujú do týchto funkcií) je zdrojom veľkého množstva bezpečnostných chýb v softvérových systémoch.

Úloha 4-9

Určite výstup, ktorý vypíše nasledujúci program. Na pomoc si zoberte aj tabuľku ASCII kódov znakov.

```
1  int main(void)
2  {
3      printf("%c", *"abc" - ' ');
4      printf("%c%c", "sneh"[2] + 3, *("zima" + 2) + 2);
5      printf("%c\0 : )\n", *"slnko" - "+-0"[2] / 5);
6      return 0;
7  }
```

Úloha 4-10

Určite výstup, ktorý vypíše nasledujúci program.

```
1  char *hraj(char *s)
2  {
3      if (s == "kamen") return "papier";
4      if (s == "noznice") return "kamen";
5      if (s == "papier") return "noznice";
6      return "?";
7  }
8
9  int main(void)
10 {
11     char i, *x = "kamen";
12     for (i = 0; i < 47; i++)
13         x = hraj(x);
14     printf("%s\n", x);
15     return 0;
16 }
```

Úloha 4-11

Napíšte funkciu `na_male_pismena()`, ktorá prevedie reťazec na malé písmená anglickej abecedy. Analyzujte nasledujúce riešenie a zistite, či je správne. Napr. reťazec `str = "EU"` funkcia (správne) prevedie na reťazec `"eu"`.

Ak je uvedený kód chybný, navrhните vlastné správne riešenie a zdôvodnite, prečo je správne. Okrem uvedeného iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

```
1 void na_male_pismena(char *str)
2 {
3     int i;
4     for (i = 0; str[i]; i++)
5         str[i] = str[i] - 'A' + 'a';
6 }
```

Úloha 4-12

Postupka je reťazec, v ktorom nasledujú písmená abecedy za sebou, pričom za písmenom `z` nasleduje opäť písmeno `a`. Postupka je zložená alebo celá z malých písmen anglickej abecedy, alebo celá z veľkých písmen anglickej abecedy.

Napíšte funkciu `postupka()`, ktorá pre daný počiatočný znak `c` postupky a jej dĺžku `n` vráti reťazec postupky začínajúci znakom `c` dĺžky `n`.

Úloha 4-13

V hre Obesenec (angl. Hangman) hráč hľadá skrytú správu hádaním písmen. Napr. reťazec `A_ECE_A` je stav hry, v ktorej hráč uhádol písmená `A`, `C`, `E`.

Napíšte funkciu `obesenec_stav()`, ktorá dostane ako argument reťazec písmen predstavujúci skrytú správu a reťazec písmen, ktorého znaky sú písmená, ktoré hráč už skúšal, a vráti stav hry (reťazec) ako návratovú hodnotu. Napr. pre skrytú správu `"ABECEDA"` a písmená, ktoré hráč skúšal `"EAUIC"`, bude výsledok (stav hry) `"A_ECE_A"`.

Vo svojom riešení sa pokúste vhodne využiť funkcie na prácu s reťazcami zo štandardnej knižnice `string.h`.

Úloha 4-14

Daný je reťazec **str** obsahujúci anglický text. Doplňte chýbajúce časti funkcie **najcastejsi_znak()**, ktorá prostredníctvom výstupného argumentu vráti najčastejšie sa vyskytujúci znak malej anglickej abecedy v reťazci **str** a ako návratovú hodnotu vráti počet výskytov tohto znaku. Uvažujte len znaky malej anglickej abecedy. Napr. pre reťazec **str = "Dobromilka a Svetlusik sa ucia programovat!"** je najčastejšie sa opakujúce písmeno malej anglickej abecedy písmeno **'a'**, ktoré sa opakuje 6-krát.

```

1  [ ] najcastejsi_znak(char *str, [ ])
2  {
3      int i, max, pocet[ ] = { 0 };
4      for (i = 0; [ ]; i++)
5          if (str[i] >= 'a' && str[i] <= 'z')
6              pocet[ ]++;
7      for (max = [ ], i = [ ]; i < [ ]; i++)
8          if (pocet[ ] < pocet[ ])
9              max = i;
10     *znak = [ ];
11     return pocet[ ];
12 }
```

Úloha 4-15

Doplňte chýbajúce časti funkcie **odstran_male_pismena()**, ktorá vo vstupnom reťazci odstráni písmená malej anglickej abecedy a počet písmen výsledného upraveného reťazca vráti ako návratovú hodnotu. Napr. reťazec **"SlovenskaRepublika"** upraví na **"SR"** a vráti 2.

Okrem uvedeného iteratívneho riešenia navrhните aj rekurzívne riešenie bez použitia cyklov, globálnych alebo statických premenných.

```

1  [ ] odstran_male_pismena([ ])
2  {
3      int [ ];
4      for (i = j = 0; [ ]; i++)
5          if ([ ])
6              str[ ] = str[ ];
7      str[ ] = 0;
8      return j;
9  }
```

Úloha 4-16

V hre Obesenec (angl. Hangman) hráč hľadá skrytú správu hádaním písmen. Napr. reťazec `A_ECE_A` je stav hry, v ktorej hráč uhádol písmená A, C, E.

Napíšte funkciu `obesenec_riesenie()`, ktorá pre daný stav hry vypíše najkratšiu postupnosť písmen, ktorá je potrebná na uhádnutie zvyšnej časti textu. V hre hráč hľadá skrytú správu hádaním jednotlivých písmeniek. Napr. pre skrytú správu `"ABECEDA"` a stav hry `"A_ECE_A"` sú zostávajúce písmená, ktoré treba uhádnuť `"BD"`. Funkcia vráti výsledok v návratovej hodnote ako reťazec znakov.

Úloha 4-17

Daný je reťazec `str` párnej dĺžky a znak `c`. Napíšte funkciu `vloz_do_stredu()`, ktorá do stredu reťazca `str` vloží znak `c`. Napr. pre `str="ABBA"` a znak `c='+'` bude upravený reťazec `"AB+BA"`. Predpokladajte, že pole znakov `str` má dostatok miesta pre ďalší znak. Analyzujte nasledujúce riešenie a zistite, či je správne. Ak je uvedený kód chybný, navrhните vlastné správne riešenie a zdôvodnite, prečo je správne.

```
1 void vloz_do_stredu(char *str, char c)
2 {
3     int i, n = strlen(str);
4     for (i = n - 1; i >= n / 2; i--)
5         str[i + 1] = str[i];
6     str[n / 2] = c;
7 }
```

Úloha 4-18

Daný je reťazec `str`, celé číslo `n` (typ `int`) a celé číslo `offset` (typ `int`). Napíšte funkciu `odstran_znaky()`, ktorá z reťazca `str` od pozície `offset` odstráni `n` znakov. Ak nie je možné z reťazca od pozície `offset` odstrániť `n` znakov, funkcia nič nevykoná a vráti 1. Inak požadované znaky z reťazca odstráni a vráti 0.

Napr. pre `str = "totojedruhyretazec"`, volanie `odstran_znaky(str, n=5, offset=6)` vráti 0 a v reťazci `str` bude `"totojeretazec"`.

Úloha 4-19

Dané je pole znakov **dst** dĺžky **len** (typ **int**), reťazec **src** a celé číslo **offset** (typ **int**). Napíšte funkciu **vloz_reťazec()**, ktorá do reťazca **dst** vloží od pozície **offset** kópiu reťazca **src**. Argument **len** určuje počet znakov vyhradených pre pole **dst** (vrátane ukončovacieho znaku **\0**). Ak nie je možné do vyhradeného miesta pre reťazec **dst** reťazec **src** vložiť, funkcia nič nevykoná a vráti 1. Inak znaky reťazca na požadované miesto vloží a vráti 0. Napr. pre **dst = "totojeretazec"**, **offset=6**, **src="druhy"** volanie **vloz_reťazec(dst, len=50, src, offset=6)** vráti 0 a v reťazci **dst** bude **"totojedruhyretazec"**.

Úloha 4-20

Knižnica funkcií pre prácu s reťazcami **string.h** obsahuje aj funkcie, ktoré pracujú len s prvými **n** znakmi vstupného reťazca, napr. **strncpy()**, **strncat()**. Všeobecne sú tieto funkcie považované za „bezpečné“ alternatívy pôvodných funkcií bez písmena **n**: **strcpy()**, resp. **strcat()**, v ktorých nie je špecifikované obmedzenie na počet spracovaných znakov a ľahko sa preto stane, že vyhradená pamäť bude presiahnutá a vzniknuté nedefinované správanie programu môže spôsobiť narušenie systému nepovolanými útočníkmi.

Prax však ukazuje, že funkcie **strncpy()** a **strncat()** sú bezpečné len zdanlivo. Tieto funkcie majú málo známe a trochu neočakávané obmedzenia a ich nepozorné použitie môže ľahko viesť k podobným problémom ako pri pôvodných funkciách. V tejto úlohe sa pokúsime napísať spoľahlivé alternatívy pre **strncpy()** a **strncat()**, resp. **strcpy()** a **strcat()**.

- a) Funkcia **strncpy(char *dst, const char *src, size_t n)** skopíruje najviac **n** znakov z reťazca **src** do reťazca **dst**. Ak dĺžka **src** je menšia ako **n**, tak zvyšné znaky do dĺžky **n** nastaví na **\0**. Čo je horšie: ak medzi prvými **n** znakmi reťazca **src** nie je ukončovací znak **\0**, tak funkcia do **dst** nakopíruje postupnosť prvých **n** znakov **src** neukončenú znakom **\0**. Výsledná hodnota **dst** teda nebude platný reťazec! Napíšte funkciu **my_strncpy(char *dst, const char *src, int n)**, ktorá do **dst** nakopíruje najviac **n-1** znakov

zo **src**, ktoré ukončí znakom `\0`. Pole znakov **dst** by teda malo byť dlhé aspoň **n** znakov.

- b) Funkcia **strncat(char *dst, const char *src, size_t n)** pripojí najviac **n** znakov z reťazca **src** na koniec reťazca v poli **dst**. Výsledný reťazec **dst** je vždy ukončený znakom `\0`. Ak reťazec **src** má dĺžku **n** alebo viac znakov, tak funkcia pripojí len prvých **n** znakov **src** a nakoniec ešte do **dst** zapíše ukončovací znak `\0`, teda celkovo zapíše až **n+1** znakov! Najčastejšie chyby pri použití **strncat()** sú preto spôsobené nesprávnou hodnotou parametra **n**, dôležité je nepočítať ukončovací znak, resp. pole **dst** musí mať celkovú veľkosť **strlen(dst)+n+1**, argument **n** pre **strncat()** sa zvyčajne určuje výpočtom podľa dĺžky reťazca v **dst**, a pri nepozornom výpočte ľahko vznikne chyba. Napíšte funkciu **my_strncat(char *dst, const char *src, int n)**, ktorá do **dst** pripojí najviac toľko znakov z reťazca **src**, aby spolu s ukončovacím znakom `\0`, nepresiahli dĺžku **n** znakov. Pole znakov **dst** by teda malo byť dlhé aspoň **n** znakov.

Doplňujúca úloha: Funkcia **strncpy()** (aj **strncpy()**) má problém ak kopírujeme reťazec do toho istého reťazca. Napr. pre daný reťazec **buf**, volanie funkcie **strncpy(buf+3, buf)** zvyčajne nedopadne veľmi dobre. Čo sa stane? Vyskúšajte si to. Napíšte alternatívnu funkciu **my_strncpy(char *dst, const char *src)**, ktorá bude reťazec kopírovať správne aj v prípade, keď sa reťazec **src** a reťazec **dst** v pamäti prekrývajú. Možno túto funkciu implementovať bez využitia pomocného poľa?

4.3 Dlhé čísla

Polia v jazyku C umožňujú pomerne efektívne reprezentovať dlhé čísla s veľkým počtom cifier. Zabudované jednoduché typy majú obmedzený rozsah hodnôt s ktorým dokážu pracovať, pričom číslo väčšie ako tento rozsah nie je možné v premennej uložiť. Typ **unsigned int** (32 bitov) dokáže reprezentovať najviac 4 294 967 295. Typ **unsigned long** (64 bitov) najviac 18 446 744 073 709 551 615.

Najjednoduchšia reprezentácia dlhých čísel je v poli znakov (typ **char**) konštantnej dĺžky. V závislosti od typu dlhých čísel, ktoré chceme reprezentovať, môžu byť prvky poľa ľubovoľného dátového typu, ktorý dokáže reprezentovať všetky cifry čísla: v prípade desiatkových čísel dokáže typ **char** pohodlne reprezentovať hodnoty 0, 1, ..., 9. Konštantná dĺžka nám zabezpečí, že pri operáciách nemusíme uvažovať nad správnou dĺžkou, ktorá sa môže operáciami priebežne zväčšovať (prenosom do vyšších rádov).

Rozdiel medzi reprezentáciou čísla 1234 ako dlhého čísla v poli cifier dĺžky 100 a ako reťazca je znázornený na nasledujúcom obrázku:

	0	1	2	3	4	5	...	99
reťazec	'1'	'2'	'3'	'4'	'\0'	??	??	??
ASCII kód	49	50	51	52	0	??	??	??
Dlhé číslo	4	3	2	1	0	0	0	0

Rozdiel je ten, že v prípade dlhých čísel reprezentujeme v poli cifry postupne od najnižších rádov (4,3,2,1), pričom v prípade reťazcov (ľudsky čitateľnej formy čísla) sú cifry uvedené od najvyšších rádov (1,2,3,4). A tiež všetky cifry v poli pre dlhé číslo musia mať inicializovanú hodnotu, naopak v prípade reťazcu môžu byť hodnoty znakov za ukončovacím znakom **'\0'** ľubovoľné.

Nasledujúce funkcie **dlhecislo()** a **retazec()** prevádzajú tieto dve reprezentácie medzi sebou. Funkcia **dlhecislo()** pre reťazec (číslo v desiatkovej sústave) na vstupe vráti ako návratovú hodnotu dlhé číslo reprezentované v poli cifier konštantnej dĺžky (1000 cifier). Funkcia **retazec()** dlhé číslo na vstupe prevedie na reťazec a vráti ho ako návratovú hodnotu.

```

1  #define MAX_DLZKA_CISLA 1000
2
3  char *dlhecislo(const char *str)
4  {
5      int i, n = strlen(str);
6      if (n > MAX_DLZKA_CISLA)
7          return NULL; // prilis dlhy retazec
8      char *dlhe = calloc(MAX_DLZKA_CISLA, 1);
9      for (i = 0; i < n; i++)
10         dlhe[i] = str[n - i - 1] - '0';
11     return dlhe;
12 }
13
14 char *retazec(const char *dlhe)
15 {
16     char str[MAX_DLZKA_CISLA];
17     int i, j = 0;
18     for (i = MAX_DLZKA_CISLA - 1; i > 0; i--)
19         if (dlhe[i] > 0)
20             break;
21     while (i >= 0)
22         str[j++] = dlhe[i--] + '0';
23     str[j] = 0;
24     return strdup(str);
25 }

```

Výpis hodnoty dlhého čísla možno realizovať prevodom na reťazec a funkciou **printf()**, porovnanie prebieha po cifrách od najvyššieho rádu. Nasledujúci program prečíta dve dlhé čísla zo vstupu ako reťazce, prevedie ich do reprezentácie dlhých čísel v poli cifier konštantnej dĺžky a s využitím funkcie **porovnaj()** vypíše, ktoré je väčšie:

```

1  int porovnaj(const char *a, const char *b)
2  {
3      int i;
4      for (i = MAX_DLZKA_CISLA - 1; i > 0; i--)
5          if (a[i] > 0 || b[i] > 0)
6              break;
7      while (i > 0 && a[i] == b[i])
8          i--;
9      if (a[i] < b[i])
10         return -1;
11     if (a[i] > b[i])
12         return 1;
13     return 0;
14 }
15
16 int main(void)
17 {

```

```

18 char str1[2000], str2[2000];
19 scanf("%s %s", str1, str2);
20 char *a = dlhecislo(str1), *b = dlhecislo(str2);
21 if (porovnaj(a, b) > 0)
22     printf("Vacsie cislo: %s\n", retazec(a));
23 else if (porovnaj(a, b) < 0)
24     printf("Vacsie cislo: %s\n", retazec(b));
25 else
26     printf("Rovnake cisla!\n");
27 return 0;
28 }

```

Ďalšie základné funkcie na prácu s dlhými číslami v tejto reprezentácii prebiehajú v cykle po cifrách zaužívaným spôsobom.

Úloha 4-21

Dané sú nezáporné dlhé čísla **a** a **b** reprezentované v poli znakov konštantnej dĺžky ako cifry v desiatkovej sústave od najnižšieho rádu. Doplňte chýbajúce časti funkcie **pripocitaj()**, ktorá k dlhému číslu **a** pripočíta hodnotu dlhého čísla **b**. Napr. pre **a = {4,2,0,1}** (číslo 1 024) a **b = {9,9,9,9}** (číslo 9 999) bude po vykonaní funkcie **pripocitaj(a,b)** upravená hodnota **a = {3,2,0,1,1}** (číslo 11 023).

```

1 void pripocitaj(char *a, const char *b) // a += b
2 {
3     int i,            = 0;
4     for (i = 0; i < MAX_DLZKA_CISLA; i++)
5     {
6                    +=           ;
7                    =           ;
8         prenos /= 10;
9     }
10 }

```

Úloha 4-22

Daný je reťazec obsahujúci jednoduchý aritmetický výraz obsahujúci nezáporné dlhé čísla a operácie sčítania (znamienko +). Napíšte funkciu **plus_vyraz()**, ktorá reťazec s výrazom vyhodnotí a ako návratovú hodnotu vráti výslednú hodnotu ako dlhé číslo.

Napr. pre reťazec **"11+22+33+10000000"** bude výsledkom volania funkcie **plus_vyraz()** dlhé číslo **{6,6,0,0,0,0,0,1}** (číslo 10 000 066).

Úloha 4-23

Dané je nezáporné dlhé číslo **a** a celé číslo **num** (typ **int**). Doplňte chýbajúce časti funkcie **prinasob_int()**, ktorá dlhé číslo **a** vynásobí hodnotou **num**. Napr. pre **a = {4,2,0,1}** (číslo 1 024) a **num = 64** bude po vykonaní funkcie **prinasob_int(a,num)** upravená hodnota **a = {6,3,5,5,6}** (číslo 65 536).

```

1 void prinasob_int(char *a, int num) // a *= num
2 {
3     int i, _____ = 0;
4     for (i = 0; i < MAX_DLZKA_CISLA; i++)
5     {
6         _____ += _____;
7         _____ = _____;
8         prenos /= 10;
9     }
10 }
```

Úloha 4-24

Daný je reťazec obsahujúci binárne číslo s veľkým počtom cifier. Napíšte funkciu **binarne()**, ktorá reťazec prevedie na dlhé číslo v desiatkovej sústave: ako návratovú hodnotu vráti výslednú hodnotu ako dlhé číslo (pole cifier od najnižších rádov).

Napr. pre reťazec **"1010101010101"** bude výsledkom dlhé číslo **{5,4,8,1,2}** (číslo 21 845).

Úloha 4-25

Dané sú nezáporné dlhé čísla **a** a **b** reprezentované v poli znakov konštantnej dĺžky ako cifry v desiatkovej sústave od najnižšieho rádu. Doplňte chýbajúce časti funkcie **prinasob()**, ktorá k dlhému číslu **a** prinásobí hodnotu dlhého čísla **b**. Napr. pre **a = {4,2,0,1}** (číslo 1 024) a **b = {4,2,0,1}** (číslo 1 024) bude po vykonaní funkcie **prinasob(a,b)** upravená hodnota **a = {6,7,5,8,4,0,1}** (číslo 1 048 576).

```

1 void prinasob(char *a, const char *b) // a *= b
2 {
3     char x[2 * MAX_DLZKA_CISLA] = { 0 }; // vysledok
4     int i, j, prenos;
```

5	for (j = 0; j < MAX_DLZKA_CISLA; j++)
6	{
7	for ()
8	{
9	+= ;
10	= ;
11	prenos /= 10;
12	}
13	}
14	for (i = 0; i < MAX_DLZKA_CISLA; i++)
15	a[i] = x[i];
16	}

Úloha 4-26

Napíšte program, ktorý pre kladné celé číslo **n** (typ **int**) na vstupe vypíše na výstup hodnotu faktoriálu **n!** Program navrhnete tak, aby dokázal rýchlo pracovať s čo najväčšími hodnotami **n**. Zistite, aký najväčší faktoriál dokáže navrhnutý program vypočítať, ak ho necháte bežať 1 minútu.

Stihne do 1 minúty vypočítať faktoriál pre **n = 10 000**, ktorý má 35 660 cifier?

Stihne vypočítať aj faktoriál pre **n = 100 000**, ktorý má 456 574 cifier?

4.4 Smerník na smerník

Pripomeňme si postup ako vytvoríme v kóde novú funkciu: najskôr premyslíme čo by mala funkcia robiť, potom určíme hlavičku funkcie (vstupy a výstupy) a nakoniec napíšeme samotnú implementáciu funkcie. Zamerajme sa teraz na hlavičku funkcie: na určenie hlavičky funkcie si musíme rozmyslieť zoznam vstupov (vstupných argumentov) a výstupov (výstupné argumenty a návratová hodnota). Pre každý vstup aj výstup je dôležité určiť typ. V prípade výstupných argumentov je použitý typ v argumente rozšírený o znak ***** hviezdičku (smerník), teda napr. ak chceme cez výstupný argument „vrátiť“ typ **double**, tak výstupný argument bude typu **double***.

Otázka na zamyslenie: Akého typu bude výstupný argument (nie návratová hodnota) funkcie, ak chceme vrátiť smerník na reťazec? (reťazec je **char*** – smerník na znak)

Odpoveď: Použijeme zaužívaný postup: ak chceme cez výstupný argument „vrátiť“ typ **char***, tak typ rozšírime o znak ***** (smerník), a typ argumentu bude **char****: smerník na **char*** (čítame ako smerník na „smerník na **char***“).

Praktický tip: Niekedy sa takýto smerník (s dvomi hviezdčkami) označuje tiež ako **dvojitý smerník**. V programe sa môžu vyskytovať aj komplikovanejšie smerníky s viac hviezdčkami (trojitý smerník atď.). Dôležité je, aby sme si pri každom použití smerníku uvedomovali typ premennej a podľa príslušného typu premennej (smerníku) rozpoznať v akej situácii je možné takúto premennú v kóde použiť:

Smerník (akéhokoľvek typu) sa dá použiť ako výstupný argument (funkcie) pre typ bez prvej hviezdčky z typu smerníku.

Teda napr. premennú `int **a` možno použiť ako výstupný argument pre hodnotu typu `int*`. Premennú typu `double****` môžeme použiť pre výstupný argument typu `double***`, ktorý môžeme použiť pre výstupný argument typu `double**`, ktorý môžeme použiť pre výstupný argument typu `double*`, ktorý môžeme použiť pre výstupný argument typu `double`. Veľa šťastia!

Nasledujúci program používa argument typu smerník na „smerník na `int`” na výmenu hodnôt dvoch smerníkov `p` a `q`, cez ktoré pristupujeme k prvkom polí `a` a `b`, čím prakticky v hlavnej funkcii `main()` vymení „polia“ `p` a `q`:

```




1 void vymen(int **x, int **y)
2 {
3     int *tmp = *x;
4     *x = *y;
5     *y = tmp;
6 }
7
8 int main(void)
9 {
10     int a[] = { 1,2,3 };
11     int b[] = { 9,8,7 };
12     int i, *p = a, *q = b;
13     printf("p: %d %d %d\n", p[0], p[1], p[2]); // 1 2 3
14     printf("q: %d %d %d\n", q[0], q[1], q[2]); // 9 8 7
15     vymen(&p, &q);
16     printf("p: %d %d %d\n", p[0], p[1], p[2]); // 9 8 7
17     printf("q: %d %d %d\n", q[0], q[1], q[2]); // 1 2 3
18     return 0;
19 }
```

Smerník na smerník je teda obyčajný smerník: premenná, v ktorej je uložená adresa (zvyčajne iného) smerníka. Uvažujme napr. smerník na smerník na `int`: `int **pp`; Dereferenčný operátor (*) v prípade premennej `pp`, ktorá je typu smerník na smerník

môžeme použiť aj druhýkrát na prístupenie k hodnote (****pp**), ktorá je v pamäti uložená na adrese (***pp**), na ktorú ukazuje smerník, ktorého adresu (**pp**) obsahuje premenná **pp** ako svoju hodnotu. V nasledujúcom príklade opíšeme kroky pri práci s takýmto smerníkom. Uvažujme zdrojový kód:



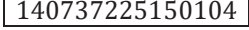
1	<code>int a, *p, **pp;</code>
2	<code>pp = &p;</code>
3	<code>*pp = &a;</code>
4	<code>**pp = 10;</code>

Po definíciách premenných v riadku 1 bude stav pamäte takýto:


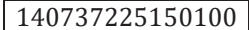
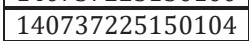
<code>int a</code>		140737225150100
<code>int *p</code>		140737225150104
<code>int **pp</code>		140737225150112

Všetky hodnoty premenných v pamäti po deklarácii sú iba nepoužiteľné hodnoty z predchádzajúcich výpočtov. Premenná **pp** (smerník na smerník) je typu **int****, teda **pp** ukazuje na miesto v pamäti, kde je uložená premenná typu **int***(smerník na **int**).

V riadku 2 do premennej **pp** program priradí adresu premennej **p**. Stav pamäte bude:

<code>int a</code>		140737225150100
<code>int *p</code>		140737225150104
<code>int **pp</code>		140737225150112

Príkaz v riadku 3 adresu premennej **a** (&a je 140737225150100) uloží na miesto, ktoré má adresu zhodnú s adresou uloženou v premennej **pp**, teda do premennej **p**. Pamäť:

<code>int a</code>		140737225150100
<code>int *p</code>		140737225150104
<code>int **pp</code>		140737225150112

Po vykonaní príkazu v riadku 3 **pp** (smerník na smerník) ukazuje na premennú **p** (smerník) a táto premenná ukazuje na premennú **a**.

Nakoniec, v riadku 4 je hodnota 10 zapísaná v pamäti na miesto, ku ktorému sa z identifikátoru premennej **pp** dostaneme prostredníctvom dvoch odkazov (operátor *****) nasledujúcim spôsobom. Keďže premenná **pp** je smerník na smerník obsahuje adresu smerníka **p**. Na tejto adrese (140737225150104) je smerník, teda uložená hodnota je opäť adresa (140737225150100), prostredníctvom ktorej sa dostaneme k pamäťovému miestu, kde bude hodnota 10 zapísaná. Keďže v riadku 4

je na adrese **p** (140737225150104) zapísaná adresa premennej **a** (140737225150100), bude hodnota 10 uložená do pamäte premennej **a**.

Konečný stav pamäte:

int	a	10	140737225150100
int	*p	140737225150100	140737225150104
int	**pp	140737225150104	140737225150112

Uvažujme teraz nasledujúci program, ktorý vo funkcii **najdi_parne()** nájde všetky párne čísla zo vstupného poľa. Funkcia musí vrátiť pole, teda je potrebné, aby okrem smerníka na začiatok výsledného poľa vrátila aj počet prvkov (vráteného) poľa. Počet prvkov funkcia vráti ako návratovú hodnotu (typ **int**), smerník na začiatok poľa funkcia vráti ako výstupný argument (typ **int****):

```

1  int najdi_parne(int *a, int n, int **vysledok)
2  {
3      int i, m, *pole;
4      for (m = i = 0; i < n; i++)
5          if (a[i] % 2 == 0)
6              m++;
7      pole = malloc(m * sizeof(int));
8      for (m = i = 0; i < n; i++)
9          if (a[i] % 2 == 0)
10             pole[m++] = a[i];
11     *vysledok = pole;
12     return m;
13 }
14
15 int main(void)
16 {
17     int i, n, *p, x[] = { 1, 2, 3, 8, 5, 6, 7, 4, 9, 10 };
18     n = najdi_parne(x, 10, &p);
19     for (i = 0; i < n; i++)
20         printf("%d ", p[i]); // 2 8 6 4 10
21     return 0;
22 }
```

Funkcia **najdi_parne()** v riadkoch 4 až 6 najskôr prejde vstupné pole a spočíta (do premennej **m**) počet párnych čísel v poli **a**. Vieme teda, že výsledok bude pole veľkosti **m** prvkov typu **int**. V riadku 7 dynamicky alokuje príslušné **pole**, a v riadkoch 8 až 10 opätovne prejde vstupné pole **a**, z ktorého do poľa **pole** postupne po jednom priradí hodnoty prvkov obsahujúcich párne čísla. Nakoniec v riadku 11

priradí do výstupného argumentu **vysledok** smerník na dynamicky alokované **pole**, ktoré obsahuje výsledné prvky a v riadku 12 vykonávanie funkcie ukončí vrátením počtu prvkov výsledného poľa ako návratovej hodnoty.

V hlavnej funkcii **main()** môžeme k výsledku pristupovať prostredníctvom smerníku **p** (použitím zátvoriek **[]**) ako k poľu.

Pole reťazcov (char **)

Smerník na smerník sa v programoch používa v rôznych ďalších situáciach. Uvažujme program, v ktorom chceme reprezentovať pole reťazcov. Nasledujúci program načíta **n** reťazcov do poľa a pole (reťazce) vypíše v opačnom poradí:

```

1  int main(void)
2  {
3      char buf[50], **str;
4      int n, i;
5      scanf("%d", &n);
6      str = malloc(n * sizeof(char*));
7      for (i = 0; i < n; i++)
8      {
9          scanf("%s", buf, 50);
10         str[i] = strdup(buf);
11     }
12     for (i = n - 1; i >= 0; i--)
13         printf("%s\n", str[i]);
14     return 0;
15 }
```

Vstup:

Výstup:

1	5	1	eva
2	adam	2	daniel
3	brona	3	cilka
4	cilka	4	brona
5	daniel	5	adam
6	eva	6	

V programe dynamicky alokujeme v riadku 6 pole **n** reťazcov. Samotná pamäť pre reťazce týmto zatiaľ nie je pridelená. Potom v cykle v riadkoch 7 až 11 načítavame reťazec do poľa znakov **buf**, pričom načítané znaky **i**-teho reťazca zo vstupu odpamätáme v poli reťazcov v **i**-tom prvku (**str[i]**). Všimnime si, že pri opakovaných načítaniach do poľa znakov (riadok 9) musíme načítaný reťazec zduplicovať do samostatného reťazca: **strdup(buf)** vytvorí v pamäti novú kópiu

reťazca, ktorý je v poli znakov **buf**. Odkaz na vytvorený reťazec, ktorý vráti **strdup()** uložíme do **i**-teho prvku poľa **str**.

Príbeh zo života: Smerník na smerník bežne stretávame aj v každodennom živote.

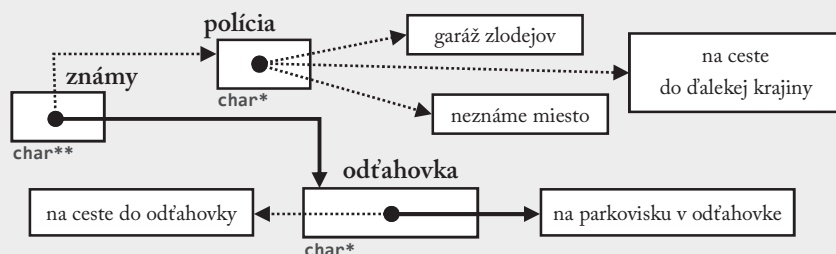
Silný zážitok so smerníkom na smerník sa stal na poslednej dovolenke s rodinou:

Po dlhej niekoľko hodinovej ceste autom sme odparkovali, zašli na obed a s rodinou strávili veľmi príjemný deň. Večer, pri návrate na ubytovanie, idem obzrieť auto, len tak, či je všetko v poriadku, keď tu zrazu: auto nikde!

Prvá myšlienka: „Ukradli ho!“, „Čo teraz?“, „Sme v ďalekej krajine, ako sa dostaneme domov?“, „Čo poviem bratovi, od ktorého sme si auto požičali? Jóóój.“ Potom prišli racionálnejšie myšlienky: „No nič, mohli sa stať aj horšie veci. Hlavne, že sme celí a zdraví...“ „Musíme to nahlásiť na políciu!“ „Ako to nahlásiť, keď neviem prečítať ani nápisy na ulici, nieto ešte rozumieť tomu čo hovoria a komunikovať s nimi“. Prišiel záblesk nádeje: „Nemuseli ho hneď ukradnúť“, „možno ho len odtiahli“, „odparkovali sme na suchej hline, tak ako ostatní ...“.

Našťastie sme poobede stretli známeho, ktorý v tom meste pracuje, hovorí po anglicky a len pred chvíľou (hodinou) sme si vymenili telefónne čísla. Zavolali sme mu, aby nám pomohol zistiť, či auto odtiahli alebo ukradli.

Vzniknutú situáciu môžeme modelovať smerníkom na smerník:



Obdĺžniky predstavujú premenné v pamäti. Známy je smerník na smerník (**char****), ktorý nás odkáže na políciu alebo odťahovú službu (oba **char***), ktorá nás odkáže na konkrétne umiestnenie „strateného“ auta. Riešenie úlohy je znázornené plnou čiarou, alternatívne hodnoty smerníkov prerušovanými čiarami.

Uloha 4-27

Zistite, v ktorých riadkoch nasledujúceho programu sa zmení hodnota celočíselnej premennej **a** a určite aké hodnoty bude nadobúdať.

```
1 int a, *p, **pp;  
2 pp = &p;  
3 p = &a;  
4 **pp = 33;  
5 *p = a + **pp;  
6 a = *p + **pp;
```

Uloha 4-28

Zistite, v ktorých riadkoch nasledujúceho programu sa zmenia hodnoty celočíselných premenných **a** a **b**, a určite aké hodnoty budú nadobúdať.

```
1 int a, b, *p, *q, **pp;  
2 p = &a;  
3 pp = &q;  
4 *pp = &b;  
5 *p = 5;  
6 *q = 2;  
7 **pp = a * b;
```

Úloha 4-29

Na vstupe sa nachádzajú celé čísla oddelené medzerami, počet čísel nie je vopred určený. Napíšte funkciu **nacitaj_pole()**, ktorá všetky čísla zo vstupu načíta do poľa, ktoré vráti ako výstupný argument a počet prvkov poľa vráti ako návratovú hodnotu.

Úloha 4-30

Dané je pole **a** obsahujúce **n** celých čísel (typ **int**) usporiadaných od najmenšieho po najväčšie, celé číslo **cislo**, ktoré do poľa vkladáme a celé číslo **kapacita** (typu **int**) zodpovedajúce dĺžke, resp. veľkosti poľa (**n <= kapacita**).

Napíšte funkciu **vloz()**, ktorá vloží prvok **cislo** do poľa **a[]**, výsledné pole musí zostať usporiadané. Predpokladajte, že pole **a[]** nemusí byť dostatočne veľké pre ďalšie číslo. V prípade úpravy poľa **a** musí funkcia prostredníctvom výstupných argumentov upraviť hodnoty **n** a **kapacita**.

Úloha 4-31

Napíšte funkciu **spoj_reťazce()**, ktorá pre vstupné pole reťazcov (typ **char****) a oddeľovací znak **z** vytvorí jeden spojený reťazec, v ktorom budú reťazce zo vstupného poľa oddelené znakom **z**.

Napr. pre pole reťazcov { "Ahoj", "mily", "svet!" } a znak **z** = ' ' (medzera) funkcia vytvorí reťazec "Ahoj mily svet!".

4.5 Viacrozmerné polia

Niekedy môžeme chcieť v premennej reprezentovať viacrozmernú štruktúru. Jazyk C umožňuje definovať tzv. viacrozmerné polia. Napr. **int p[4][3]** je dvojrozmerné pole 4 x 3 = 12 prvkov. V prípade dvojrozmerných polí zvyčajne prvý rozmer nazývame riadky a druhý rozmer stĺpce (ale je to len vec predstavivosti). Potom k prvku v **i**-tom riadku a **j**-tom stĺpci pristúpime ako **p[i][j]**. Každé pole dostane v pamäti pridelený súvislý blok pamäte, pre spomínané pole **p** je veľký **12*sizeof(int) = 48** bytov. Nasledujúci obrázok znázorňuje pole **p**, ktorému bola pridelená adresa 40, v pamäti:

40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	...	84	85	86	87
p[0][0]	p[0][1]	p[0][2]	p[1][0]	...	p[3][2]															

Prvky poľa sú v pamäti uložené po riadkoch za sebou. Nie je však úplne nevyhnutné poznať skutočné rozloženie jednotlivých prvkov poľa v pamäti: pre základnú prácu s dvojrozmernými poľami stačí jednoduchá predstava rozloženia prvkov v tabuľke:

p[0][0]	p[0][1]	p[0][2]
p[1][0]	p[1][1]	p[1][2]
p[2][0]	p[2][1]	p[2][2]
p[3][0]	p[3][1]	p[3][2]

Samotná hodnota premennej **p** je adresa na prvý prvok poľa (**p[0][0]**). Pomocou premennej **p** teda môžeme k prvkom poľa pristupovať takto:

<code>* (* (p+0)+0)</code>	<code>* (* (p+0)+1)</code>	<code>* (* (p+0)+2)</code>
<code>* (* (p+1)+0)</code>	<code>* (* (p+1)+1)</code>	<code>* (* (p+1)+2)</code>
<code>* (* (p+2)+0)</code>	<code>* (* (p+2)+1)</code>	<code>* (* (p+2)+2)</code>
<code>* (* (p+3)+0)</code>	<code>* (* (p+3)+1)</code>	<code>* (* (p+3)+2)</code>

Vzhľadom na rozloženie prvkov postupne v pamäti môžeme pre prístup k prvku `p[i][j]` využiť aj presný posun od základného smerníku: `*(&p[0][0] + 3*i+j)`, ako aj posun smerníkov po súradniciach (v kombinácií so zátvorkami): `* (* (p+i)+j)`, `* (p[i]+j)`, `* (p+i)[j]`.

Viacrozmerné polia ako argumenty funkcií

Premennú dvojrozmerného (alebo viacrozmerného) poľa môžeme tiež použiť ako argument vo funkcii. Pripomeňme, že pre jednorozmerné polia uvádzame typ argumentu ako smerník (`int *pole`) alebo ho zapisujeme ako pole (`int pole[]`), v prípade viacrozmerných polí musíme uviesť všetky rozmery okrem prvého, ktorý je nepovinný (ale môžeme ho uviesť ako pomôcku pre programátora), teda pre pole `p` rozmerov 3 x 4 to je argument `int pole[3][4]` alebo `int pole[][4]`.

Nasledujúci program definuje a inicializuje pole `p` veľkosti 3 x 4 a použitím funkcie `vypis()` ho vypíše, všimnite si inicializáciu poľa (nemusí, ale môže, obsahovať prvý rozmer), ako aj typ argumentu vo funkcii `vypis()`:

```

1  #include <stdio.h>
2
3  int p[][3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} };
4
5  void vypis(int a[][3], int n, int m)
6  {
7      int i, j;
8      for (i = 0; i < n; i++)
9          for (j = 0; j < m; j++)
10         printf("%d ", a[i][j]);
11 }
12
13 int main(void)
14 {
15     vypis(p, 4, 3); // 1 2 3 4 5 6 7 8 9 10 11 12
16     return 0;
17 }
```

Problém nastáva v prípade, keď chceme napísať funkciu, ktorá spracuje aj pole iných rozmerov, v tomto prípade je problém pri zmene druhého rozmeru. Funkciu **vypis()** môžeme implementovať pre výpis staticky alokovaného poľa ľubovoľných rozmerov s využitím posunu od základného smerníku takto:

```

1 void vypis(int *a, int n, int m)
2 {
3     int i, j;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < m; j++)
6             printf("%d ", *(a + i*m + j));
7 }
8 // ukazka volania
9 vypis((int*)p, 4, 3); // 1 2 3 4 5 6 7 8 9 10 11 12

```

V prípade, že v čase písania zdrojového kódu nevieme presnú veľkosť poľa je potrebné pole dynamicky alokovať za behu programu.

V nasledujúcom programe dynamicky alokujeme pole **q** veľkosti 4 x 3, prvky naplníme hodnotami a vypíšeme použitím upravenej funkcie **vypis()**:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void vypis(int **a, int n, int m)
5 {
6     int i, j;
7     for (i = 0; i < n; i++)
8         for (j = 0; j < m; j++)
9             printf("%d ", a[i][j]);
10 }
11
12 int main(void)
13 {
14     int i, j, **q = malloc(4 * sizeof(int*));
15     for (i = 0; i < 4; i++)
16         q[i] = malloc(3 * sizeof(int));
17     for (i = 0; i < 4; i++)
18         for (j = 0; j < 3; j++)
19             q[i][j] = 3*i + j + 1;
20
21     vypis(q, 4, 3); // 1 2 3 4 5 6 7 8 9 10 11 12
22     return 0;
23 }

```


Odkaz na pole, ktoré do funkcie **vypis()** v argumente odovzdáme, musí byť bez udania rozmerov, teda realizované ako dvojité smerník. V prípade dvojrozmerného pola s prvkami typu **int** to bude argument typu „smerník na smerník na **int**“ (**int **pole**). V programoch sa zvyčajne alebo používa staticky alokované pole rozmerov, ktoré sú postačujúce pre riešenie úloh podľa zadania úlohy, alebo sa pole dynamicky alokuje podľa veľkosti vstupu. Použije sa teda jeden z vyššie uvedených spôsobov.

Všimnime si tiež pružnosť dynamickej alokácie viacrozmerného pola: jednotlivé riadky môžu mať rôzne rozmery, čím vznikne trochu zvláštne pole niekedy tiež nazývané zubaté (ragged) alebo zúbkované (jagged):

```
1 int **r = malloc(3*sizeof(int*));
2 r[0] = malloc(2*sizeof(int));
3 r[1] = malloc(3*sizeof(int));
4 r[2] = malloc(1*sizeof(int));
```

r[0][0]	r[0][1]	
r[1][0]	r[1][1]	r[1][2]
r[2][0]		

Úloha 4-32

Doplňte chýbajúce príkazy v implementácii písmenkového hadíka. Program by mal pre nezáporné celé číslo **N** na vstupe vykresliť hadíka z písmen malej anglickej abecedy (znaky: **a, b, c, ..., z**) v dvojrozmernej matici **N x N** podľa ukážky nižšie. Písmenká sú v matici v abecednom poradí: od **a** do **z**, pričom za písmenkom **z** nasleduje opäť **a**. Zo štandardného vstupu program spracuje postupne všetky čísla. Úloha vyžaduje dynamicky alokovať pamäť pre dvojrozmerné pole.

Ukážka vstupu:

3
6

Výstup pre ukážkový vstup:

```
a b c
f e d
g h i

a b c d e f
l k j i h g
m n o p q r
x w v u t s
y z a b c d
j i h g f e
```

Vysvetlenie:

```
a b c
f e d
g h i

a b c d e f
l k j i h g
m n o p q r
x w v u t s
y z a b c d
j i h g f e
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void hadik(char **a, int n)
5  {
6      int i, j;
7              
8      for (j = 0; j < n; j++)
9      {
10         if (i % 2 == 0)
11             a[i][j] =                 ;
12         else
13             a[i][j] =                 ;
14     }
15 }
16
17 int main(void)
18 {
19     int i, j, n;
20     char **a;
21     while (                  )
22     {
23         a =                 ;
24         for (i = 0; i < n; i++)
25                             ;
26
27         hadik(a, n);
28
29         for (i = 0; i < n; i++)
30         {
31             for (j = 0; j < n; j++)
32                                 
33             printf("\n");
34         }
35     }
36 }
```

Úloha 4-33

Doplňte implementáciu programu, ktorý pre postupnosť čísel z intervalu $\langle 0, 99 \rangle$ na vstupe vytvorí histogram/grafikon po desiatkach, podobne ako pre mestskú hromadnú dopravu sa po hodinách vypisujú minúty príchodu autobusov.

Tento histogram pre každú číslicu, ktorá sa vyskytuje na pozícii desiatok, vypíše všetky číslice predstavujúce jednotky v týchto číslach, usporiadané vzostupne.

Vstup obsahuje až do konca celé čísla z intervalu <0, 99>. Nie je vopred zadáný počet čísel na vstupe, čítajte do konca vstupu. Na štandardný výstup vypíšte histogram/grafikon podľa požiadaviek.

Ukážka vstupu:

1 2 5 2 25 27 93 4 93 93 58 51

Výstup pre ukážkový vstup:

```
0 | 12245
2 | 57
5 | 18
9 | 333
```

```

1  #include <stdio.h>
2
3  int g[10][10], c[10];
4
5  int main()
6  {
7      int i, j;
8      while (scanf("%d", &i) > 0)
9      {
10         g[i/10][i%10]++;
11         c[i/10]++;
12     }
13     for (i = 0; i < 10; i++)
14         if (c[i])
15         {
16             printf("%d | ", i);
17             for (j = 0; j < 10; j++)
18                 while (g[i][j] > 0)
19                 {
20                     printf("%d", g[i][j]);
21                     g[i][j]--;
22                 }
23             printf("\n");
24         }
25     return 0;
26 }
```

Doplňujúca úloha: Upravte uvedenú implementáciu, aby namiesto dvojrozmerného poľa **g** používala jednorozmerné pole **g**.

Úloha 4-34

Štvorec je magický vtedy, keď súčet prvkov vo všetkých riadkoch, stĺpcoch a uhlopriečkach je rovnaký.

Doplňte chýbajúce časti funkcie `magicky_stvorec()`, ktorá pre dvojrozmerné pole v tvare štvorca na vstupe zistí, či je daný štvorec magický. Funkcia vráti 1 ak je štvorec magický, inak vráti 0.

Magický štvorec:

8	1	6
3	5	7
4	9	2

```

1  int magicky_stvorec(int **a, int n)
2  {
3      int i, j, sucet = 0, k;
4      for (j = 0; j < n; j++)
5          sucet += a[ ][ ];
6
7      for (i = 1; i < n; i++)
8      {
9          for (k = 0, j = 0; j < n; j++)
10             k += [ ];
11         if (k != [ ])
12             return [ ];
13     }
14
15     for (j = 0; j < n; j++)
16     {
17         for (k = 0, i = 0; i < n; i++)
18             k += [ ];
19         if ([ ])
20             return [ ];
21     }
22
23     for (k = 0, i = 0; i < n; i++)
24         k += [ ];
25     if ([ ])
26         return [ ];
27
28     for (k = 0, i = 0; i < n; i++)
29         k += [ ];
30     if ([ ])
31         return [ ];
32
33     return [ ];
34 }
```

Riešenia úloh (Kapitola 4)

Úloha 4-1 (riešenie)

Preskúmame či zdrojový kód v zadaní obsahuje chybu. Ak obsahuje chybu mala by sa prejavíť tak, že program za určitých okolností vypočíta a vypíše nesprávny výsledok. Program cyklom prechádza postupne prvky poľa: **i**-tý prvok spracuje tak, že zistí, či jeho hodnota je väčšia ako hodnota priebežného maxima (riadok 6). Ak áno, tak si v **max** zapamätáme nové maximum (riadok 9) a do **max2** si odložíme predchádzajúce maximum. Program teda druhé najväčšie číslo (**a[max2]**) určuje ako „vedľajší produkt“ pri určovaní absolútne najväčšieho čísla (**a[max]**).

Môže sa stať, že týmto postupom nenájde správne druhé najväčšie číslo? Inými slovami, môže sa stať, že pri takomto postupnom spracovaní čísel druhé najväčšie číslo nebolo predtým absolútne najväčším číslom? Samozrejme, že sa to môže stať. Napr. v prípade klesajúcej postupnosti 5, 4, 3, 2, 1 bude maximum hneď prvý prvok poľa (5), a teda premennú **max2** ani len nenastavíme na platný prvok, pričom druhý najväčší prvok existuje a je 4. Každé číslo, ktoré je menšie ako priebežné maximum teda musíme ešte porovnať s priebežným druhým najväčším prvkom: ak má väčšiu hodnotu (**a[max2] < a[i]**) tak si ho zapamätáme (**max2 = i**).

Výsledná doplnená implementácia funkcie je nasledovná:

```
1 int druhe_najvacsie(int *a, int n)
2 {
3     int i, max = 0, max2;
4     for (i = 1; i < n; i++)
5     {
6         if (a[max] <= a[i])
7         {
8             max2 = max;
9             max = i;
10        }
11        +     else if (a[max2] < a[i])
12        +         max2 = i;
13    }
14    return max2;
15 }
```

Úloha 4-2 (riešenie)

Navrhujeme najskôr hlavičku funkcie **najvacsie_zaporne()**: na vstupe je pole celých čísel **a** (typ **int[]** alebo **int***), ku ktorému musíme mať aj počet prvkov – celé číslo **n** (typu **int**). Výsledok je index najväčšieho záporného čísla vo vstupnom poli. Funkcia ako návratovú hodnotu vráti alebo index do poľa alebo -1 ak sa v poli záporné číslo nenachádza – postačí nám preto použiť typ **int**.

Začnime od výsledku: index prvku poľa. Aby funkcia našla index najväčšieho záporného čísla, musí preskúmať každé číslo v poli. Niektoré čísla nie sú záporné, a preto pri prechádzaní prvkov vstupného poľa **a** budeme uvažovať len tie prvky, ktoré sú záporné. Môže sa tiež stať, že v poli nebudú záporné čísla a výsledok bude -1. Výsledok si vo funkcii budeme reprezentovať v premennej **max**, ktorej hodnotu po skončení výpočtu funkcia vráti ako návratovú hodnotu. Premennú **max** inicializujeme na -1, a pri prechádzaní prvkami poľa (**i = 0, ..., n**) budeme rozlišovať nasledujúce situácie:

- Číslo **a[i]** nie je záporné – pokračujeme na ďalší prvok.
- Číslo **a[i]** je záporné, pričom hodnota **max** je -1 (ešte sme nenašli iné záporné číslo), tak hodnotu premennej **max** nastavíme na index **i** (**max = i**).
- Číslo **a[i]** je záporné, pričom hodnota **max** je iná ako -1 (už sme našli iné záporné číslo), tak v prípade, že hodnota prvku na indexe **max** je menšia ako hodnota prvku na indexe **i** (**a[max] < a[i]**), tak **max** nastavíme na index **i** (**max = i**).

Uvedené situácie spojíme do zloženej podmienky: **a[i] < 0 && (max < 0 || a[max] < a[i])**

Výsledná implementácia funkcie je takáto:

```
1 int najvacsie_zaporne(int *a, int n)
2 {
3     int i, max = -1;
4     for (i = 0; i < n; i++)
5     {
6         if (a[i] < 0 && (max < 0 || a[max] < a[i]))
7             max = i;
8     }
9     return max;
10 }
```

Rekurzívne riešenie: Rekurzívne riešenie spracuje v jednom rekurzívnom volaní jeden prvok poľa. Základný prípad, resp. terminálna podmienka, v ktorom rekurzívna funkcia určí výsledok bez ďalšieho rekurzívneho volania (vnorenia), nastáva keď pole **a** neobsahuje prvky ($n == 0$), a teda neobsahuje ani záporné číslo a výsledok je preto -1. Musíme teraz vymyslieť krok riešenia vo všeobecnom prípade, keď pole **a** obsahuje nejaké prvky ($n > 0$): uvažujeme prvý prvok poľa (**a[0]**), do pomocnej premennej **max** si vypočítame výsledok (najväčšie záporné číslo) pre zvyšok poľa (čísla **a[1]**, ..., **a[n]**) a využitím týchto dvoch hodnôt (**a[0]** a **max**) musíme určiť najväčšie záporné číslo (výsledok) pre pôvodné pole.

Otázka na zamyslenie: Ako získame index najväčšieho záporného prvku spomedzi prvkov **a[1]**, ..., **a[n]**?

Odpoveď: Rekurzívne volanie: **max = najvacsie_zaporne(a + 1, n - 1)** určí túto hodnotu vzhľadom na posunuté pole (začínajúce od **a[1]**), musíme ju ešte posunúť vzhľadom na pole **a** začínajúce od prvku **a[0]** nasledujúcim podmieneným príkazom: Ak je získaná hodnota **max** nezáporná ($\text{max} \geq 0$), zvýšime ju o 1. Týmto spôsobom zachováme prípadnú hodnotu -1 ak v poli **a** začínajúcom od **a[1]** nie je záporné číslo.

Podľa hodnôt **a[0]** a **max** v ďalšom výpočte rozlíšime tieto situácie:

- Ak hodnota **a[0]** je záporná ($\text{a[0]} < 0$) a hodnota **max** je -1 (medzi prvkami **a[1]**, ..., **a[n]** nie sú záporné čísla, tak výsledok volania funkcie **najvacsie_zaporne(a,n)** je 0.
- Ak hodnota **a[0]** je záporná ($\text{a[0]} < 0$) a hodnota **max** je nezáporná ($\text{max} \geq 0$) a zároveň ($\text{a[0]} > \text{a[max]}$), tak výsledok volania funkcie je 0.
- Ak hodnota **a[0]** je záporná ($\text{a[0]} < 0$) a hodnota **max** je nezáporná ($\text{max} \geq 0$) a zároveň ($\text{a[0]} < \text{a[max]}$), tak výsledok volania funkcie je **max**.
- Ak hodnota **a[0]** je nezáporná ($\text{a[0]} \geq 0$), tak výsledok volania funkcie je **max**.

Uvedené situácie implementujeme do rekurzívnej funkcie takto:

1	<code>int najvacsie_zaporne(int *a, int n)</code>
---	---

```
2 {
3   if (n == 0)
4     return -1;
5   int max = najvacsie_zaporne(a + 1, n - 1);
6   if (max >= 0)
7     max++;
8   if (a[0] < 0)
9   {
10    if (max < 0 || a[0] > a[max])
11      return 0;
12    return max;
13  }
14  return max;
15 }
```

Úloha 4-3 (riešenie)

Niektoré chýbajúce miesta vieme doplniť veľmi ľahko: premennú **vysledok**, ktorá nie je vo funkcii ešte deklarovaná, doplníme v riadku 3. Návratová hodnota funkcie zodpovedá hodnote prvku poľa – typ **int**. Vstupné pole **a** musí do funkcie prísť ako vstupný argument, preto do argumentov funkcie v riadku 1 doplníme **int *a** alebo **a[]**. Funkcia musí prejsť každý prvok poľa **a**, preto cyklus v riadku 4 má podmienku **i < n**.

Zostáva doriešiť telo cyklu (riadky 5 až 14) pre premennú **i**. Všimnime si premennú **pocet**, ktorú používame v podmienke (riadok 9), pomocou ktorej program rozhodne, či si uloží novú hodnotu premennej **vysledok**. Podmienka musí platiť vtedy, keď sa číslo, ktoré spracúvame v aktuálnej iterácii cyklu (**a[i]**), vyskytuje v poli **a** častejšie ako hodnota premennej **pocet**. Počet výskytov čísla **a[i]** je teda potrebné v tele cyklu určiť a zistenú početnosť si v nejakej premennej uchovať. Zostávajúcu premennú **k** preto použijeme nato, že vo vnútornom cykle (riadky 6 až 8) ju pre každý výskyt čísla **a[i]** zvýšime o 1.

Podmienka v riadku 7 overuje či číslo **a[j]** je rovné **a[i]** (**a[j] == a[i]**), ak áno, zvýšime počet výskytov (**k**). Nakoniec, v riadku 9, overíme, či počet výskytov (**k**) čísla **a[i]** je väčší ako doteraz najvyšší dosiahnutý počet výskytov (**pocet**), ak áno, zapamätáme si novú hodnotu **pocet** (**pocet = k**) a novú hodnotu najčastejšie sa vyskytujúceho prvku (**vysledok = a[i]**).

Výsledná doplnená funkcia vyzerá takto:


```

1  int najcastejsi(int *a, int n)
2  {
3      int i, j, k, pocet=0, vysledok;
4      for (i = 0; i < n; i++)
5      {
6          for (j = k = 0; j < n; j++)
7              if (a[j] == a[i])
8                  k++;
9          if (pocet < k)
10             {
11                 pocet = k;
12                 vysledok = a[i];
13             }
14     }
15     return vysledok;
16 }

```

Ako môžeme funkciu upraviť tak, aby výsledok odovzdala vo výstupnom argumente namiesto návratovej hodnoty?

Stačí jednoduchá úprava argumentov (riadok 1) a priradenia do premennej **vysledok** (riadok 12). Návratová hodnota bude teraz počet výskytov čísla **vysledok**. V upravenej funkcii bude argument **vysledok** smerník na celé číslo (typ **int***), úpravy sú takéto:

```

1  void najcastejsi(int *a, int n, int *vysledok)
12     *vysledok = a[i];
16     return pocet;

```

Rekurzívne riešenie: V jednom rekurzívnom volaní funkcie **najcastejsi()** spracujeme vždy prvý prvok poľa a potom sa rekurzívne zavoláme na zvyšok poľa (čiže vo vnorenom volaní bude prvým prvkom už nasledujúci prvok). Prvok poľa spracujeme tak, že využitím pomocnej rekurzívnej funkcie **pocet()** určíme jeho početnosť vo zvyšku poľa (medzi číslami **a[1]**, ..., **a[n-1]**).

Pomocná funkcia **pocet()** pre dané pole (argumenty: **int a[]**, **int n**) a číslo (typ **int**) určí početnosť čísla **cislo** medzi číslami **a[0]**, ..., **a[n-1]**. Funkcia **pocet()** bude pracovať podobne ako funkcia **najcastejsi()**: v jednom rekurzívnom volaní spracuje vždy jeden (prvý) prvok poľa. Prvok **a[0]** spracuje tak,

že ho porovná s hodnotou **cislo**, ak sa rovnajú, výsledok bude o 1 väčší ako výsledok pre zvyšok poľa, inak výsledok pre zvyšok poľa nezvýši:

```
1 int pocet(int *a, int n, int cislo)
2 {
3     if (n == 0)
4         return 0;
5     return (a[0] == cislo) + pocet(a + 1, n - 1, cislo);
6 }
```

Vo funkcii **najcastejsi()** nám rekurzívne volanie vypočíta najčastejšie sa vyskytujúci prvok vo zvyšku poľa do premennej **vysledok** ako aj jeho početnosť (označme **i**).

Podľa aktuálnych hodnôt **a[0]**, ***vysledok** a **i** potom musíme rozlíšiť tieto prípady:

- Ak prvé číslo (**a[0]**) je rovnaké ako najčastejšie sa vyskytujúce číslo vo zvyšku poľa (**a[0] == *vysledok**), tak výsledok tohto volania funkcie (najčastejšie sa vyskytujúce číslo v celom poli) je **a[0]** a jeho početnosť (návratová hodnota) je **i+1**.
- Ak číslo **a[0]** nie je rovnaké ako najčastejšie sa vyskytujúce číslo vo zvyšku poľa (**a[0] != *vysledok**) tak určíme jeho početnosť **k** (pomocou volania **k = pocet(a, n, a[0])**) a ak **k > i**, tak máme nové najpočetnejšie číslo **a[0]** s početnosťou (návratová hodnota) **k**. Inak, pre celé pole zostáva číslo ***vysledok** najpočetnejšie číslo, s početnosťou **i**:

Rekurzívne volania sa vnášajú do skráteného poľa. Základný prípad, v ktorom nie je už potrebné rekurzívne volanie, je prípad pre pole dĺžky 1, v ktorom je najpočetnejšie číslo prvé číslo (**a[0]**) s početnosťou (návratovou hodnotou) 1.

Výsledný zdrojový kód rekurzívnej funkcie **najpocetnejjsi()** podľa uvedeného opisu je takýto:

```
8  int najcastejsi(int *a, int n, int *vysledok)
9  {
10     int i, j, k;
11     if (n == 1)
12     {
13         *vysledok = a[0];
14         return 1;
15     }
16     i = najcastejsi(a + 1, n - 1, vysledok);
17     if (a[0] == *vysledok)
18         return i + 1;
19     k = pocet(a, n, a[0]);
20     if (k > i)
21     {
22         *vysledok = a[0];
23         return k;
24     }
25     return i;
26 }
```

V predchádzajúcich prístupoch sme pre každý prvok poľa, ktorých je n , zisťovali jeho početnosť tým, že sme pole opäť prešli. Koľko sme pritom vykonali porovnaní prvkov poľa medzi sebou? Pole má n prvkov, a keď by sme jeho hodnotu porovnali so všetkými n prvkami, tak vykonáme n^2 porovnaní. Pre $n = 100$ to bude rádovo 10 000 porovnaní, keď nepočítame porovnania riadiacich premenných v cykle a iné.

Pokročilejší efektívnejší postup: Existuje ešte algoritmickejšie efektívnejšie postupy, pri ktorom vykonáme len rádovo $n \log n$ porovnaní prvkov. Zníženie počtu potrebných porovnaní dosiahneme ak prvky poľa a usporiadame podľa veľkosti.

Na určenie početnosti príslušných hodnôt v poli a stačí v usporiadanom poli spočítať dĺžky za sebou idúcich postupností rovnakých čísel, napr. pre pôvodné pole $a[] = \{ 3, 1, 8, -7, 1, 3, -1, 2, -1, -1 \}$, je v usporiadanom poli $b[] = \{ 7, -1, -1, -1, 1, 1, 2, 3, 3, 8 \}$ najpočetnejší úsek rovnakých čísel -1.

Efektívne algoritmy pre usporiadanie čísel potrebujú vykonať rádovo $n \log n$ operácií a nasledujúce vypočítanie početností za sebou idúcich rovnakých úsekov čísel vyžaduje vykonať rádovo n operácií.

Výsledný program, ktorý používa implementáciu usporiadania pomocou optimalizovanej implementácie algoritmu QuickSort použitím funkcie `qsort()` je takýto:

```
1 int najcastejsi(int *a, int n, int *vysledok)
2 {
3     int *b = memdup(a, sizeof(int) * n);
4     qsort(b, n, sizeof(int), int_cmp);
5
6     int i, j = 1, k = b[0], pocet = 1;
7     *vysledok = b[0];
8     for (i = 1; i < n; i++)
9     {
10         if (b[i] == k)
11         {
12             j++;
13             if (pocet < j)
14             {
15                 pocet = j;
16                 *vysledok = k;
17             }
18         }
19         else
20         {
21             k = b[i];
22             j = 1;
23         }
24     }
25     return pocet;
26 }
```

Funkcia využíva pomocné funkcie, ktoré sme už implementovali: funkciu `memdup()` na vytvorenie pomocného poľa `b[]` skopírovaním z pôvodného poľa `a[]` a funkciu `int_cmp()` na porovnanie dvoch celých čísel.

Vo funkcii počítame dĺžku úsekov (`j`) za sebou idúcich rovnakých čísel. Ak je dosiahnutá početnosť (`j`) väčšia ako najdlhšia doteraz dosiahnutá početnosť (`pocet`), tak si zapamätáme nový výsledok (riadky 13-16).

Úloha 4-4 (riešenie)

Výhernosť hry pre hráča na ťahu je jednoznačne určená počtom kamienkov na kôpke, tzv. stav hry. Na určenie, či stav je pre hráča na ťahu výherný alebo nie a vyhrá hráč, ktorý bude ťahať ako druhý, je potrebné preskúmať všetky možné ťahy. Ak zo stavu s počtom kamienkov `i` existuje ťah do stavu s počtom kamienkov `j`, ktorý je prehrávajúci (pre hráča na ťahu), tak stav `i` je vyhrávajúci (pre hráča na ťahu). Týmto spôsobom budeme pri určovaní výhernosti stavov hry postupovať od stavov s nižším

počtom kamienkov k vyšším počtom postupne pre $i = 1, \dots, n$. Pre stav i preskúmame stavy $i-1$, $i-2$ a $i-3$, ak medzi nimi existuje prehrávajúci tak nutne je stav i vyhrajúci.

Implementácia tohto postupu je takáto:

```

1  int vyhra(int n)
2  {
3      int stav[100001], i;
4      stav[0] = 0; // prehra
5
6      for (i = 1; i <= n; i++)
7      {
8          // ak existuje tah do prehravajuceho stavu
9          if (i - 1 >= 0 && !stav[i - 1])
10             stav[i] = 1; // vyhra
11          if (i - 2 >= 0 && !stav[i - 2])
12             stav[i] = 1; // vyhra
13          if (i - 3 >= 0 && !stav[i - 3])
14             stav[i] = 1; // vyhra
15      }
16      return stav[n];
17  }
```

Funkcia **vyhra()** použije pomocnú pamäť pole **stav[]**, v ktorom pre vstupné číslo **n** určí výhernosť stavov pre $i = 1, \dots, n$. Nevýhodou je, že pri viacnásobných volaniach funkcie pre rôzne hodnoty **n** sa pole **stav[]** počíta vždy od začiatku. Upravte implementáciu funkcie tak, aby hodnotu **stav[i]** pre nejaké konkrétne **i** počítali počas vykonania celého programu len raz. Uvažujme napr. volania **vyhra(100)** a **vyhra(50)**: v prípade druhého volania (pre $n = 50$) by funkcia už nemusela hodnotu **stav[50]** opätovne počítať, len by vrátila hodnotu, ktorú už vypočítala pri predchádzajúcom volaní funkcie (pre $n = 100$).

Implementáciu môžeme upraviť použitím statického poľa **stav[]** takto:

```

1  int vyhra(int n)
2  {
3      static int stav[100001], i;
4      // vyhra: stav[i] == 1, prehra stav[i] == 2
5      for (i = n; i > 0; i--)
6          if (stav[i] > 0)
7              break;
8      stav[0] = 2;
```

```

9    // dopocitame hodnoty stav[i], ..., stav[n]
10   for (i++; i <= n; i++)
11   {
12       stav[i] = 2; // prehra
13       if (i - 1 >= 0 && stav[i - 1] == 2)
14           stav[i] = 1; // vyhra
15       if (i - 2 >= 0 && stav[i - 2] == 2)
16           stav[i] = 1; // vyhra
17       if (i - 3 >= 0 && stav[i - 3] == 2)
18           stav[i] = 1; // vyhra
19   }
20   return stav[n] % 2;
21 }

```

Prvky statického poľa **stav[]** sú inicializované na hodnotu 0, preto prehrávajúci stav hry musíme reprezentovať iným číslom – zvolíme číslo 2. Volanie funkcie najskôr (riadky 5 až 7) určí stav hry s čo najvyšším počtom kamienkov, ktorého výhernosť už bola vypočítaná a pokračuje vo výpočte výhernosti stavov s vyšším počtom kamienkov (riadky 10 až 19).

Analýzou výhernosti stavov môžeme pozorovať, že prehrávajúce sú stavy, v ktorých je počet kamienkov na kôpke deliteľný 4. Ostatné stavy sú vyherné. Preto výhernosť stavu môžeme v prípade ťahov 1, 2 alebo 3 realizovať aj triviálnou funkciou:

```

1  int vyhra(int n)
2  {
3      return n % 4;
4  }

```

Riešenie doplnujúcej úlohy: Iná situácia nastane v prípade, keď sú povolené ťahy iné, napr. 3, 5, 7 a 9, prip. konkrétne ťahy môžu byť uvedené na vstupe. V tomto prípade je potrebné upraviť podmienky, v ktorých funkcia overuje či sa možno zo stavu s počtom kamienkov **i** dostať jedným ťahom do prehrávajúceho stavu hry.

Rekurzívne riešenie: Základné rekurzívne riešenie určí výhernosť ťahu na základe výhernosti stavov hry do ktorých sa vieme dostať jedným ťahom. Základný prípad, ktorý nevyžaduje rekurzívne volanie je volanie pre **n = 0**, vtedy je hra prehrávajúca. Inak (**n > 0**) skúsime vykonať povolené ťahy a ak sa niektorým dostaneme do prehrávajúceho stavu, tak stav hry pre **n** kamienkov je výherný. Implementácia rekurzívnej funkcie **vyhra()** je týmto spôsobom priamočiara:

```
1 int vyhra(int n)
2 {
3     if (n == 0)
4         return 0;
5     if (n - 1 >= 0 && !vyhra(n - 1))
6         return 1;
7     if (n - 2 >= 0 && !vyhra(n - 2))
8         return 1;
9     if (n - 3 >= 0 && !vyhra(n - 3))
10        return 1;
11    return 0;
12 }
```

Veľkou nevýhodou tohto postupu je veľký počet volaní funkcie **vyhra()** pre opakujúce sa hodnoty **n**. Napr. pre **n=10** volanie **vyhra(10)** spôsobí ďalších 227 volaní funkcie **vyhra()**, volanie **vyhra(20)** spôsobí ďalších 39 214 volaní, volanie **vyhra(30)** až ďalších 3 843 199 volaní. Pre vyššie **n** začína byť počet volaní už neúnosný aj pre moderné počítače a výpočet trvá veľmi dlho.

Riešením tohto problému, pri ktorom sa funkcia **vyhra()** počas vykonávania výpočtu viackrát zavolá pre rovnaké hodnoty vstupného argumentu **n**, je pamätanie vypočítaných výsledkov podobne ako v prípade iteratívneho riešenia. V prípade, že výpočet pre konkrétne **n** už bol realizovaný a výsledok máme zapamätaný, tak ho bez opakovania výpočtu vrátime ako návratovú hodnotu. Aj v rekurzívnom riešení môžeme využiť statické pole:

```
1 int vyhra(int n)
2 {
3     static int stav[100001];
4
5     if (n == 0)
6         return 0;
7     if (stav[n] > 0) // vysledok uz je vypocitane
8         return stav[n] - 1;
9     if (n - 1 >= 0 && !vyhra(n - 1))
10        return (stav[n] = 2) - 1; // vyhra
11    if (n - 2 >= 0 && !vyhra(n - 2))
12        return (stav[n] = 2) - 1; // vyhra
13    if (n - 3 >= 0 && !vyhra(n - 3))
14        return (stav[n] = 2) - 1; // vyhra
15    return (stav[n] = 1) - 1; // prehra
16 }
```

Úloha 4-5 (riešenie)

Najskôr určíme vstupné a výstupné argumenty funkcie. Podľa zadania je jediný vstup pole celých čísel, preto vstupné argumenty sú dva: adresa prvého prvku poľa (typ `int*`) a počet prvkov poľa (typ `int`). Cieľom funkcie je výpis na výstup, teda výstupné argumenty ani návratovú hodnotu funkcia nemá.

Všimnime si v ukážke vstupu, že známky vo vstupnom poli sú v ľubovoľnom poradí. Jednotlivé počty známok preto musíme určiť predtým ako začneme výpis na výstup. Tieto počty (výsledok) budeme reprezentovať poľom celých čísel **pocet[6]**. Pre každú hodnotu známky (1, 2, 3, 4 a 5) určíme počet známok vo vstupnom poli **znamky[]** s touto hodnotou. Na začiatok prvky poľa **pocet[]** inicializujeme na 0, cyklom prejdeme každú známku v poli **znamky[]** a pre každú známku zvýšime príslušnú hodnotu v poli **pocet[]** o 1. Nakoniec vypočítané hodnoty vypíšeme. Výsledný program vyzerá takto:

```
1 void vypis_histogram(int *znamky, int n)
2 {
3     int i, j, pocet[6] = { 0 };
4
5     for (i = 0; i < n; i++)
6         pocet[znamky[i]]++;
7     for (i = 1; i <= 5; i++)
8     {
9         printf("%d: ", i);
10        for (j = 0; j < pocet[i]; j++)
11            printf("*");
12        if (pocet[i])
13            printf(" ");
14        printf("(%d)\n", pocet[i]);
15    }
16 }
```

Úloha 4-6 (riešenie)

V riadku 11 funkcia zvyšuje premennú **n** (pravdepodobne vyjadrujúcu počet prvkov poľa) využitím smerníku v argumente. Do hlavičky funkcie preto doplníme **n** ako smerník na celé číslo (typ `int*`).

Funkcia musí správne určiť miesto v poli, do ktorého umiestni nové číslo, a preto v riadku 4 musíme prejsť všetky čísla. Podľa podmienky a kroku cyklu je

zrejme, že pole prechádzame od konca, preto v riadku 4 doplníme index posledného prvku poľa $((*n)-1)$. Hľadanie miesta (riadok 6) sa zastaví vtedy, ak spracúvané číslo $a[i]$ je menšie ako vkladané číslo **cislo**. Na uloženie nového čísla musíme v poli vytvoriť voľné miesto. Miesto, kam číslo uložíme, mohlo potenciálne obsahovať existujúci prvok, ktorý musíme presunúť na nasledujúce miesto, preto v riadku 8 doplníme **i+1** (presunieme číslo z $a[i]$ do $a[i+1]$). Po skončení cyklu vložíme nové číslo na vytvorené voľné miesto, doplníme **i+1**.

Výsledná doplnená funkcia vyzerá takto:

```

1 void vloz(int cislo, int *a, int *n)
2 {
3     int i;
4     for (i = (*n)-1; i >= 0; i--)
5     {
6         if (a[i] < cislo)
7             break;
8         a[i+1] = a[i];
9     }
10    a[i+1] = cislo;
11    (*n)++;
12 }
```

Rekurzívne riešenie: V rekurzívnom riešení musíme zabezpečiť vytvorenie voľného miesta, do ktorého nové číslo uložíme. Postupovať budeme od konca poľa. V jednom rekurzívnom volaní funkcia posunie aktuálne číslo na nasledujúce miesto v poli a rekurzívne sa zavolá pre pole kratšie o jeden prvok. Základný prípad, v ktorom rekurzia nepotrebuje ďalšie vnorenie, je prípad, že sme sa dostali na začiatok poľa ($n==0$) alebo predchádzajúce číslo je menšie ako vkladané číslo ($a[n-1] < \text{cislo}$). V základnom prípade uložíme nové číslo na príslušné miesto v poli (**n**) a skončíme (rekurzívne volania sa postupne ukončia).

Rekurzívna funkcia **vloz()** musí ešte upraviť dĺžku poľa **a[]** využitím smerníka v argumente. Hodnotu dĺžky poľa (**n**) bude pri vnáraní znižovať o 1, pri vložení čísla a pri vynáraní ju bude zväčšovať o 1, čo v konečnom dôsledku po vložení prvku spôsobí zvýšenie dĺžky poľa o 1. Implementácia rekurzívnej funkcie **vloz()** podľa uvedeného opisu je takáto:

```

1 void vloz(int cislo, int *a, int *n)
2 {
```

```

3   if (*n == 0 || a[*n] - 1 < cislo)
4   {
5       a[*n] = cislo;
6       (*n)++;
7       return;
8   }
9   a[*n] = a[*n] - 1;
10  (*n)--; // rekurzívne pokračujeme predchadzajucim prvkom
11  vloz(cislo, a, n);
12  (*n)++; // pri navrate vratime *n do povodneho stavu
13  }

```

Úloha 4-7 (riešenie)

Uvedené riešenie síce vymení každý prvok poľa **a[]** s jeho obráteným prvkom, ale urobí to postupne pre prvky **i=0,...,n-1**: pre každý prvok dvakrát a preto po uskutočnení všetkých výmen hodnôt v poli **a[]** vlastne zostanú rovnaké ako na začiatku. Pre konkrétnu ilustráciu sledujme výmeny prvku **a[0]**: najskôr pre **i=0** vymeníme **a[0]** s **a[n-1]**, a potom pre **i=n-1** vymeníme **a[n-1]** s **a[0]**, teda oba prvky dostanú svoje pôvodné hodnoty.

Výmeny preto musíme realizovať len do polovice poľa, čím sa každý prvok vymení práve raz (resp. v prípade poľa nepárnej dĺžky sa hodnota prvku v strede poľa nevymieňa).

Upravená správna funkcia vyzerá takto:

```

8   void otoc_pole(int *a, int n)
9   {
10      int i;
11      for (i = 0; i < n/2; i++)
12          vymen(&a[i], &a[n - i - 1]);
13  }

```

Rekurzívne riešenie: Základný prípad (v ktorom už ďalšie rekurzívne volania nie sú potrebné) je pole, v ktorom netreba nič meniť: teda pole dĺžky **n=0** alebo **n=1**. Pre dlhšie pole (**n > 1**) vymeníme hodnoty prvého a posledného prvku (**vymen(&a[0], &a[n-1])**) a rekurzívne pokračujeme s poľom skráteným o jeden prvok na začiatku aj na konci (**otoc_pole(a+1, n-2)**).

Výsledná rekurzívna funkcia je takáto:

```
8 void otoc_pole(int *a, int n)
9 {
10     if (n > 1)
11     {
12         vymen(&a[0], &a[n - 1]);
13         otoc_pole(a + 1, n - 2);
14     }
15 }
```

Úloha 4-8 (riešenie)

Najskôr doplníme hlavičku funkcie o argumenty: vstupné polia **a[]** (smerník na začiatok a dĺžka **na**), **b[]** (smerník na začiatok a dĺžka **nb**) a výstupné pole **c[]** (smerník na začiatok, dĺžku netreba uvádzať v argumente lebo ju možno vypočítať ako **na+nb**).

V programe chýbajú miesta dvoch typov: priradenia do poľa **c[]** (riadky 7, 9, 12 a 14) a podmienky, na základe ktorých priradenia vykonáme (riadky 6, 11 a 13). Premenné **ia**, **ib**, **ic** označujú indexy do polí **a[]**, **b[]** a **c[]**. V prípade priradení zo vstupného poľa (rovnako to platí pre **a[]** aj **b[]**) sa musíme týmito indexami v poliach posunúť, preto priradenie hodnoty z poľa **a[]** do poľa **c[]** bude **c[ic++] = a[ia++]**, a z poľa **b[]** to poľa **c[]** bude **c[ic++] = b[ib++]**. Podmienka (riadok 6) musí rozlíšiť, či priradovať z poľa **a[]** alebo z poľa **b[]**, a preto zrejme bude vzájomne porovnávať aktuálny prvok z poľa **a[]** (**a[ia]**) s aktuálnym prvkom z poľa **b[]** (**b[ib]**) a podľa výsledku porovnania vykoná priradenie z poľa, ktorého aktuálny prvok je menší.

V prípade riadkov 11 a 13 ide o dočerpanie zvyšných prvkov v poli, pričom v ostatnom poli už prvky nie sú a preto nie je potrebné porovnanie hodnôt prvkov ale postačuje porovnanie indexu s počtom prvkov v poli: **ia < na** v riadku 11 a **ib < nb** v riadku 13.

Výsledný doplnený program je takýto:

```

1 void spoj( int *a, int na, int *b, int nb, int *c )
2 {
3     int ia, ib, ic;
4     for (ia = ib = ic = 0; ia < na && ib < nb; )
5     {
6         if ( a[ia] < b[ib] )
7             c[ic++] = a[ia++];
8         else
9             c[ic++] = b[ib++];
10    }
11    while ( ia < na )
12        c[ic++] = a[ia++];
13    while ( ib < nb )
14        c[ic++] = b[ib++];
15 }

```

Riešenie dopĺňujúcej úlohy č. 1 (funkcia `spoj()` používajúca len 1 cyklus): Našou úlohou je nejako včleniť cykly `while` v riadkoch 11-12 a 13-14 do hlavného cyklu `for`. Hlavný `for` cyklus sa opakuje len v prípade keď oba indexy `ia` aj `ib` označujú platný prvok poľa (`ia < na && ib < nb`), pričom v zostávajúcich `while` cykloch už platí slabšia podmienka: že platný prvok označuje len jeden z `ia` a `ib`. Upravený spojený cyklus teda musí zahŕňať oba stavy (reprezentované podmienkou: `ia < na || ib < nb`) a jednotlivé prípady musíme rozlíšiť v rámci tela cyklu, napr. takto:

```

1 void spoj(int *a, int na, int *b, int nb, int *c)
2 {
3     int ia, ib, ic;
4     for (ia = ib = ic = 0; ia < na || ib < nb; )
5     {
6         if (ia < na && ib < nb)
7         {
8             if (a[ia] < b[ib])
9                 c[ic++] = a[ia++];
10            else
11                c[ic++] = b[ib++];
12        }
13        else
14        {
15            if (ia < na)
16                c[ic++] = a[ia++];
17            if (ib < nb)
18                c[ic++] = b[ib++];
19        }
20    }
21 }

```

Riešenie doplnujúcej úlohy č. 2 (funkcia `spoj()` používajúca len 1 cyklus a 1 podmienku): Všimnime si, že priradenia v riadkoch 7 a 12 sú rovnaké, ako aj priradenia v riadkoch 9 a 14. Môžeme teda vytvoriť jednu presnejšiu podmienku, ktorá rozlíši oba prípady v jednom zloženom logickom výraze takto:

```
1 void spoj(int *a, int na, int *b, int nb, int *c)
2 {
3     int ia, ib, ic;
4     for (ia = ib = ic = 0; ia < na || ib < nb; )
5     {
6         if (ia < na && ib < nb && a[ia] < b[ib] || ib >= nb)
7             c[ic++] = a[ia++];
8         else
9             c[ic++] = b[ib++];
10    }
11 }
```

Riešenie doplnujúcej úlohy č. 3 (funkcia `spoj()` bez použitia cyklov): Vzhľadom nato, že potrebujeme spracovať **na** prvkov v poli **a[]** a **nb** prvkov v poli **b[]** nie je možné ich spracovanie programom s konštantným počtom príkazov bez možnosti opakovaného vykonania príkazov: v tomto prípade nám zostáva ako forma opakovaného vykonania príkazov len volanie funkcií – rekurzia.

Rekurzívne riešenie navrhujeme v zmysle riešenie z doplnujúcej úlohy č. 1. Pri priradení prvku do poľa **c[]** sa rozhodujeme medzi prvými nepriradenými prvkami zo vstupných polí. V rekurzívnom volaní budú prvé nepriradené prvky zodpovedať prvým prvkom polí vo vstupných argumentoch. Porovnáme teda hodnoty prvých prvkov: priradíme prvky z príslušného poľa a rekurzívne sa zavoláme na zmenšenom poli (bez prvého prvku), čím pokračujeme v priradovaní pre najbližší ďalší nepriradený prvok.

Vo všeobecnom prípade, keď v oboch vstupných poliach ešte zostali prvky (**na > 0 && nb > 0**), porovnáme hodnoty týchto dvoch prvkov (***a < *b**). V obmedzenom prípade (**na > 0 && nb == 0**) priradíme prvok zo zostávajúceho poľa **a[]**, resp. ak (**na == 0 && nb > 0**) z poľa **b[]**. Základný prípad, keď výpočet už nevyžaduje ďalšie rekurzívne vnorenie, je ten, že sme obe polia vyčerpali. Rekurzívne riešenie podľa tohto opisu je takéto:

```
1 void spoj(int *a, int na, int *b, int nb, int *c)
2 {
3     if (na > 0 && nb > 0)
4     {
5         if (*a < *b)
6         {
7             *c = *a;
8             spoj(a + 1, na - 1, b, nb, c + 1);
9         }
10        else
11        {
12            *c = *b;
13            spoj(a, na, b + 1, nb - 1, c + 1);
14        }
15    }
16    else
17    {
18        if (na > 0)
19        {
20            *c = *a;
21            spoj(a + 1, na - 1, b, nb, c + 1);
22        }
23        if (nb > 0)
24        {
25            *c = *b;
26            spoj(a, na, b + 1, nb - 1, c + 1);
27        }
28    }
29 }
```

Riešenie dopĺňujúcej úlohy č. 4 (funkcia **spoj()** bez použitia cyklov s jednou podmienkou): Podobne ako v riešení dopĺňujúcej úlohy č. 2, aj v prípade rekurzívneho riešenia z dopĺňujúcej úlohy č. 3 môžeme podmienky zlúčiť do jednej. Výsledné riešenie vyzerá pomerne elegantne:

```
1 void spoj(int *a, int na, int *b, int nb, int *c)
2 {
3     if (na > 0 && nb > 0 && *a < *b || nb == 0)
4     {
5         *c = *a; spoj(a + 1, na - 1, b, nb, c + 1);
6     }
7     else
8     {
9         *c = *b; spoj(a, na, b + 1, nb - 1, c + 1);
10    }
11 }
```

Úloha 4-9 (riešenie)

Program po znakoch vypisuje znaky odvodené od reťazcových konštánt.

Prvý vypísaný znak (riadok 3) je znak, ktorého hodnotu dostaneme, keď od prvého znaku reťazca **"abc"** (znak **'a'**, kód 97) odpočítame kód znaku **' '** (medzera, kód 32), teda $97 - 32 = 65$, čo zodpovedá kódu znaku **'A'**.

V riadku 4 program vypíše znak **'e'** posunutý o 3 (znak **'h'**), a znak **'m'** zvýšený o dva (znak **'o'**). V riadku 5 program vypíše posledný jeden znak, ktorý získame komplikovaným výpočtom: prvý znak reťazca **"slnko"** (znak **'s'**, kód 115) znížený o hodnotu znaku **'o'** (kód 48) deleno 5 ($48/5=9$), teda výsledný znak **'j'**.

Ďalšie znaky (ako napr. **:**) už program nevypíše, lebo formátovací reťazec je (predčasne) ukončený znakom **'\0'**.

Úloha 4-10 (riešenie)

V riadku 13 do premennej **x** priradíme smerník na reťazcovú konštantu **"kamen"**. Následne 47-krát vykonaním funkcie **hraj()** a priradením výsledku do **x** upraví hodnotu premennej **x**. Porovnania reťazcov pomocou **==** môžu vzbudzovať podozrenie, že sa reálne neporovnávajú hodnoty reťazcov, ale len smerníky, a to je aj pravda, v tomto programe nie je potrebné porovnávať reťazce po znakoch, pretože stačí porovnať smerníky reťazcových konštánt, ktoré sú rovnaké pre rovnaké reťazce a rôzne pre rozdielne reťazce.

V prípade **x = "kamen"** je výsledok **hraj(x) == "papier"**, v nasledujúcom prípade **x = "papier"** je výsledok **hraj(x) == "noznice"** a v nasledujúcom prípade **x = "noznice"** je výsledok **hraj(x) == "kamen"**, čím premenná **x** (po každých troch volaniach funkcie **hraj()**) nadobudne tú istú hodnotu. Ak riadok 16 vykonáme celkovo 47-krát, tak výsledný účinok bude rovnaký ako by sme ho vykonali len 2-krát (pretože $47 \equiv 2_{\text{mod } 3}$). Program preto nakoniec vypíše reťazec **"noznice"**.

Úloha 4-11 (riešenie)

Funkcia **na_male_pismena()** uvedená v zadání nesprávne spracuje iné znaky ako veľké písmená. Správne riešenie neupravuje znaky, ktoré nie sú písmená alebo už sú

malé písmená, upravuje len veľké písmená. Správne riešenie teda pred úpravou znaku ešte podmienkou skontroluje, či je znak veľké písmeno:

```

1 void na_male_pismena(char *str)
2 {
3     int i;
4     for (i = 0; str[i]; i++)
5         if (str[i] >= 'A' && str[i] <= 'Z')
6             str[i] = str[i] - 'A' + 'a';
7 }
```

Rekurzívne riešenie: V jednom volaní rekurzívnej funkcie spracujeme jeden (prvý) znak podľa rovnakého postupu ako je v iteratívnom riešení: ak je znak veľké písmeno, tak ho upravíme na malé písmeno, iné znaky neupravujeme. Po spracovaní prvého znaku reťazca rekurzívne pokračujeme volaním pre reťazec bez prvého znaku (so začiatkom posunutým o jeden znak ďalej), čím postupne spracujeme každý znak z pôvodného reťazca. Končíme (základný prípad, kedy už rekurzívne volania nie sú potrebné) ak narazíme na ukončovací znak `\0`. Výsledná rekurzívna funkcia je takáto:

```

1 void na_male_pismena(char *str)
2 {
3     if (*str)
4     {
5         if (*str >= 'A' && *str <= 'Z')
6             *str = *str - 'A' + 'a';
7         na_male_pismena(str + 1);
8     }
9 }
```

Úloha 4-12 (riešenie)

Určíme hlavičku funkcie: vstupné argumenty sú podľa zadania počiatočný znak postupky **c** (typ **char**) a požadovaná dĺžka postupky **n** (typ **int**), výstup je návratová hodnota vytvoreného reťazca (typ **char***).

Vytvorený reťazec (postupka) by mala zostať v pamäti aj po ukončení volania funkcie, preto je potrebné v pamäti vyhradiť požadované miesto pre reťazec dĺžky **n** znakov (**str = alloc(n+1)**). Vo výslednom reťazci (**str**) musíme naplniť postupne každý znak **str[i]** pre **i=0, ..., n-1**, a posledný **n**-tý znak musí byť 0 (ukončenie reťazca). Akú hodnotu priradíme do **i**-teho znaku (**str[i]**)?

Do prvého znaku priradíme hodnotu **c** (**str[0] = c**). Do nasledujúceho znaku by sme mali priradiť hodnotu znaku, ktorý nasleduje po **c**. Musíme si uvedomiť, že v prípade, že **c** je posledný znak ('z' alebo 'Z') tak nasledujúci znak nemusí mať ASCII kód **c+1**.

Vo výslednej implementácii funkcie preto najskôr zvýšime hodnotu **c** o 1 a v prípade, že sme presiahli abecedu, tak nastavíme hodnotu **c** opäť na začiatok abecedy 'a' (resp. 'A'). Vo výslednom programe toto zopakujeme **n**-krát:

```
1 char *postupka(char c, int n)
2 {
3     char *str = malloc(n + 1);
4     int i;
5     for (i = 0; i < n; i++)
6     {
7         str[i] = c++;
8         if (c == 'z' + 1)
9             c = 'a';
10        if (c == 'Z' + 1)
11            c = 'A';
12    }
13    str[i] = 0;
14    return str;
15 }
```

Úloha 4-13 (riešenie)

Najskôr určíme hlavičku funkcie. Vstupné argumenty podľa zadania sú: skrytá správa ako reťazec (typ **char***) a hádané písmená tiež ako reťazec (typ **char***). Výstupom je stav hry ako reťazec (typ **char***), ktorý funkcia vráti ako návratovú hodnotu.

Výstupný reťazec (označme **str**), ktorý má funkcia vytvoriť má rovnakú dĺžku ako skrytá správa a niektoré písmená v ňom (tie, ktoré sú v reťazci hádaných znakov, označme **znaky**) zodpovedajú písmenám v skrytej správe a iné (ktoré ešte neboli hádané) majú hodnotu '_' (podčiarkovník).

Najjednoduchší postup preto bude, ak funkcia najskôr zduplicuje pôvodnú správu do nového reťazca (**str = strdup(sprava)**) a potom v reťazci **str** prepíše tie znaky, ktoré nezodpovedajú niektorému z hádaných písmen. Prejdeme teda každý znak reťazca **str** a v prípade, že sa nenachádza v reťazci **znaky** prepíšeme jeho hodnotu (**str[i] = '_'**). Pre vyhľadanie znaku v reťazci môžeme využiť funkciu **strchr()**.

Výsledný upravený reťazec **str** vrátime:

```

1 char *obesenec_stav(char *sprava, char *znaky)
2 {
3     char *str = strdup(sprava);
4     int i;
5     for (i = 0; str[i]; i++)
6         if (!strchr(znaky, str[i]))
7             str[i] = '_';
8     return str;
9 }
```

Úloha 4-14 (riešenie)

Najskôr doplníme hlavičku funkcie. Podľa zadania funkcia vráti počet výskytov, doplníme celé číslo typu **int**, a prostredníctvom výstupného argumentu vráti znak, preto do argumentov v riadku 1 doplníme smerník na znak (typ **char***).

Zrejme musíme prejsť každý znak vstupného reťazca **str**, preto v riadku 4 doplníme podmienku cyklu **str[i]** (čo zodpovedá **str[i] != 0**) a v riadku 6 započítame do poľa **pocet[]** výskyt znaku **str[i]**. Vzhľadom nato, že program spracúva len znaky malej anglickej abecedy (podmienka v riadku 5), môžeme pole počet obmedziť na 26 prvkov (počet anglických písmen od a do z) a v riadku 6 započítať znak do zodpovedajúceho miesta v poli (**pocet[str[i]-'a']++**).

V cykle v riadkoch 7 až 9 sa určuje index znaku s najčastejším výskytom. Priradenie (**max = i**) v riadku 9 program musí vykonať len v prípade, ak početnosť **i**-teho znaku je väčšia ako početnosť doterajšieho maxima (podmienka: **pocet[max] < pocet[i]**). Cyklus bude prebiehať do 26 (**i < 26**), počiatočné hodnoty môžu byť **max=0** a **i=1**, čím zaručíme, že každý index od 0 do 26 bude uvažovaný ako potenciálne maximum.

Nakoniec funkcia uloží najčastejšie sa vyskytujúci znak do výstupného argumentu (***znak = 'a' + max**) a vráti jeho početnosť (**return pocet[max]**).

Výsledný doplnený program je takýto:

```

1 int najcastejsi_znak(char *str, char *znak)
2 {
3     int i, max, pocet[26] = { 0 };
4     for (i = 0; str[i]; i++)
5         if (str[i] >= 'a' && str[i] <= 'z')
6             pocet[str[i] - 'a']++;
```

```

7   for (max = 0, i = 1; i < 26; i++)
8       if (pocet[max] < pocet[i])
9           max = i;
10  *znak = 'a' + max;
11  return pocet[max];
12  }

```

Úloha 4-15 (riešenie)

Najskôr doplníme hlavičku funkcie. Podľa zadania je jediným vstupným argumentom reťazec (typ `char*`), ktorého názov je v programe uvedený v riadkoch 6 a 7 (**str**). Návratová hodnota je počet (typu `int`).

V riadku 3 doplníme deklaráciu identifikátorov použitých premenných (**i**, **j**). Cyklus (riadky 4 až 6) musí prejsť každý znak reťazca. V cykle zvyšujeme riadiacu premennú **i**, preto v podmienke cyklu overíme, či **i**-tý znak vstupného reťazca je platný (logický výraz **str[i]**, ktorý zodpovedá výrazu **str[i] != 0**, ktorý overuje koniec reťazca). Reťazec **str** vo funkcii upravíme tak, že postupne od začiatku do neho priradíme tie znaky, ktoré nie sú malými písmenami (veľké písmená, interpunkcia a pod.), podmienka v riadku 5 overí, či **i**-tý znak reťazca nie je malé písmeno (**str[i] < 'a' || str[i] > 'z'**): ak nie je, v riadku 6 ho priradíme do upraveného reťazca, do ktorého priraďujeme postupne (**str[j++] = str[i]**) od začiatku (začíname pre **j = 0**). Premenná **j** označuje prvé miesto v poli **str[]**, do ktorého funkcia ešte nepridelila znak, a preto keď v podmienke v riadku 6 funkcia rozhodne, že **i**-tý znak by mal v reťazci zostať, tak ho priradí na miesto **str[j]** a index **j** zvýši o 1 (na ďalšie miesto). Nakoniec je potrebné do poľa **str[]** priradiť ukončovací kód (**str[j] = 0**) čím sa reťazec ukončí. Funkcia vráti počet zapísaných znakov (**j**).

Výsledná doplnená funkcia podľa uvedeného opisu je takáto:

```

1  int odstran_male_pismena(char *str)
2  {
3      int i, j;
4      for (i = j = 0; str[i]; i++)
5          if (str[i] < 'a' || str[i] > 'z')
6              str[j++] = str[i];
7      str[j] = 0;
8      return j;
9  }

```

Rekurzívne riešenie: Rekurzívna funkcia spracuje prvý znak reťazca na vstupe (**str**). Ak je prvý znak iný ako písmeno malej anglickej abecedy, tak ho v reťazci ponechá a rekurzívne sa zavolá na skrátený reťazec bez prvého písmena s tým, že výsledná návratová hodnota bude o 1 vyššia ako získaná návratová hodnota z kratšieho reťazca (pretože sme písmeno ponechali). V prípade, že prvý znak je písmeno malej anglickej abecedy, tak ho odstránime a rekurzívne sa zavoláme na zvyšok reťazca. Ako odstránime prvý znak z reťazca tak, že ostatné znaky posunieme o jednu pozíciu v poli skôr, pričom pole musíme aj skrátiť (umiestniť ukončovaci kód o jednu pozíciu skôr)?

Na odstránenie jedného znaku použijeme pomocnú rekurzívnu funkciu **odstran_znak()**, ktorá odstráni znak tak, že hodnotu nasledujúceho znaku priradí do aktuálneho znaku, pričom sa rekurzívne zavolá, čo v konečnom dôsledku posunie každý znak od odstráneného znaku ďalej o jednu pozíciu skôr, vrátane ukončovacieho kódu `\0`, čo skráti dĺžku reťazca o 1 (odstránenie jedného znaku).

Výsledná implementácia rekurzívnej funkcie podľa uvedeného opisu:

```
1 void odstran_znak(char *str)
2 {
3     if (*str == 0)
4         return;
5     *str = *(str + 1);
6     odstran_znak(str + 1);
7 }
8
9 int odstran_male_pismena(char *str)
10 {
11     if (*str == 0)
12         return 0;
13     if (str[0] < 'a' || str[0] > 'z')
14         return 1 + odstran_male_pismena(str + 1);
15     odstran_znak(str);
16     return odstran_male_pismena(str);
17 }
```

Úloha 4-16 (riešenie)

Najskôr určíme hlavičku funkcie: vstupné argumenty sú hľadaná správa (typ **char***) a stav hry (typ **char***), ako návratovú hodnotu funkcia vráti nový reťazec zodpovedajúci písmenám (typ **char***), ktoré treba ešte pre dokončenie hry uhádnuť.

Zadanie neuvádza aký typ znakov hľadaná správa obsahuje, a teda môže obsahovať v princípe akékoľvek znaky. Výstupom preto môže byť reťazec obsahujúci až 255 rôznych znakov a pre premennú, ktorá bude výsledok reprezentovať, musíme vyhraďiť pole dĺžky 256 znakov (**char buf[256]**). Potom budeme prechádzať stav hry (reťazec) v cykle po znakoch a znaky, ktoré treba ešte uhádnuť, zapisovať do výsledného reťazca **buf**. Ak je **i**-ty znak stavu hry podčiarkovník (znak **'_'**) a zároveň sme ešte zodpovedajúci znak z hľadanej správy do **buf** nezapísali tak ho zapíšeme (**buf[n++] = sprava[i]**).

Ako zistíme, či sa už nejaký znak nachádza vo vytváranom reťazci **buf**? Jedna možnosť je reťazec **buf** opäť prejsť a znak v ňom vyhľadať (napr. funkciou **strchr()**, pričom **buf** musí byť ukončený kódom 0). Efektívnejšia možnosť, ktorá nevyžaduje pre každý znak prechádzať reťazec **buf**, použije pomocné pole dĺžky 256, v ktorom si pre každý znak pamätáme hodnotu 0 ak ešte nebol znak zapísaný do **buf**, alebo hodnotu 1 ak už bol zapísaný do **buf**. Na začiatok pole inicializujeme na 0.

Výsledné riešenie podľa uvedeného opisu, ktoré využíva pole príznakov:

```
1 char *obesenec_riesenie(char *sprava, char *stav)
2 {
3     char buf[256];
4     int i, n = 0, znak[256] = { 0 };
5
6     for (i = 0; stav[i]; i++)
7         if (stav[i] == '_' && znak[sprava[i]] == 0)
8         {
9             buf[n++] = sprava[i];
10            znak[sprava[i]] = 1;
11        }
12    buf[n] = 0;
13    return strdup(buf);
14 }
```

Úloha 4-17 (riešenie)

Uvedené riešenie nesprávne prekopíruje ukončovaci kód **0**. Cyklus v riadku 4 začína na poslednom znaku, ktorí priradí do ukončovacieho kódu, ktorý už v upravenom reťazci nebude. Chybu môžeme odstrániť napr. tak, že cyklus začneme už pre hodnotu **i = n**. Upravené (správne) riešenie je takéto:

```
1 void vlož_do_stredu(char *str, char c)
2 {
3     int i, n = strlen(str);
4     for (i = n; i >= n / 2; i--)
5         str[i + 1] = str[i];
6     str[n / 2] = c;
7 }
```

Úloha 4-18 (riešenie)

Hlavička funkcie priamo vyplýva zo zadania: vstupné argumenty sú reťazec **str** (typ **char***), celé číslo **n** (typ **int**) a celé číslo **offset** (typ **int**), výstupný argument bude reťazec **str** a návratová hodnota je celé číslo.

Zamyslime sa, v akých prípadoch nie je možné **n** znakov od pozície **offset** odstrániť? Označme **len** dĺžku vstupného reťazca **str**. Znakov nie je možné odstrániť vtedy, ak od pozície **offset** reťazec obsahuje menej ako **n** znakov (**len - offset < n**), funkcia vráti 1. V prípade, že znaky možno odstrániť, tak cyklom od pozície **i = offset + n** prejdeme znaky (**str[i]**) po jednom až do konca (vrátane ukončovacieho znaku **\0**) a každý presunieme o **n** pozícií skôr (**str[i - n] = str[i]**). Presný postup je uvedený v nasledujúcej implementácii:

```
1 int odstran_znaky(char *str, int n, int offset)
2 {
3     int i, len = strlen(str);
4     if (len - offset < n)
5         return 1;
6     for (i = offset + n; i <= len; i++)
7         str[i - n] = str[i];
8     return 0;
9 }
```

Úloha 4-19 (riešenie)

Najskôr určíme hlavičku funkcie: vstupné argumenty sú pole znakov **dst** (typ **char***) dĺžky **len** (typ **int**), reťazec **src** (typ **char***) a **offset** (typ **int**), výstupné argumenty sú pole **dst** a návratová hodnota je celé číslo (typ **int**). Reťazec **src** je len vstupný, takže mu pre prehľadnosť môžeme pridať modifikátor **const**.

Pole znakov **dst** obsahuje reťazec neznámej dĺžky, označme ju **d1**, dĺžku reťazca **src** označme **s1**.

Kedy nie je možné reťazec **src** vložiť do reťazca v poli **dst** od pozície **offset**? V prípade, že **offset** je za koncom reťazca (**offset > dl**) alebo v prípade, že súčet dĺžok reťazcov je väčší alebo rovný (kvôli ukončovaciemu znaku **\0**) ako je dĺžka poľa **dst** (**dl + sl >= len**). V týchto prípadoch funkcia nevykoná žiadnu úpravu **dst** a vráti 1 (nepodarilo sa reťazec vložiť).

V opačnom prípade musí najskôr vytvoriť v reťazci **dst** miesto pre **sl** znakov a až potom do vzniknutej medzery vložiť znaky reťazca **src**. Voľné miesto pre **sl** znakov vytvoríme postupným posunom znakov reťazca **dst** od konca, tak že každý posunieme (priradíme jeho hodnotu) do pozície o **sl** znakov ďalej. Presný postup je uvedený v nasledujúcej implementácii:

```
1 int vlož_retazec(char *dst, int len, const char *src, int offset)
2 {
3     int dl = strlen(dst), sl=strlen(src);
4     if (offset > dl || dl + sl >= len)
5         return 1;
6     int i;
7     for (i = dl+sl; i >= offset+sl; i--)
8         dst[i] = dst[i-sl];
9     for (i = 0; i < sl; i++)
10        dst[offset+i] = src[i];
11     return 0;
12 }
```

Úloha 4-20 (riešenie)

- a) Implementácia pomocou cyklu priradí najviac **n-1** znakov z reťazca **src** do **dst** a ukončí ho **\0**:

```
1 void my_strncpy(char *dst, const char *src, int n)
2 {
3     int i = 0;
4     n--; // rezervuj pre ukoncovaci znak \0
5     while (i < n && *src)
6     {
7         dst[i++] = *src;
8         src++;
9     }
10    dst[i] = 0;
11 }
```

- b) Implementácia je podobná ako v prípade a). Rozdiel je v počiatočnom indexe (**i**), od ktorého sa znaky vo výstupnom reťazci **dst** priraďujú. Pôvodný obsah reťazca **dst** (ak je dlhý aspoň **n** znakov) zostane preto zachovaný:

```
1 void my_strncat(char *dst, const char *src, int n)
2 {
3     int i = strlen(dst);
4     n--; // rezervuj pre ukoncovaci znak \0
5     while (i < n && *src)
6     {
7         dst[i++] = *src;
8         src++;
9     }
10    dst[i] = 0;
11 }
```

Riešenie doplňujúcej úlohy: S použitím pomocného poľa je implementácia priamočiara: najskôr zdublikuj reťazec **src** do pomocného reťazca **tmp**, a potom tento pomocný reťazec nakopíruj do cieľového **dst**:

```
1 char *my_strcpy(char *dst, const char *src)
2 {
3     char *tmp = strdup(src);
4     strcpy(dst, tmp);
5     free(tmp);
6     return dst;
7 }
```

Môže byť kopírovanie reťazcov, ktoré sa v pamäti prekrývajú, implementované bez využitia pomocného poľa? Odpoveď na túto otázku je komplikovanejšia. Na prvý pohľad sa zdá, že môžeme porovnať vzájomnú polohu smerníkov **dst** a **src** v pamäti a určiť charakter prekryvu: či je **src** pred **dst** alebo opačne a podľa toho prispôbiť (smer) kopírovania znakov. Určite je to zaujímavé mentálne cvičenie.

Problém tohto prístupu je ten, že funguje spoľahlivo (teda podľa štandardu ANSI C) len vtedy, ak **src** a **dst** skutočne pochádzajú z rovnakého reťazca (takže napr. **strcpy(buf+3, buf)**). V okamihu, keď reťazce sú rôzne (čo je asi najčastejšia situácia) spôsobí porovnanie smerníkov pochádzajúcich z rôznych „objektov“ nedefinované správanie programu. Inými slovami na rôznych počítačoch sa program môže správať rôzne, a preto takýto program určite nie je správny.

Jediná správna alternatíva je trochu triková: treba použiť funkciu na kopírovanie pamäte **memmove()**, ktorá podľa štandardu jazyka ANSI C je (jediný) platný spôsob kopírovania potenciálne prekrywajúcej sa pamäte. Teda **memmove()** (na rozdiel od **strcpy()**) garantovane kopíruje pamäť (reťazec) správne aj v prípade prekryvania kopírovaných blokov:

```
1 char *my_strcpy(char *dst, const char *src)
2 {
3     memmove(dst, src, strlen(src) + 1);
4     return dst;
5 }
```

Úloha 4-21 (riešenie)

V riadku 3 doplníme chýbajúcu premennú **prenos**. Pri výpočte súčtu budeme postupovať zaužívaným spôsobom postupne od najnižších rádov s tým, že súčet nad 10 sa prenese do vyššieho rádu.

Priebežný súčet na aktuálnom ráde si budeme počítat v premennej **prenos**. Jediná príležitosť, kde môžeme do čísla **a** priradiť novú (pripočítanú) hodnotu je v riadku 7, teda poslednú cifru priebežného súčtu priradíme ako **i**-tu cifru čísla **a** (**a[i] = prenos%10**). V riadku 6 pripočítame k prenosu z minulého rádu hodnotu súčtu na aktuálne spracovanom ráde (**prenos += a[i] + b[i]**):

```
1 void pripocitaj(char *a, const char *b) // a += b
2 {
3     int i, prenos = 0;
4     for (i = 0; i < MAX_DLZKA_CISLA; i++)
5     {
6         prenos += a[i] + b[i];
7         a[i] = prenos % 10;
8         prenos /= 10;
9     }
10 }
```

Úloha 4-22 (riešenie)

Využijeme v tejto kapitole uvedenú reprezentáciu dlhých čísel ako cifry v desiatkovej sústave od najnižšieho rádu v poli znakov konštantnej dĺžky.

Najskôr určíme hlavičku funkcie: vstupný argument bude reťazec **str** obsahujúci aritmetický výraz (typ **char***) a ako návratová hodnota bude dlhé číslo **x** (pole znakov) typ **char***.

Pri výpočte výrazu budeme postupovať zaužívaným spôsobom zľava doprava. Na začiatok hodnotu priebežného výsledku **x** inicializujeme na 0, postupne budeme prechádzať čísla zľava doprava a pripočítavať ich k priebežnému výsledku (súčtu). Postupovať budeme po znakoch: spracujeme každý znak reťazca **str**: ak je znak **str[i]** číslíca, tak ju pridáme na koniec pomocného reťazca **num**, v ktorom si ukladáme dlhé číslo ako reťazec. Ak narazíme na znak **+** (alebo nasleduje koniec reťazca), tak predchádzajúce číslo (**num**) je ukončené a pripočítame ho priebežnému výsledku, využijeme implementáciu funkcie **pripocitaj()**:

```
1 char *plus_vyraz(char *str)
2 {
3     char *x = dlhecislo("0");
4     char num[MAX_DLZKA_CISLA];
5     int i, k;
6
7     for (i = k = 0; str[i]; i++)
8     {
9         if (str[i] >= '0' && str[i] <= '9')
10            num[k++] = str[i];
11        if (k > 0 && (str[i] == '+' || str[i+1] == 0))
12        {
13            num[k] = 0; // ukonci predchadzajuce cislo
14            char *y = dlhecislo(num);
15            pripocitaj(x, y); // x += y
16            k = 0;
17        }
18    }
19
20    return x;
21 }
```

Úloha 4-23 (riešenie)

Postupovať budeme obvyklým spôsobom. V riadku 3 doplníme chýbajúcu premennú **prenos**. Pri výpočte súčinnu budeme postupovať od najnižších rádov s tým, že súčinn nad 10 sa preniesie do vyššieho rádu.

Priebežný súčinn na aktuálnom ráde si budeme počítvať v premennej **prenos**. Do dlhého čísla **a** priradíme v riadku 7 poslednú cifru priebežného súčinnu ako **i**-tu cifru (**a[i] = prenos%10**).

V riadku 6 pripočítame k prenosu z minulého rádu hodnotu súčinnu na aktuálne spracovanom ráde (**prenos += num * a[i]**). Výsledná doplnená funkcia:

```
1 void nasob_int(char *a, int num) // a *= num
2 {
3     int i, prenos = 0;
4     for (i = 0; i < MAX_DLZKA_CISLA; i++)
5     {
6         prenos += num * a[i];
7         a[i] = prenos % 10;
8         prenos /= 10;
9     }
10 }
```

Úloha 4-24 (riešenie)

Najskôr určíme hlavičku funkcie: vstupný argument bude reťazec **str** obsahujúci binárne číslo (typ **char***) a ako návratová hodnota bude dlhé číslo **x** (pole znakov) typu **char***.

Pri výpočte budeme postupovať analogickým spôsobom ako v prípade, že by sme výsledok reprezentovali bežným celým číslom typ **int**. V binárnom čísle postupne spracujeme binárne cifry od najnižšieho rádu: ak je príslušná cifra na **i**-tom ráde 1, tak do desiatkového zápisu pripočítame hodnotu mocniny 2^i . Mocniny si budeme pre **i = 0, ..., n-1** (kde **n** je počet bitov čísla v reťazci **str**) priebežne počítvať v dlhom čísle **bin**: pri každom spracovanom ráde hodnotu **bin** využitím funkcie **prinasob_int()** vynásobíme číslom 2 a dostaneme hodnotu mocniny 2^{i+1} .

Výsledná implementácia je takáto:

```

1 char *binarne(const char *str)
2 {
3     char *bin = dlhecislo("1"), *x = dlhecislo("0");
4     int i, n = strlen(str);
5     for (i = n - 1; i >= 0; i--)
6     {
7         if (str[i] == '1')
8             pripocitaj(x, bin); // x += bin
9         nasob_int(bin, 2); // bin *= 2
10    }
11    return x;
12 }

```

Úloha 4-25 (riešenie)

Postupovať budeme obvyklým spôsobom násobenia dlhých čísel: číslo **a** vynásobíme postupne jednotlivými ciframi čísla **b** a pripočítame s posunutím o príslušný rád k celkovému výsledku (**x**). Pri výpočte súčinu postupujeme v dvoch vnorených cykloch od najnižších rádo (i-tý rád v čísle **a**, j-tý rád v čísle **b**) s tým, že súčin nad 10 sa prenesie do vyššieho rádu. V riadku 7 v cykle prejdeme postupne cifry čísla **i** od najnižšieho rádu. Pribežný súčin na aktuálnom ráde si počítame v premennej **prenos**, inicializujeme na hodnotu 0. Do výsledku **x** priradíme v riadku 10 poslednú cifru pribežného súčinu ako (**i+j**)-tu cifru (**x[i+j] = prenos%10**). V riadku 9 pripočítame k prenosu z minulého rádu hodnotu súčinu na aktuálne spracovanom ráde a aktuálnu cifru z výsledku **x** (**prenos += a[i] * b[j] + x[i+j]**).

Výsledná doplnená funkcia je takáto:

```

1 void prinasob(char *a, const char *b) // a *= b
2 {
3     char x[2 * MAX_DLZKA_CISLA] = { 0 }; // vysledok
4     int i, j, prenos;
5     for (j = 0; j < MAX_DLZKA_CISLA; j++)
6     {
7         for (prenos = i = 0; i < MAX_DLZKA_CISLA; i++)
8         {
9             prenos += a[i] * b[j] + x[i+j];
10            x[i+j] = prenos % 10;
11            prenos /= 10;
12        }
13    }
14    for (i = 0; i < MAX_DLZKA_CISLA; i++)
15        a[i] = x[i];
16 }

```

Úloha 4-26 (riešenie)

Pri výpočte faktoriálu pre $n = 100\,000$ rýchlo vzniknú dlhé čísla v rozsahu niekoľko sto tisíc cifier. Implementácia dlhých čísiel pomocou polí konštantnej dĺžky je už pri malých číslach pomalá kvôli veľkej maximálnej dĺžke čísel. Implementáciu urýchlime tým, že si dĺžku čísla (počet platných cifier v poli) budeme explicitne pamätať a potrebné operácie vykonávať len pre potrebný počet platných cifier.

Z operácií s dlhými číslami potrebujeme v programe implementovať iba operáciu prinásobenia: $a *= i$. Výpočet postupuje ako v prípade dlhých čísel s konštantnou dĺžkou poľa ale s rozdielom, že v cykle násobenie pokračuje dovtedy, kým nedosiahneme rád zodpovedajúci aktuálnej dĺžke čísla, alebo kým je hodnota prenosu nenulová, čím môžeme dĺžku zvýšiť.

Čas výpočtu programu meriame funkciou `clock()` z knižnice `time.h`, ktorá vracia relatívnu hodnotu času: počet cyklov procesora od spustenia počítača. Na odmeranie konkrétneho času v štandardných jednotkách (sekundy) je potrebné vrátené hodnoty od seba odpočítať a normalizovať konštantou `CLOCKS_PER_SEC`.

Výsledná implementácia výpočtu, ktorá sa zastaví po prekročení 1 min:

```
1  #include <stdio.h>
2  #include <time.h>
3
4  char c[500000];
5  int n;
6
7  int main(void)
8  {
9      int i, f=100000, j, v, prenos;
10     clock_t start = clock();
11     double time;
12
13     c[0] = n = 1;
14     for (i = 2; i <= f; i++)
15     {
16         // vypocet: c = c*i
17         for (j = v = prenos = 0; j < n || prenos > 0; j++)
18         {
19             prenos += i * c[j];
20             c[j] = prenos % 10;
21             prenos /= 10;
22         }
23         n = j;
24         time = (double)(clock() - start) / CLOCKS_PER_SEC;
```

```
25     if (time > 60.0)
26         break;
27     }
28     for (j = n - 1; j >= 0; j--)
29         printf("%d", c[j]);
30     printf("\n");
31     return 0;
32 }
```

Rýchlosť výpočtu je na rôznych počítačoch rôzna. V tomto prípade nie je výpočet obmedzený veľkosťou dostupnej operačnej pamäte, pretože nám postačuje relatívne malé pole 500 000 znakov (približne 0,5 MB). Najvýznamnejší činiteľ, ktorý ovplyvní rýchlosť výpočtu, je rýchlosť procesora (CPU) a v prípade, že sa nezmestí do vyrovnávacej pamäte (cache) procesora, tak aj rýchlosť operačnej pamäte. Dokonca každé jedno spustenie rovnakého programu môže trvať (trochu) iný čas v závislosti od aktuálneho zaťaženia operačného systému inými bežiacimi procesmi.

Výpočty sme spúšťali na počítači s procesorom i7-4702MQ (2,2 GHz) z roku 2013 s pamäťou 8GB DDR3L 1600. Bez ďalších optimalizácií programu kompilátorom čas výpočtu programu prekročil 60 s po výpočte hodnoty 45 414! Hodnotu 100 000! vypočítal až za 332 s. So zapnutými optimalizáciami kompilátora program za 60 s vypočítal hodnotu 99 274!

Akú hodnotu sa podarilo vypočítať do 60 s vám? Skúste to porovnať s našim počítačom (z roku 2013), ako odvtedy pokročil vývoj? Vytvorte tabuľku, v ktorej znázorníte závislosť času výpočtu (potrebného pre výpočet hodnoty $n!$) a hodnoty n .

Uloha 4-27 (riešenie)

V riadku 4 priradíme prostredníctvom smerníka na smerník (**pp**) do premennej **a** hodnotu 33, v riadku 5 ju prostredníctvom smerníka (**p**) zdvojnásobíme, v riadku 6 ju opäť zdvojnásobíme. Výsledná hodnota bude $33 \cdot 2 \cdot 2 = 132$.

Uloha 4-28 (riešenie)

Predtým ako budeme do celočíselných premenných priraďovať hodnoty v riadkoch 5 až 7 si všimnime riadok 4, v ktorom zmeníme hodnotu, na ktorú ukazuje **pp** (ukazuje na smerník **q**, pozri riadok 3) na adresu premennej **b** – smerník **q** teda bude

ukazovať na premennú **b**. V riadku 5 priradíme prostredníctvom smerníku (**p**) do premennej **a** hodnotu 5. V riadku 6 priradíme prostredníctvom smerníku (**q**) do premennej **b** hodnotu 2. V riadku 7 nakoniec prostredníctvom smerníku na smerník (**pp**) priradíme do premennej **b** hodnotu $2 * 5 = 10$.

Úloha 4-29 (riešenie)

Najskôr určíme hlavičku funkcie: pole celých čísel má typ **int***, preto výstupný argument pre takéto pole bude mať typ **int****. Počet čísel (návratová hodnota) bude typu **int**, iné argumenty nepotrebujeme.

Nevieme vopred počet čísel, preto budeme pamäť pre výsledné pole dynamicky alokovať postupne podľa počtu načítaných čísel. Predtým ako nejaké načítané číslo priradíme do pamäte, musíme mať túto pamäť vyhradenú. Na začiatok nemáme pre výsledné pole pridelenú žiadnu pamäť a v prípade, že na vstupe nie sú žiadne čísla ani žiadnu pamäť prideliť nechceme. V programe teda pri načítavaní budeme priebežne zväčšovať pole čísel (**pole[]**) s tým, že si budeme pamätať počet prvkov (**n**), ktorý máme pre **pole[]** v pamäti vyhradený (na začiatok **n=0**).

Po úspešnom načítaní ďalšieho čísla do pomocnej premennej (**cislo**) musíme pred priradením na posledné miesto poľa **pole[]** skontrolovať, či máme v poli ešte nepoužívané prvky. Ak už boli všetky prvky poľa **pole[]** naplnené hodnotou (**i == n**) musíme pole zväčšiť, aby sme mohli na ďalšie miesto pridať práve načítanú hodnotu (**cislo**). Ako zväčšíme **pole[]**?

Priebežné zväčšenie poľa môžeme realizovať funkciou **realloc()**, ktorá existujúcemu smerníku „pridelí“ novú (väčšiu alebo menšiu) pamäť. V skutočnosti **realloc()** dynamicky alokuje novú pamäť, do ktorej prekopíruje obsah existujúcej pamäte (v tomto prípade, poľa **pole[]**), starú pamäť uvoľní a vráti smerník na novú pridelenú pamäť. Takáto operácia je pomerne náročná pre väčší počet existujúcich prvkov v poli, preto je dobré ju vykonávať čo najmenej. Presný počet prvkov nevieme, preto musíme počet priebežne zväčšovať. Zvyčajne sa pole zväčšuje o nejakú konštantnú dĺžku odhadnutú podľa charakteru dát, napr. o 10 prvkov alebo o 100 prvkov. Pokiaľ počet načítaných prvkov môže túto odhadnutú konštantnú dĺžku

výrazne prekročiť, tak najlepšie výsledky (najmenej realokácií a teda najrýchlejšie načítanie) dosiahneme ak dĺžku poľa budeme zdvojnásobovať.

Výsledný program s implementáciou funkcie **nacitaj_pole()** podľa uvedeného opisu ako aj ukážkou volania v hlavnej funkcii **main()**:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int nacitaj_pole(int **vysledok)
5  {
6      int i = 0, n = 0, *pole = NULL, cislo;
7      for (i = n = 0; scanf_s("%d", &cislo) == 1; i++)
8      {
9          if (i == n)
10         {
11             n = 2 * n + 1;
12             pole = realloc(pole, n * sizeof(int));
13         }
14         pole[i] = cislo;
15     }
16     *vysledok = pole;
17     return i;
18 }
19
20 int main(void)
21 {
22     int i, pocet, *pole;
23     pocet = nacitaj_pole(&pole);
24     printf("Pocet cisel: %d\n", pocet);
25     for (i = 0; i < pocet; i++)
26         printf("%d\n", pole[i]);
27     return 0;
28 }
```

Úloha 4-30 (riešenie)

Určíme najskôr hlavičku funkcie: vstupné argumenty sú vkladané číslo (**cislo**), pole **a[]**, do ktorého vkladáme, počet prvkov **n** poľa **a[]** ako aj dĺžka poľa **a[]** argument **kapacita**. Aké budú výstupné argumenty? Výstupné argumenty by mali zahŕňať tie položky, ktoré funkcia môže pri svojej činnosti upraviť: načítané pole môže zväčšiť (výstupný argument pre pole **a[]** bude typu **int ****) a teda upraviť jeho veľkosť **n** (typ **int***) ako aj dĺžku **kapacita** (typ **int***). Pole, jeho veľkosť a dĺžka sú zároveň vstupné aj výstupné argumenty: čo sa týka použitého typu, v argumentoch funkcie

budú uvedené ako výstupné argumenty (aby sme do nich mohli zapisovať) ale budeme ich používať aj ako vstupné argumenty (budeme z nich hodnoty aj čítať).

V prípade, že by číslo, ktoré do poľa vkladáme, malo presiahnuť dĺžku poľa ($n+1 > \text{kapacita}$), tak pole zväčšíme funkciou `realloc()`. V tomto riešení demonštrujeme zväčšenie poľa o konštantný prírastok (o 10 prvkov) lebo predpokladáme, že funkcia sa bude v programe volať len sporadicky.

Potom ako funkcia `vloz()` zaručí, že pole má dostatočnú kapacitu na pridanie ďalšieho prvku je potrebné nájsť umiestnenie nového prvku vzhľadom na usporiadanie prvkov v poli. Ak správne miesto nájdeme, je možné, že prvky, ktoré sa nachádzajú za týmto miestom budeme musieť posunúť na nasledujúce miesto v poli, čím vytvoríme miesto pre nový prvok, do ktorého priradíme novú hodnotu. Posúvanie prvkov o jedno miesto ďalej môžeme realizovať v cykle súčasne s hľadaním správneho miesta pre pridávaný prvok tak, že prvky poľa budeme prechádzať (*i*) od konca (posledného prvku) smerom na začiatok (prvý prvok): ak hodnota *i*-teho prvku je menšia ako pridávaná hodnota (*cislo*), tak vieme že potom ako *i*-tý prvok presunieme na *i+1* miesto (`a[i+1] = a[i]`), tak na uvoľnené *i*-te miesto vložíme hodnotu *cislo*, inak pokračujeme ďalej pre prvok *i-1* (ktorý presunieme na *i*-te miesto atď).

Výsledná implementácia funkcie `vloz()` podľa uvedeného opisu:

```
1 void vloz(int cislo, int **a, int *n, int *kapacita)
2 {
3     int i;
4     if ((*n) + 1 > *kapacita)
5     {
6         *kapacita += 10;
7         *a = realloc(*a, *kapacita * sizeof(int));
8     }
9
10    for (i = (*n) - 1; i >= 0; i--)
11    {
12        if ((*a)[i] < cislo)
13            break;
14        (*a)[i + 1] = (*a)[i];
15    }
16    (*a)[i + 1] = cislo;
17    (*n)++;
18 }
```

Pripomeňme ešte, že v prípade, keď túto funkciu budeme na jednej postupnosti volať veľký počet krát: napr. ak bude použitá ako usporadúvací algoritmus pre dlhé postupnosti, tak je výrazne efektívnejšie dĺžku poľa pri predlžovaní zdvojnásobovať. Napr. pri 10 000 prvkoch by sme ju museli predĺžiť 1000-krát po 10 a vykonať približne $1000 * 5000$ operácií, naopak v prípade zdvojnásobenia by sme ju predlžovali len 14-krát a vykonali len zhruba $14 * 5000$ operácií.

Úloha 4-31 (riešenie)

Určíme najskôr hlavičku funkcie: pre vstupné pole reťazcov **s** (typ **char****) musíme na vstupe dostať aj počet reťazcov **n** (typ **int**), oddeľovací znak **z** je typu **char**, a výsledný reťazec vrátime ako návratovú hodnotu typu **char***.

Funkcia **spoj_retazce()** vytvorí jeden dlhý reťazec. Pre výsledný reťazec musíme dynamicky alokovať miesto v pamäti. Akú veľkú pamäť musíme vyhradiť?

Celková dĺžka reťazca bude súčet dĺžok reťazcov v poli reťazcov **s** spolu s oddeľovacími znakmi medzi nimi: pre **n** (**n > 0**) reťazcov, potrebujeme **n-1** oddeľovacích znakov. Špeciálny prípad je situácia pre **n = 0**, keď je výsledný reťazec prázdny. Funkcia teda najskôr určí výslednú dĺžku reťazca, dynamicky alokuje potrebné miesto a potom postupne vloží vstupné reťazce z poľa **s** znak po znaku do výsledného reťazca. Oddeľovací znak vloží vždy pred vkladáním ďalšieho reťazca.

Výsledná implementácia funkcie **spoj_retazce()** spolu s ukázkou volania v hlavnej funkcii **main()**:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 char *spoj_retazce(char **s, int n, char z)
6 {
7     int i, j, len = n - 1;
8     char *str;
9
10    if (n <= 0)
11        return calloc(1, 1); // prazdny retazec
12    for (i = 0; i < n; i++)
13        len += strlen(s[i]);
14    str = malloc(len + 1);
```

```

15   for (i = len = 0; i < n; i++)
16   {
17       if (i > 0)
18           str[len++] = z;
19       for (j = 0; s[i][j]; j++)
20           str[len++] = s[i][j];
21   }
22   str[len++] = 0;
23   return str;
24 }
25
26 int main(void)
27 {
28     char *s[] = { "Ahoj", "mily", "svet!" };
29     printf("%s\n", spoj_retazce(s, 3, '+'));
30     return 0;
31 }

```

Úloha 4-32 (riešenie)

Analyzujeme chýbajúce miesta v programe a pokúsime sa doplniť najskôr tie miesta, pre ktoré jednoznačne vyplýva, čo do nich treba doplniť.

Program načíta a spracuje zo vstupu každé číslo po jednom, preto v riadku 21 doplníme načítanie `scanf("%d", &n) == 1`. V riadkoch 23 až 25 musíme alokovať dvojrozmerné pole znakov veľkosti $n \times n$, preto doplníme alokáciu zaužívaným spôsobom pomocou funkcie `malloc()`.

Vo funkcii `hadik()` naplníme alokované pole znakmi. Riadok 7 musí obsahovať cyklus pre i (doplníme `for (i = 0; i < n; i++)`) aby hodnota nadobudla v riadkoch 10 až 13 platné hodnoty. V riadku 11 naplníme hodnoty pre nepárne riadky (číslujeme od 0, preto riadok 0 je „prvý“, riadok 2 je „tretí“) hadíka, v riadku 12 pre párne riadky.

Všimnime si najskôr nepárne riadky: so stúpajúcou hodnotou j znak stúpa. Určíme najskôr offset (poradie) znaku, ktorý prvý vypíšeme v i -tom riadku: v riadkoch 0, ..., $i-1$ sme už vypísali $n \cdot i$ znakov (s offsetmi 0, ..., $n \cdot i - 1$), preto prvý znak na i -tom riadku bude mať offset $n \cdot i$, po zohľadnení dĺžky použitej abecedy (26 znakov) to bude znak `'a' + (n*i)%26`. Pre zvyšujúci sa stĺpec j , to bude znak `'a' + (n*i+j)%26`. V párnych riadkoch sa znaky s pribúdajúcimi stĺpcami

„znižujú”, takže začneme od offsetu $(n+1)*i$ a pre rastúce j budeme hodnotu postupne znižovať: `'a' + (n*i+n+j-1)%26`.

V riadkoch 29 až 34 prebieha výpis výsledku, preto v riadku 32 doplníme výpis znaku z dvojrozmerného poľa, ktoré sme naplnili: `printf("%c", a[i][j]);`

Výsledný doplnený program podľa uvedeného opisu:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void hadik(char **a, int n)
5  {
6      int i, j;
7      for (i = 0; i < n; i++)
8          for (j = 0; j < n; j++)
9              {
10                 if (i % 2 == 0)
11                     a[i][j] = 'a' + (n*i+j) % 26;
12                 else
13                     a[i][j] = 'a' + (n*i+n-j-1) % 26;
14             }
15  }
16
17  int main(void)
18  {
19      int i, j, n;
20      char **a;
21      while (scanf("%d", &n) == 1)
22      {
23          a = malloc(sizeof(char*)*n);
24          for (i = 0; i < n; i++)
25              a[i] = malloc(n);
26
27          hadik(a, n);
28
29          for (i = 0; i < n; i++)
30              {
31                  for (j = 0; j < n; j++)
32                      printf("%c", a[i][j]);
33                  printf("\n");
34              }
35      }
36      return 0;
37  }
```

Úloha 4-33 (riešenie)

Analyzujeme chýbajúce miesta v programe a pokúsime sa doplniť najskôr tie miesta, pre ktoré jednoznačne vyplýva, čo do nich treba doplniť.

V programe je v riadku 10 použité dvojrozmerné pole **g**, v ktorom prvku **g[i/10][i%10]** zvýšime hodnotu o 1. Používame premennú **i**, ktorá musí mať platnú hodnotu, preto v cykle v riadku 8 musí premenná **i** nadobudnúť platnú hodnotu. Riadok 8 obsahuje formátovací reťazec, ktorý naznačuje, že ide o načítanie čísel zo vstupu, preto doplníme názov funkcie **scanf()** a načítavať budeme do premennej **i**. Na vstupe sú celé čísla bez uvedenia počtu, preto budeme čítať do konca vstupu (doplníme **> 0**). Ďalej doplníme definíciu dvojrozmerného poľa **g** v riadku 3, vzhľadom na interval čísel na vstupe (od 0 do 99) a zvyšovanie **g** v riadku 10 musí mať pole rozmery 10 x 10.

Ako program vypíše histogram na výstup? V histograme nie sú riadky, v ktorých by nebolo uvedené číslo, preto hlavičku riadku ("**%d |**") vypíšeme len pre tie riadky, pre ktoré sme na vstupe načítali číslo. Použijeme nato premennú **c**, ktorá bude pre **i**-tý riadok obsahovať počet čísel zo vstupu, ktoré patria do **i**-teho riadku histogramu (v riadku 11 doplníme **i/10**) a v definícii v riadku 3 (**c[10]**). Doplníme tiež výpis hlavičky riadkov: hranice cyklus pre **i** v riadku 13 sú **i = 0** až **i < 10**. Hlavičku pre **i**-tý riadok vypíšeme len, ak riadok obsahuje nejaké čísla (v riadku 14 doplníme **c[i] > 0** resp. postačuje **c[i]**). V riadku 16 vypíšeme hlavičku **i**-teho riadku: doplníme **printf** a vypíšeme hodnotu premennej **i** (všimnime si, že premenná **j** zatiaľ nemá platnú hodnotu).

Zostáva doplniť výpis hodnôt v **i**-tom riadku histogramu (riadky 17 až 23). V riadku 20 budeme vypisovať (doplníme **printf**) premennú **j**. Nakoniec po vypísaní všetkých čísel v riadku musíme výpis riadku ukončiť znakom nového riadku (v riadku 23 doplníme "**\n**"). Znaky, ktoré budeme vypisovať sú cifry od 0 po 9, preto hranice cyklu pre **j** v riadku 17 sú od **j = 0** do **j < 10**.

Cyklus **while** v riadkoch 18 až 22 musí zaručiť vypísanie správneho počtu cifier **j**, ktoré zodpovedajú dvojciferným číslam **ij**, ktoré boli načítané na vstupe. Počet dvojciferných čísel **ij**, ktoré sme načítali, je uložený v dvojrozmernom poli **g**,

preto v riadku 18 doplníme `g[i][j] > 0`, a v riadku 21 po vypísaní jednej cifry túto hodnotu (`g[i][j]`) znížime o 1.

Výsledný doplnený program podľa uvedeného opisu:

```

1  #include <stdio.h>
2
3  int g[10][10], c[10];
4
5  int main()
6  {
7      int i, j;
8      while (scanf("%d", &i) > 0)
9      {
10         g[i/10][i%10]++;
11         c[i/10]++;
12     }
13     for (i = 0; i < 10; i++)
14         if (c[i])
15         {
16             printf("%d | ", i);
17             for (j = 0; j < 10; j++)
18                 while (g[i][j] > 0)
19                 {
20                     printf("%d", j);
21                     g[i][j]--;
22                 }
23             printf("\n");
24         }
25     return 0;
26 }
```

Riešenie doplnujúcej úlohy: Program nepotrebuje nutne dvojrozmerné pole. Stačí si v poli `g` pamätať počet načítaných čísel pre každú možnú hodnotu na vstupe (na vstupe sú celé čísla z intervalu od 0 do 99).

Definovali by sme teda jednorozmerné `int g[100]`, v riadku 10 doplníme `g[i]++`, v riadku 18 doplníme `g[10*i+j] > 0` a v riadku 21 `g[10*i+j]--`;

Úloha 4-34 (riešenie)

Analyzujeme chýbajúce miesta v programe a pokúsime sa doplniť najskôr tie miesta, pre ktoré jednoznačne vyplýva, čo do nich treba doplniť.

Vzhľadom na definíciu magického štvorca treba overiť či súčet v každom riadku, stĺpci a diagonále (sú dve) je rovnaký. Program, ktorý máme doplniť, obsahuje

opakujúce sa bloky podobného kódu, ktoré zodpovedajú jednotlivým smerom (riadkom, stĺpcom a diagonálam). Všimnime si malý rozdiel, že v riadku 7 cyklus začína o 1, teda ako keby sme 0-tý riadok (alebo stĺpec) neprešli. Ako je to možné? Jednoducho. Predtým ako budeme kontrolovať, či súčet čísel v riadkoch, stĺpcoch alebo diagonálach je rovnaký, musíme hodnotu súčtu určiť: napr. podľa 0-tého riadku. V riadku 5 preto doplníme **a[0][j]**, čím v cykle v riadkoch 4-5 určíme do premennej (**sucet**) súčet čísel, voči ktorému budeme ostatné súčty porovnávať (**sucet**).

Prehliadnutím zostávajúceho programu si všimneme, že súčty v riadkoch, stĺpcoch a diagonálach si v cykloch počítame do premennej **k**, ktorú by sme mali porovnať voči referenčnému súčtu (**sucet**). Ak je hodnota rôzna, tak štvorec nie je magický (**return 0** – doplníme v riadkoch 12, 20, 26 a 31), nakoniec v riadku 33 vieme, že je magický (**return 1**). V riadkoch 11, 19, 25 a 30 doplníme porovnanie zisteného súčtu s referenčným súčtom (**k != sucet**).

Zostáva doplniť chýbajúce miesta v riadkoch 10, 18, 24 a 29, v ktorých budeme postupne počítať súčty pre riadky, stĺpce, hlavnú a vedľajšiu diagonálu. V riadku 10 doplníme **a[i][j]**, v riadku 18 tiež **a[i][j]**, v riadku 24 doplníme **a[i][i]** a v riadku 29 doplníme **a[i][n-i-1]**.

Výsledná doplnená funkcia podľa uvedeného opisu:

```

1  int magicky_stvorec(int **a, int n)
2  {
3      int i, j, sucet = 0, k;
4      for (j = 0; j < n; j++) // sucet v riadku 0
5          sucet += a[0][j];
6
7      for (i = 1; i < n; i++) // riadky
8      {
9          for (k = 0, j = 0; j < n; j++)
10             k += a[i][j];
11             if (k != sucet)
12                 return 0;
13         }
14
15         for (j = 0; j < n; j++) // stlpce
16         {
17             for (k = 0, i = 0; i < n; i++)
18                 k += a[i][j];
19             if (k != sucet)
20                 return 0;
21         }

```

```
22
23   for (k = 0, i = 0; i < n; i++) // hlavna diagonala
24       k += a[i][i];
25   if (k != sucet)
26       return 0;
27
28   for (k = 0, i = 0; i < n; i++) // vedlajsia diagonala
29       k += a[i][n-i-1];
30   if (k != sucet)
31       return 0;
32
33   return 1;
34 }
```


Projekty

Zvyčajnou úlohou programátora je podľa zadania vytvoriť program riešiaci danú úlohu. Konečným cieľom programovania však nemôže byť len písať programy ale riešiť skutočné problémy ľudí. Práca programátora je len prvý krok na ceste stať sa efektívnym softvérovým inžinierom. Na tejto ceste je potrebné neustále hľadať nové lepšie riešenia, vyhľadávať konštruktívnu kritiku svojich riešení a, v neposlednom rade, obklopiť sa rovnako zmýšľajúcimi ľuďmi.

Programovanie nie je obmedzené na napísanie zdrojového kódu programu, ale postupne zahŕňa tieto aktivity: definovanie cieľov programu, návrh programu, implementáciu vytvorením zdrojového kódu, vytvorenie samotného (spustiteľného) programu, spustenie vytvoreného programu, testovanie a hľadanie chýb, a nakoniec aj úpravy a údržbu v prípade dodatočných požiadaviek.

V predchádzajúcich kapitolách sme sa v úlohách venovali prevažne častiam návrhu programu (čo, ako a prečo bude program robiť), implementácii tohto návrhu zdrojovým kódom a tiež sme skúmali správnosť vzniknutého programu.

Praktické zručnosti pri programovaní ďalej zahŕňajú overenie (testovanie) správnosti vytvoreného programu v rozličných situáciách a na rozličných vstupoch. V prípade objavenia nesprávnych výsledkov je potrebné nájsť chybu, pričom hľadanie chyby zvyčajne prebieha najskôr ako čítanie a analýza zdrojového kódu programu, a tiež aj ako analýza výpočtu nesprávneho výsledku v špecializovaných nástrojoch.

Každý programovací jazyk má svoje obmedzenia a programovací jazyk C je tým, že umožňuje priamu prácu so systémovými zdrojmi, obzvlášť náchylný na ťažko odhaliteľné chyby. Je veľmi náročné písať robustné programy v jazyku C, pretože ochranné mechanizmy, ktoré sú v novších programovacích jazykoch bežne dostupné, sa v jazyku C nenachádzajú: napr. jeden chybný prístup do poľa za hranice vyhradenej pamäte môže spôsobiť nepredvídateľné správanie programu, a na výsledok programu sa už nemôžeme spoľahnúť.

V tejto kapitole sa pokúsime na viacerých ukážkach opísať celý proces tvorby riešenia. Keď sme konfrontovaní so zložitejšími úlohami, môže sa stať, že nevieme kde a ako začať. Dôležité je komplikovaný problém vhodne rozdeliť (dekomponovať) na menšie podproblémy, ktorých vyriešením a spojením dielčích výsledkov dostaneme riešenie pôvodného komplikovaného problému.

V prvej časti si podrobne opíšeme celý proces tvorby riešenia na príklade štandardnej úlohy (maximum z čísel na vstupe). V ďalších častiach sa venujeme projektom väčšieho rozsahu, ktoré prepájajú znalosti z viacerých oblastí programovania. Výsledné zdrojové kódy riešení sa skladajú z viacerých funkcií a vhodne využívajú viaceré prvky programovacieho jazyka.

Naším cieľom je zrozumiteľne a prístupne pre začínajúceho programátora opísať ako postupovať pri riešení zložitejších úloh.

A. Maximum

Proces tvorby riešenia zahŕňa rôzne nepredvídané situácie a rozhodnutia, ktoré musí programátor spraviť, aby dospel k nejakému uspokojivému riešeniu.

Pokúsime sa na príklade pomerne jednoduchej úlohy spolu preskúmať a objaviť možnosti, ktoré sa otvárajú, pri riešení aj takejto zdanlivo jednoduchej úlohy.

Zadanie: Na vstupe je postupnosť reálnych čísel, ktoré chceme analyzovať.

Prvé číslo vstupu N označuje počet čísel postupnosti, potom nasleduje $N > 1$ reálnych čísel. Chceme nájsť maximum tejto postupnosti.

Napište program, ktorý načíta čísla zo vstupu a na výstup vypíše maximum z načítaných čísel vo formáte podľa ukážky nižšie. Výsledok vypíše zaokrúhlený na dve desatinné miesta.

Ukážka vstupu:

4
15.451
5.5
4.78
6

Výstup pre ukážkový vstup:

Maximum: 15.45

Riešenie: Pri návrhu riešenia (akejkoľvek úlohy) je dôležité si uvedomiť, že našou úlohou je vytvoriť program, ktorý správne vyrieši nielen úlohu pre ukážkové údaje zo zadania, ale pre akékoľvek vstupné údaje spĺňajúce špecifikáciu úlohy. Súčasťou tejto špecifikácie je najmä **formát vstupu a výstupu**.

Formát vstupu jednoznačne opisuje, ako porozumieť údajom na vstupe, aby sme ich mohli načítať a spracovať. Skutočná úloha je teda vytvoriť program, ktorý očakáva (a spracuje) údaje dodržiavajúce tento formát. Ak údaje na vstupe nespĺňajú špecifikáciu úlohy a formát vstupu, ako je uvedené v zadaní úlohy, tak nie je chybou programu, ak na takýchto vstupných údajoch vypíše nesprávny výsledok.

Vráťme sa späť k našej úlohe. Text tejto úlohy opisuje zadanie pre náš program ako zistenie maxima spomedzi prvkov zadanej postupnosti. Formát vstupu nie je bližšie opísaný, keďže text jednoznačne (formálne) neuvádza, akým spôsobom sú jednotlivé čísla zadané: môžu byť v desiatkovej sústave alebo v binárnej sústave alebo akokoľvek inak zadané, napr. aj slovne. Presná špecifikácia často obsahuje podrobné opisy, ktoré začínajúceho programátora môžu odradiť. V praxi však často ani nie sú potrebné, pretože skúsení programátori už vedia, čo môžu očakávať. Ako alternatívu presnej špecifikácie formátu vstupu a výstupu vždy uvádzame v úlohách ukážku vstupu/výstupu, ktorá pre skúseného programátora úplne a jednoznačne vysvetľuje podstatu vstupného aj výstupného formátu.

Otázka: Čo keď sú na vstupe dáta, ktoré nie sú v požadovanom formáte?

Predstavme si, že programátor Jakub dostal v práci za úlohu vytvoriť program riešiaci vyššie zadanú úlohu. Jakub program vytvoril a odovzdal nadriadenému, ktorý pri preberaní chcel overiť jeho funkčnosť tým, že zadal na vstup tieto údaje (čísla sú rovnaké ako v ukážke v zadaní):

Štyri

Pätnásť celých štyristopäťdesiatjeden tisícín

Päť celých päť desatín

Štyri celé sedemdesiatosem stotín

Šesť

Po spustení na tomto vstupe, Jakubov program ale nepracoval správne, pretože program očakával na vstupe údaje vo formáte, ktorý bol uvedený v ukážke v zadaní:

čísla boli zadané v desiatkovej sústave zaužívaným spôsobom. Nie je možné, aby Jakub predvídal všetky možné iné spôsoby zadania čísel, navyše ak ich zadanie úlohy nespomínalo. Chyba nie je vo vytvorenom Jakubovom programe, ale vo formáte vstupu, ktorý zadal jeho nadriadený.

Dodržiavanie formátu údajov na vstupe je základná požiadavka, ktorá platí pri každej úlohe, ktorú budeme riešiť. Nezabúdajme však pritom, že v praxi sú údaje na vstupe často nesprávne zadané, najmä ak vstup voľne zadával používateľ (napr. návštevník webovej stránky alebo hacker usilujúci sa spôsobiť chybu). Predtým, ako sa vstup dostane do nášho program, by mala vždy prebehnúť kontrola formátu vstupu a v prípade chyby je potrebné upozorniť používateľa a žiadať nápravu.

Formálne môžeme kontrolu formátu považovať za samostatnú úlohu, pri ktorej vstupné údaje sú ľubovoľné a výstupom je určenie (štruktúrovanej) hodnoty zo vstupu. Dôslednej kontrole formátu vstupu sa v úlohách alebo ich riešeníach explicitne nevenujeme preto, aby sme sa sústredili na podstatu dôležitých princípov programovania. Pri každej riešenej úlohe predpokladáme, že na vstupe program dostane údaje presne zodpovedajúce formátu vstupu opísaného v zadaní.

Otázka: Čo keď program vypíše výsledok “trochu inak”?

Nesprávne formátovanie výstupu je veľmi častá chyba. Uvažujme napr. že by náš program síce správne určil hodnotu pre maximum, ale na výstup by ju vypísal takto:

```
Maximum je 15.45
```

Aj keď princíp výpočtu v takomto programe je zjavne správny (lebo výsledná hodnota je správna) a človek by takýto výstup správne interpretoval, tak výstup nespĺňa formát výstupu požadovaný v zadaní (v uvedenej ukážke). Aj keď to napohľad vyzerá ako maličkosť (zopár znakov navyše), je to vážny nedostatok, lebo výstup nášho programu môže byť určený pre ďalšie spracovanie ako vstup pre iný program, ktorý ale na vstupe očakáva dohodnutý formát. Aj malý rozdiel vo formáte výstupu nášho riešenia môže ľahko spôsobiť, že následujúci výpočet ďalšieho programu nemusí pracovať správne.

Zamyslime sa nad vytvorením zdrojového kódu programu (riešenia). Bez ohľadu na konkrétny spôsob riešenia, musí zdrojový kód programu postupne obsahovať tieto štyri časti:

1. Načítanie vstupu (podľa formátu vstupu zo zadania)
2. Reprezentácia výsledku
3. Výpočet výsledku
4. Výpis výsledku na výstup (podľa formátu výstupu zo zadania)

Otázka: Ktorú z týchto častí budeme riešiť ako prvú?

Odpoveď: Ak bez ďalšieho premýšľania začneme načítavať vstup (1), robíme zbytočnú prácu. V tomto okamihu totiž zatiaľ nevieme, čo s načítavanými hodnotami na vstupe vykonávať, aby sme vhodne (s čo najmenšou námahou) úlohu vyriešili. Je totiž možné, že pre vyriešenie úlohy je vstup nutné načítať celý do pamäte a potom ho vo viacerých prechodoch spracúvať. Naopak, možno postačí, ak budeme vstup priebežne spracúvať bez nutnosti uloženia v pamäti. Pri tvorbe nášho riešenia nemá zmysel sa v tomto okamihu zamýšľať nad detailmi načítania vstupu, keďže sme zatiaľ nepremysleli ďalší postup výpočtu výsledku. Z pohľadu vstupu je pre nás dôležitá najmä podstata údajov na vstupe: že sú to reálne čísla a nie konkrétny formát vstupu.

Výpočet (3) aj výpis výsledku (4) nie je možné realizovať bez uvedenia konkrétnej reprezentácie výsledku (2). Najskôr sa teda musíme zamyslieť a rozhodnúť, ako budeme výsledok v programe reprezentovať: úlohou je zistiť maximum spomedzi prvkov zadanej postupnosti reálnych čísel. Výsledok (maximum) je preto najlepšie reprezentovať desatinným číslom s dvojitou presnosťou (typ **double**) – premennou **max**.

Otázka: Ako zistím, či zvolená reprezentácia výsledku je vhodná?

To sa nedá s určitosťou vopred povedať. Nevyhnutné podmienky sú: aby sme využitím zvolenej reprezentácie vedeli výsledok vypísať v požadovanom formáte (4), a aby sme dokázali výsledok vo zvolenej reprezentácii aj vypočítať (3). Inú možnú reprezentáciu výsledku v tejto úlohe preskúmame v cvičeniach.

Výpis výsledku (4) v tejto reprezentácii podľa požiadaviek zadania je potom priamočiare použitie funkcie **printf()**, detaily rozoberáme nižšie v texte.

Zamerajme sa teraz na **priebeh výpočtu výsledku (2)**: Pri spustení programu je hodnota premennej **max** nedefinovaná, ale po skončení výpočtu už potrebujeme, aby obsahovala správnu výslednú hodnotu – maximum z čísel na vstupe. V programe teda potrebujeme správne vypočítať hodnotu **max**. Je potrebné načítať úplne všetky čísla zo vstupu? Samozrejme, že áno. Ak by sme totiž niektoré z čísel (napr. posledné) nenačítali, mohlo by práve to nenačítané byť väčšie ako všetky ostatné a zistená hodnota pre maximum by bola nesprávna, lebo by nezohľadňovala číslo, ktoré sme nenačítali. V tejto úlohe preto čísla načítame postupne všetky. Každé načítané číslo (označme ho **d**) môže upraviť hodnotu maxima (**max**). Ak po každom načítanom čísle budeme v premennej **max** udržiavať aktuálnu hodnotu maxima z doteraz načítaných čísel, tak po načítaní všetkých čísel bude premenná **max** obsahovať hodnotu maxima spomedzi všetkých čísel na vstupe. Úpravu hodnoty maxima (**max**) po načítaní čísla **d** vykonáme takto:

```
if (max < d)
    max = d;
```

ak je hodnota priebežného maxima (**max**) menšia ako hodnota **d**, znamená to, že práve načítané číslo je nové maximum všetkých doteraz načítaných čísel.

Zostáva doriešiť načítanie čísel zo vstupu (1). Načítať údaj zo vstupu znamená preniesť hodnotu zo vstupu do pamäte programu, pričom načítavanie vstupu sa posunie ďalej na nasledujúci údaj – teda opätovným načítaním sa načíta ďalšia hodnota na vstupe a nie hodnota, ktorá už bola načítaná.

Ako už bolo spomenuté, k správne načítaniu je potrebné poznať formát vstupných údajov – dátové typy a počet. V našom prípade sa na vstupe podľa zadania najskôr nachádza číslo **N** ($N > 1$) počet čísel postupnosti. Zrejme teda **N** je prirodzené číslo (typ **int**, resp. **unsigned int**), pretože počet prvkov nemôže byť záporné ani desatinné číslo. Nasledujúcich **N** čísel na vstupe sú reálne, ktoré môžeme reprezentovať ako desatinné čísla s dvojitou presnosťou (typ **double**).

Ak načítavame jeden prvok, tak funkcia **scanf()** očakáva dva argumenty: formát údajov (čo načítavame) a adresu do pamäte (kam načítať). Funkcia **scanf()** po vykonaní do pamäte programu naplní hodnotu prečítanú zo vstupu podľa požadovaného formátu. Napr. nasledujúcim volaním: **scanf("%d", &n);** načítame zo vstupu prirodzené číslo (formátovací symbol **%d**) a uložíme jeho hodnotu do pamäte, ktorá je vyhradená pre premennú **n**. Týmto sme v programe úspešne načítali prirodzené číslo **N** – počet čísel postupnosti.

Potom v cykle postupne načítame **N** desatinných čísel a po každom aktualizujeme hodnotu priebežne zistenej hodnoty maxima.

Prvý úplný zdrojový kód riešenia je takýto:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i;
6      double d, max;
7      scanf("%d", &n);
8      for (i = 1; i <= n; i++)
9      {
10         scanf("%lf", &d);
11         if (max < d)
12             max = d;
13     }
14     printf("Maximum: %.2lf\n", max);
15     return 0;
16 }
```

Telo cyklu (riadky 9 až 13) sa budú opakovať postupne pre hodnoty **i = 1, 2, ... n**. Formát výpisu **%.2lf** vo funkcii **printf()** určuje výpis hodnoty **max** ako reálne číslo s dvoma desatinnými miestami. Po výpise určeného textu a čísla nasleduje znak presunu na nový riadok (**\n**).

Uvedený zdrojový kód obsahuje všetky potrebné časti riešenia v rámci hlavnej funkcie **main()**, v ktorej začína vykonávanie programu. Po spustení programu sa postupne vykonávajú príkazy, ktoré tvoria telo funkcie **main()**. Výstupom funkcie **main()** je celé číslo (typ **int**), ktoré signalizuje operačnému systému výskyt chyby pri vykonávaní programu (hodnota 0 signalizuje bezchybný priebeh vykonávania).

Úloha na precvičenie: Je toto riešenie správne? Pokúste sa nájsť dôvody prečo by malo byť správne, a tiež sa pokúste nájsť dôvody, prečo je nesprávne. Riešenie si naprogramujte a vyskúšajte, či pracuje správne!

Otázka: Po spustení program vypísal správny výsledok. Je teda program správny?

Výpis správneho výsledku pre niekoľko rôznych vstupov ešte nemusí znamenať, že program by pracoval správne pre akýkoľvek vstup. Pripomeňme, že program má zmysel spúšťať len na vstupoch, ktoré spĺňajú špecifikáciu zadania a dodržiavajú formát vstupu. V našom prípade musí vstup obsahovať najskôr jedno prirodzené číslo N , ktoré musí mať hodnotu aspoň 2 a musí nasledovať aspoň N reálnych čísel. Všetky čísla na vstupe musia byť oddelené bielymi znakmi (napr. medzerami, riadkami) v zmysle ukážky vstupu a výstupu v zadaní.

Správnosť programu nezávisí len od preskúšania výpočtu na rozličných vstupoch. Dokonca aj vykonanie programu na rôznych počítačoch alebo operačných systémoch môže spôsobiť rozdiel pri vykonaní niektorých príkazov a program sa môže dopracovať k rôznym (možno nesprávnym) výsledkom.

Pri vytváraní programu je preto potrebné uvažovať predovšetkým nad vhodnosťou použitých dátových typov a operácií s nimi, aby sa zabezpečila prenositeľnosť programu medzi rôznymi prostrediami.

Pre zodpovedanie otázky preskúmame použitie premenných v jednotlivých riadkoch zdrojového kódu. V každom riadku používame nejaké premenné, pričom niekedy hodnotu premennej len z pamäte prečítame a niekedy do nej aj zapisujeme. Napr. pri úprave maxima (**max**):

11	if (max < d)
12	max = d;

v podmienke (**if**) prečítame aktuálne hodnoty premenných **max** a **d**, a ak hodnota **max** je väčšia ako hodnota **d**, tak pokračujeme na nasledujúci riadok, v ktorom najskôr (znovu) prečítame aktuálnu hodnotu premennej **d** a túto hodnotu potom zapíšeme

do premennej **max** (v tomto riadku už predchádzajúcu hodnotu premennej **max** program zbytočne nenačítava).

Výpočet programu prebieha postupne po jednotlivých príkazoch v zmysle tzv. toku riadenia a je nevyhnutné, aby hodnota premennej bola jednoznačne daná v okamihu, keď z nej potrebujeme hodnotu prečítať. Zistíme teda, na ktorých riadkoch programu načítavame hodnotu premennej **max**? Na riadkoch 11 (pri porovnaní) a 14 (pri výpise). Hodnotu do premennej však priradujeme až na riadku 12, akú hodnotu má premenná **max** pred priradením v riadku 12? Štandard ANSI C hovorí, že lokálna premenná má neurčenú počiatočnú hodnotu, a preto ak hodnotu použijeme predtým ako do premennej priradíme, program sa môže správať nedefinované – na výsledok programu sa nemôžeme spoľahnúť.

Premennej **max** teda potrebujeme priradiť počiatočnú hodnotu – potrebujeme ju inicializovať. Najlepšie vždy je premenné inicializovať hneď pri zadeinovaní, teda v našom prípade napr.

6	<code>double d, max=0.0;</code>
---	---------------------------------

Ako bude pracovať náš program, ak premennú **max** inicializujeme na hodnotu 0? Na ukážkovom vstupe je výsledok správny a možno aj na zopár ďalších vstupoch, ktoré vyskúšate, bude výsledok správny. Je teda program už správny? Aby sme to spoľahlivo povedali, tak sa skúsme zamyslieť, kedy by výsledok mohol byť nesprávny? Výsledok programu (vypísanie hodnoty premennej **max**) by mohol byť nesprávny ak by sme nevypísali prvok z postupnosti (teda počiatočná hodnota **max** by nikdy nebola prepísaná prvkom postupnosti), alebo ak by sme vypísali nejaký nesprávny prvok postupnosti (existoval by ešte väčší prvok).

Nesprávny prvok vypísať nemôžeme, pretože načítame a spracujeme každý z prvkov a ak niektorý je väčší ako hodnota **max**, tak hodnotu **max** upravíme.

Čo sa však stane v situácii, v ktorej všetky prvky postupnosti budú menšie ako počiatočná hodnota **max**? V našom prípade to je vtedy, keď by na vstupe boli len záporné čísla: potom by sme vypísali 0 napriek tomu, že 0 nie je najväčšie číslo postupnosti.

Akú zvoliť počiatočnú hodnotu premennej **max**, aby sme sa tejto chybe vyhli? Začínajúci programátori často volia nejaké veľmi malé číslo, napr. **-99999999**. Bez ohľadu na zvolenú konštantu sa vždy bude dať pripraviť vstup, pre ktorý program vypíše nesprávny výsledok. Vstup pripravíme tak, že všetky čísla na vstupe budú menšie ako konštanta uvedená v programe.

Musíme preto počiatočnú hodnotu **max** zvoliť tak, aby zodpovedala nejakému číslu zo vstupnej postupnosti. Vhodné je hneď prvé načítané číslo priradiť do **max**, a až druhé a ďalšie čísla porovnávať s hodnotou **max**. Podmienku, v ktorej potrebujeme prečítať hodnotu **max** (riadok 11), preto upravíme tak, aby sme v prípade prvého načítaného čísla hodnotu **max** nenačítavali (lebo je neinicializovaná) a len priradíme hodnotu čísla do premennej **max**.

Výsledný správny program:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i;
6      double d, max;
7      scanf("%d", &n);
8      for (i = 1; i <= n; i++)
9      {
10         scanf("%lf", &d);
11         if (i == 1 || max < d)
12             max = d;
13     }
14     printf("Maximum: %.2lf\n", max);
15     return 0;
16 }
```

V riadku 11 je príklad tzv. skráteného vyhodnotenia logického výrazu, ktoré slúži na zrýchlenie vykonávania programu: v prípade ak aktuálna hodnota premennej **i** je 1, tak sa pokračuje na riadok 12 bez toho, aby sa vyhodnocovala podmienka **max < d**.

Úlohy na zamyslenie

1. Uvažujme, že presunieme definíciu lokálnych premenných z funkcie **main()** ešte pred funkciu na riadok 2. Premenné sa tak stanú globálne, ktoré sú automaticky inicializované. Zjednodušilo by to výsledný program?

2. Ako upraviť inicializáciu premennej **max** tak, aby sme mohli zachovať pôvodnú podmienku **if (max < d)** na porovnanie (v riadku 11)?
3. Možno program zjednodušiť tak, že nepoužijeme riadiacu premennú cyklu **i**?
4. Ako by sa zmenil program, keby sme použili inú reprezentáciu výsledku: riešenie (maximum čísel zo vstupu) by sme reprezentovali ako index (poradie) čísla vo vstupnej postupnosti?
5. Ako by sa zmenil program, ak by počet čísel postupnosti nebol na vstupe vopred daný, ale chceli by sme spracovať všetky čísla až do konca vstupu?
6. Ako by sa zmenil program, ak by sme chceli určiť aj minimum?

Riešenie úloh na zamyslenie

1. Globálne a statické premenné sú inicializované na hodnotu 0, ale premennú **max** (priebežnú hodnotu maxima) treba inicializovať na prvé číslo postupnosti, nie hodnotu 0. Navrhovaná úprava by teda vôbec nepomohla.
2. Ak chceme zachovať podmienku **if (max > d)**, musíme prvé priradenie do **max** vykonať ešte pred cyklom. Kód upravíme takto:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i;
6      double d, max;
7      scanf("%d %lf", &n, &max);
8      for (i = 2; i <= n; i++)
9      {
10         scanf("%lf", &d);
11         if (max < d)
12             max = d;
13     }
14     printf("Maximum: %.2lf\n", max);
15     return 0;
16 }
```

Prvé desatinné číslo môžeme načítať spolu s číslom **N** (riadok 7) a premennú **i** inicializujeme (riadok 8) na hodnotu 2, čím zabezpečíme, že spracujeme len zostávajúce čísla a program bude funkčný pre všetky korektne zadané vstupy.

3. Nadviažeme na riešenie predchádzajúcej úlohy (2) a namiesto cyklu **for** v riadku 8 použijeme nasledujúci cyklus: **while (--n)**
4. Postupnosť čísel na vstupe by sme si museli celú zapamätať v pamäti do poľa. Priebežne by sme si určovali index maximálneho čísla a pri výpise výsledku by sme vypísali hodnotu čísla na indexe **max**. Upravený program uvádzame na konci.

Všimnime si, že okrem vytvorenia poľa potrebnej veľkosti sme museli upraviť hranice cyklov, pretože polia sú v jazyku C indexované od 0 do N-1.

Reprezentácia výsledku v tomto cvičení je pre túto úlohu nevýhodná, pretože výsledný program potrebuje pre výpočet výsledku významne viac pamäti. Riešenie bez načítania vstupnej postupnosti do pamäte je teda výrazne efektívnejšie.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int n, i, max;
7      double *cislo;
8      scanf("%d", &n);
9      cislo = (double*)malloc(n * sizeof(double));
10     for (i = 0; i <= n-1; i++)
11         scanf("%lf", &cislo[i]);
12     for (max = 0, i = 1; i <= n-1; i++)
13     {
14         if (cislo[max] < cislo[i])
15             max = i;
16     }
17     printf("Maximum: %.2lf\n", cislo[max]);
18     free(cislo);
19     return 0;
20 }
```

5. Využili by sme návratovú hodnotu funkcie **scanf()**.

Každá funkcia, ktorá načítava vstup, musí v návratovej hodnote oznámiť, či načítanie prebehlo úspešne. Funkcia **scanf()** vracia počet úspešne načítaných prvkov. Pridali by sme cyklus **while**, v ktorom by sme pomocou funkcie **scanf()** načítavali po jednom čísle pokým by vracala hodnotu 1. Keď volanie funkcie **scanf()** vráti inú hodnotu ako 1, znamená to, že na vstupe už nie je číslo a cyklus

môžeme ukončiť. Výsledný program je potom prehľadnejší, pretože zmizla réžia pre načítanie presného počtu (N) čísel zo vstupu:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      double d, max;
6      scanf("%lf", &max);
7      while(scanf("%lf", &d) == 1)
8      {
9          if (max < d)
10             max = d;
11     }
12     printf("Maximum: %.2lf\n", max);
13     return 0;
14 }
```

6. Výpočet minima je priamočiare rozšírenie pôvodného programu. Pridali by sme premennú **min** a jej hodnotu by sme priebežne udržiavali podobne ako **max**:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n, i;
6      double d, max, min;
7      scanf("%d %lf", &n, &max);
8      min = max;
9      for (i = 2; i <= n; i++)
10     {
11         scanf("%lf", &d);
12         if (min > d)
13             min = d;
14         if (max < d)
15             max = d;
16     }
17     printf("Minimum: %.2lf\n", min);
18     printf("Maximum: %.2lf\n", max);
19     return 0;
20 }
```

B. Koleso šťastia

Populárna americká televízna hra Wheel of Fortune na Slovensku známa pod názvom Koleso Šťastia je jednoduchá hra, pri ktorej hráči hľadajú neznáme slovo alebo text skúšaním jednotlivých písmen. Vašou úlohou je vyhodnotiť variant tejto hry pre jedného hráča. Hra začína zobrazením skrytého slova alebo textu tak, aby hráč vedel koľko písmen v slove alebo texte sa snaží uhádnuť. Potom hráč opakovane najskôr vytočí na kolese (šťastia) bodovú hodnotu a zvolí písmeno. Hráč získa za každé nájdené písmenko príslušný počet bodov, ktoré vytočil na kolese.

Hráči do televíznej šou chodia opakovane, úlohou je určiť mená top troch hráčov a ich bodové zisky – teda tých troch hráčov, ktorí získali čo najviac bodov spomedzi všetkých hráčov. Môžete predpokladať, že hru hralo najviac 100 hráčov a že hráči získali rôzny počet bodov.

V televíznej šou sa odohralo viacero hier, údaje o konkrétnej hre sú v súbore **hra_cislo.txt**, teda napr. **hra_1.txt** pre prvú hru, **hra_2.txt** pre druhú hru atď. Počet hier nie je vopred zadáný, súbory načítavajte až kým existujú. V súbore pre jednu hru je na prvom riadku najskôr uvedené meno hráča, na druhom riadku hľadaný (neznámy) text, a na každom ďalšom riadku až do konca súboru je počet bodov (celé číslo) a hádané písmenko (znak malej anglickej abecedy).

Ukážka vstupných súborov:

hra_1.txt	hra_2.txt
Victor Hugo bicykel nie je auto 400 e 200 a 50 i 200 e	Ernest Hemingway komu zvoní do hrobu 50 e 300 o 180 i

Vysvetlenie hry v súbore hra_1.txt: Hráč Victor Hugo hádal text bicykel nie je auto a hra prebiehala takto: hráč vytočil 400 a hádal písmeno **e**, našiel 3-krát, potom vytočil 200 a skúšal písmeno **a**, našiel raz, potom vytočil 50 a skúšal písmeno **i**, našiel dvakrát, nakoniec vytočil 200 ale pri hádaní sa splietol a opäť hádal písmeno **e**, začo nezískal

body. Súťaž sa potom z časových dôvodov skončila, nie je potrebné, aby hráč uhádol celý text. Hráč z tejto hry celkovo získal $400 \cdot 3 + 200 \cdot 1 + 50 \cdot 2 = 1500$ bodov.

Napíšte program, ktorý spracuje údaje zo záznamov hier v súboroch a na štandardný výstup vypíše mená top troch hráčov, ktorí získali najviac bodov spomedzi všetkých hráčov, ktorí sa hry zúčastnili.

Riešenie

Budeme postupovať podľa schémy v štyroch krokoch (z projektu A): navrhujeme spôsob reprezentácie výsledku a potom premyslíme jeho výpočet a výpis. Načítanie vstupu je v tomto prípade významnou časťou výpočtu.

Reprezentácia výsledku: Výsledok je meno hráča (resp. troch hráčov), ktorý získal najviac bodov. Počas vykonávania programu si preto potrebujeme pre každého hráča pamätať jeho meno a počet získaných bodov, aby sme potom na konci vedeli určiť, ktorí traja majú najviac bodov a vypísať ich.

Údaje o hráčovi združíme do štruktúry (**struct Hrac**), pričom údaje pre všetkých hráčov budeme pre jednoduchosť a dostupnosť v rôznych funkciách evidovať v globálnej premennej (**h**), počet hráčov bude **nh** (inicializovaný na 0).

```
1 struct Hrac
2 {
3     char *meno;
4     int body;
5 };
6
7 struct Hrac h[100];
8 int nh;           // pocet hracov
```

Výpis výsledku: Pri výpise výsledku musíme na základe údajov o všetkých hráčoch vypísať mená troch hráčov s najvyšším bodovým ziskom. Jednoduchšie ako spraviť funkciu na výpis troch top hráčov je trikrát zavolať funkciu na výpis jedného top hráča s tým, že hráča, ktorého sme už raz vybrali, znovu nevyberieme.

Napíšeme funkciu **najdi_vitaza()**, ktorá vráti index hráča s najvyšším bodovým ziskom s tým, že ju budeme volať opakovane, aby nám vrátila aj druhého a tretieho hráča. Niekde (v nejakej premennej) si musíme evidovať, ktorých hráčov

sme už určili ako top a pri ďalších volaniach funkcie ich už nebudeme uvažovať, aby nám ďalšie volanie funkcie vrátilo index hráča s najvyšším bodovým spomedzi zostávajúcich hráčov. Nato využijeme pole príznakov **used[]**, v ktorom si budeme pamätať, či sme už hráča z funkcie niekedy vrátili. Funkcia využijúc premennú **used[]** prejde globálne pole hráčov (**h[]**) a nájde takého, ktorý získal čo najviac bodov, a zároveň v poli **used[]** má zatiaľ poznačenú hodnotu 0. Pole **used[]** definujeme ako statické, aby jeho rozsah platnosti zostal obmedzený na túto funkciu.

```

9  int najdi_vitaza()
10 {
11     static char used[100];
12     int i, maxi = -1;
13     for (i = 0; i < nh; i++)
14         if (used[i] == 0)
15         {
16             if (maxi < 0 || h[maxi].body < h[i].body)
17                 maxi = i;
18         }
19     if (maxi >= 0)
20         used[maxi] = 1;
21     return maxi;
22 }
```

Načítanie vstupu a výpočet: Vstupné údaje sa nachádzajú vo viacerých súboroch, pričom súbory sú vzájomne nezávislé a môžeme ich spracovať jednotlivo.

Načítanie a spracovanie údajov v jednom súbore preto implementujeme do samostatnej funkcie **nacitaj_hru()**, ktorú budeme volať pre každý súbor. Spracovanie je riadené z hlavnej funkcie **main()**, ktorá bude otvárať súbory s názvami **hra_i.txt** pre **i** idúce od 1 do nekonečna, pričom pre každý úspešne otvorený súbor zavolá funkciu **nacitaj_hru()**, ktorá údaje načíta a spracuje do záznamov o hráčoch (**h[]**). Nakoniec vypíše troch top hráčov, resp. menej hráčov, ak ich máme v záznamoch menej.

Implementácia hlavnej funkcie **main()** je priamočiary prepis týchto myšlienok do zdrojového kódu:


```
62 int main(void)
63 {
64     FILE *f;
65     char buf[100];
66     int i;
67     for (i = 1; ; i++)
68     {
69         sprintf(buf, "hra_%d.txt", i);
70         if (!(f = fopen(buf, "rt")))
71             break;
72         nacitaj_hru(f);
73     }
74
75     for (i = 0; i < 3 && i < nh; i++)
76         printf("%s\n", h[najdi_vitaza()].meno);
77     return 0;
78 }
```

Zostáva nám napísať funkciu **nacitaj_hru()** pre spracovanie jednej hry zo súboru. Funkcia bude súbor načítavať podľa špecifikácie formátu v zadaní úlohy. Najskôr načítame a spracujeme meno hráča. V prípade, že už máme hráča s rovnakým menom v globálnom poli hráčov (**h[]**) zaevidovaného, nájdeme si jeho index. Inak, nového hráča zaradíme na koniec poľa **h[]** a zvýšim počet evidovaných hráčov (**nh**). V oboch prípadoch si konkrétneho hráča v evidencii reprezentujeme ako index do globálneho poľa hráčov (**h[]**). Určenie indexu hráča v globálnom poli podľa jeho mena implementujeme do pomocnej funkcie **index_hraca()** takto:

```
23 int index_hraca(char *meno)
24 {
25     int i;
26     for (i = 0; i < nh; i++)
27         if (!strcmp(h[i].meno, meno))
28             break;
29     if (i == nh)
30     {
31         h[nh].meno = strdup(meno);
32         h[nh].body = 0;
33         nh++;
34     }
35     return i;
36 }
```

Funkcia **nacitaj_hru()** pokračuje načítaním textu a potom až do konca súboru načítavame a spracúvame dvojice body a tipované písmeno. Pre vyhodnotenie

bodového zisku pre jedno písmeno by sme potrebovali určiť, koľkokrát sa písmeno vyskytuje v hádanom texte. Vytvoríme si preto pomocnú funkciu **pocet_pismen()**, ktorá pre zvolený text určí počet výskytov zvoleného znaku:

```
37 int pocet_pismen(char *text, char znak)
38 {
39     int i, pocet = 0;
40     for (i = 0; text[i]; i++)
41         if (text[i] == znak)
42             pocet++;
43     return pocet;
44 }
```

Teraz by už implementácia funkcie **nacitaj_hru()** mala byť jednoduchá:

1. Načítame meno hráča a hádaný text.
2. Pomocou funkcie **index_hraca()** určíme index hráča v globálnom poli hráčov (**h[]**).
3. Načítame a spracujeme dvojice čísel (body a písmeno) postupne po jednom do konca vstupu tak, že pomocou funkcie **pocet_pismen()** určíme počet výskytov písmena v texte a pripočítame získané body hráčovi na hráčke konto. Body môžeme pripočítat len v prípade, ak písmeno ešte nebolo doteraz v tejto hre (súbore) tipované. Použitie-nepoužitie konkrétnych písmen si budeme evidovať v poli príznakov **pouzite[]**.

```
45 void nacitaj_hru(FILE *f)
46 {
47     char c, meno[100], text[100], pouzite[26];
48     int i, body;
49     for (i = 0; i < 26; i++)
50         pouzite[i] = 0;
51
52     fscanf(f, "%[^\n]\\n%[^\n]", meno, text);
53     i = index_hraca(meno);
54     while (fscanf(f, "%d %c", &body, &c) == 2)
55         if (pouzite[c-'a'] == 0)
56         {
57             h[i].body += body * pocet_pismen(text, c);
58             pouzite[c-'a'] = 1;
59         }
60 }
```

Pole príznakov **pouzite[]** by sme mohli vylepšiť tak, že namiesto poľa dĺžky 26 bajtov použijeme jednu 32-bitovú premennú (typ **int**), v ktorej si budeme evidovať použitie písmen v bitoch: **i**-tý bit bude nastavený na 1 ak sme už použili **i**-te písmeno.

Zjednodušený kód funkcie potom vyzerá takto:

```
45 void nacistaj_hru(FILE *f)
46 {
47     char c, meno[100], text[100];
48     int i, body, pouzite=0;
49
50     fscanf(f, "%[^\\n]\\n%[^\\n]", meno, text);
51     i = index_hraca(meno);
52     while (fscanf(f, "%d %c", &body, &c) == 2)
53         if (pouzite & (1<<(c-'a')) == 0)
54         {
55             h[i].body += body * pocet_pismen(text, c);
56             pouzite |= (1<<(c-'a'));
57         }
58 }
```

C. Kameň, papier, nožnice

Hra Kameň-papier-nožnice je hra pre dvoch hráčov, ktorá pôvodne pochádza z východnej Ázie. Jeden zápas sa hrá na niekoľko kôl. V každom kole každý hráč vopred zvolí jeden z troch možných ťahov: kameň, papier alebo nožnice, bez znalosti ťahu, ktorý zvolil súper. Podľa zaužívaných pravidiel, kameň vyhráva nad nožnicami, papier nad kameňom a nožnice nad papierom, v ostatných prípadoch nevyhrá nikto.

Napíšte program, ktorý vyhodnotí víťaza turnaja v hre Kameň-papier-nožnice. Na turnaji sa odohralo niekoľko zápasov, očíslované celými číslami 1, 2, ... Údaje o konkrétnom zápase sú v súbore **zapas_cislo.txt**, teda napr. **zapas_1.txt** pre prvý zápas, **zapas_2.txt** pre druhý zápas atď. Počet zápasov nie je vopred zadáný, súbory načítavajte až kým súbory existujú.

V súbore pre jeden zápas sú najskôr uvedené mená hráčov, ktorí zápas hrajú – meno prvého hráča v prvom riadku, meno druhého v druhom riadku. Na všetkých zvyšných riadkoch sú uvedené údaje o odohranom kole – vždy dve slová (bez diakritiky, oddelené medzerou) na jednom riadku – aký ťah hráči zvolili v príslušnom kole – napr. **kamen noznice** znamená, že prvý hráč zvolil kameň a druhý zvolil nožnice, teda prvý hráč vyhral. Povolené ťahy sú reťazce: **kamen, papier, noznice**. Zápas vyhrá ten hráč, ktorý vyhral najviac kôl, ak taký nie je (obaja vyhrali rovnaký počet kôl), nevyhral nikto z nich.

Víťazom turnaja sa stáva hráč, ktorý vyhral najviac zápasov, ak sú takí viacerí, vyhráva ten z nich alebo tí z nich, ktorí vyhrali najviac kôl.

Ukážka vstupných súborov (víťaz turnaja je Ernest Hemingway):

zapas_1.txt	zapas_2.txt	zapas_3.txt
Victor Hugo Ernest Hemingway kamen papier kamen kamen papier noznice	Ernest Hemingway Karl May papier kamen noznice kamen papier papier	Victor Hugo Karl May kamen kamen noznice kamen noznice papier
0:2	1:1	1:1

Ukážka vstupných súborov (víťazi sú dvaja: Ernest Hemingway a Karl May, obaja vyhrali 1 zápas a obaja vyhrali v turnaji 3 kolá):

zapas_1.txt	zapas_2.txt	zapas_3.txt
Victor Hugo Ernest Hemingway kamen papier kamen kamen papier noznice	Ernest Hemingway Karl May papier kamen noznice kamen papier papier papier noznice	Victor Hugo Karl May kamen kamen noznice kamen noznice papier
0:2	1:2	1:1

Napište program, ktorý spracuje údaje o zápasoch a na štandardný výstup vypíše meno (alebo mená) víťazov turnaja.

Riešenie

Budeme postupovať podľa schémy v štyroch krokoch (z projektu A): navrhne spôsob reprezentácie výsledku a potom premyslíme jeho výpočet (načítanie a spracovanie vstupu) a výpis.

Reprezentácia výsledku: Výsledok je meno víťazného hráča (resp. viacerých hráčov). Na určenie a vypísanie víťaza potrebujeme pre každého hráča poznať jeho meno a počet vyhratých zápasov a počet vyhratých kôl. Údaje o hráčovi združíme do štruktúry (**struct Hrac**). Údaje pre všetkých hráčov budeme pre jednoduchosť a dostupnosť v rôznych funkciách evidovať v globálnom poli (**h[]**), počet hráčov bude **n** (inicializovaný na 0). Vopred nevieme počet hráčov na turnaji, takže priebežne ako budeme získavať informácie o ďalších hráčoch budeme pole **h[]** zväčšovať.

Premenná **kapacita** zodpovedá počtu prvkov poľa **h[]**.

1	struct Hrac
2	{
3	char *meno;
4	int zapasy; // pocet vyhratych zapasov
5	int kola; // pocet vyhratych kol
6	};
7	
8	struct Hrac *h;
9	int n, kapacita; // pocet hracov, kapacita pola h

Výpis výsledku: Pri výpise výsledku musíme na základe údajov o všetkých hráčoch vypísať meno víťaza (víťazov).

Napíšeme funkciu **vypis_vitaza()**, ktorá prejde záznamy o hráčoch a určí hodnotu **max_zapasy** (koľko najviac zápasov niektorí z hráčov vyhral). Popri tom si bude udržiavať hodnotu **max_kola** (koľko najviac kôl vyhrali hráči, ktorí vyhrali najviac zápasov). V prípade, že nájdeme hráča, ktorý vyhral viac zápasov ako doteraz zistené číslo **max_zapasy**, tak hodnotu **max_kola** prepíšeme počtom vyhratých zápasov, ktoré vyhral aj keď by hodnota bola menšia ako aktuálna hodnota **max_kola**.

```

67 void vypis_vitaza()
68 {
69     int i, max_zapasy, max_kola;
70     if (n == 0)
71     {
72         printf("Na turnaji sa nikto nezúčastnil, víťaza niet.\n");
73         return;
74     }
75     // zisti kolko najviac zápasov (resp. kol) niekto vyhral
76     max_zapasy = h[0].zapasy;
77     max_kola = h[0].kola;
78     for (i = 1; i < n; i++)
79     {
80         if (max_zapasy == h[i].zapasy)
81         {
82             if (max_kola < h[i].kola)
83                 max_kola = h[i].kola;
84         }
85         else if (max_zapasy < h[i].zapasy)
86         {
87             max_zapasy = h[i].zapasy;
88             max_kola = h[i].kola;
89         }
90     }
91     // vypis hracov, ktorí dosiahli maximum
92     for (i = 0; i < n; i++)
93         if (h[i].zapasy == max_zapasy && h[i].kola == max_kola)
94             printf("%s\n", h[i].meno);
95 }
```

Načítanie vstupu a výpočet: Vstupné údaje sa nachádzajú vo viacerých súboroch, pričom súbory sú vzájomne nezávislé a môžeme ich spracovať jednotlivo. Načítanie a spracovanie údajov v jednom súbore preto implementujeme do samostatnej funkcie **nacitaj_zapas()**, ktorú budeme volať pre každý súbor. Spracovanie je riadené

z hlavnej funkcie **main()**, ktorá bude otvárať súbory s názvami **zapas_i.txt** pre **i** idúce od 1 do nekonečna, pričom pre každý úspešne otvorený súbor zavolá funkciu **nacitaj_zapas()**, ktorá údaje načíta a spracuje do záznamov o hráčoch (**h[]**). Predpokladáme, že sa turnaja zúčastnilo nenulový počet hráčov a preto alokujeme počiatočnú kapacitu poľa (**h[]**) pre 8 hráčov. Nakoniec funkcia vypíše víťaza (resp. víťazov). Implementácia hlavnej funkcie **main()** je takáto:

```
96  int main(void)
97  {
98      kapacita = 8;
99      h = (struct Hrac*) malloc(sizeof(struct Hrac) * kapacita);
100
101      int i;
102      char nazov[100];
103      for (i = 1; ; i++)
104      {
105          sprintf(nazov, "zapas_%d.txt", i);
106          FILE *f = fopen(nazov, "rt");
107          if (f == NULL)
108              break;
109          nacitaj_zapas(f);
110      }
111      vypis_vitaza();
112      return 0;
113  }
```

Zostáva nám napísať funkciu **nacitaj_zapas()** pre spracovanie jedného zápasu zo súboru. Funkcia bude súbor načítavať podľa špecifikácie formátu v zadaní úlohy.

Najskôr načítame a spracujeme mená hráčov, pre každého nájdeme podľa jeho mena index v poli hráčov **h[]**. Ak sa tam ešte nenachádza, tak sa ho pokúsime pridať. Pred pridaním musíme ešte skontrolovať, či pole **h[]** má dostatočnú kapacitu. Ak nie, tak pole zvážšime na dvojnásobnú veľkosť. Týmto zdvojením možno vyhradiť viac pamäte ako bude skutočne potrebné (napr. vyhradiť až 128 prvkov, ale hráčov bude nakoniec len 65). Množstvo použitej pamäte bude však k celkovému počtu hráčov najviac dvojnásobné, čo je vzhľadom na očakávané počty zanedbateľný nedostatok. Naopak, ak by sme pole zvážšovali vždy po jednom (o jedného nového hráča), tak by neúmerne (násobne) narástla časová náročnosť načítavania. Určenie indexu hráča v poli **h[]** podľa jeho mena implementujeme pomocnou funkciou **index_hraca()** takto:

```

10 int index_hraca(char *meno)
11 {
12     int i;
13     for (i = 0; i < n; i++)
14         if (!strcmp(h[i].meno, meno))
15             return i; // hrac s menom meno ma index i v poli h
16     // hraca s menom meno nemame zaevidovaneho
17     if (n == kapacita)
18     {
19         kapacita *= 2;
20         h = (struct Hrac*)realloc(h, sizeof(struct Hrac)*kapacita);
21     }
22     h[n].zapasy = 0;
23     h[n].kola = 0;
24     h[n].meno = strdup(meno);
25     n++;
26     return i;
27 }

```

Funkcia **nacitaj_zapas()** pokračuje načítaním údajov o jednotlivých kolách zápasu. Každé kolo sú dva reťazce: ťah prvého hráča a ťah druhého hráča. Pre vyhodnotenie víťaza kola podľa pravidiel hry implementujeme pomocnú funkciu **vitaz_kola()**:

```

28 int vitaz_kola(char *tah1, char *tah2)
29 {
30     if (!strcmp(tah1, tah2))
31         return 0; // nevyhral nikto
32     if (!strcmp(tah1, "kamen") && !strcmp(tah2, "noznice"))
33         return 1; // vyhral prvý hrac
34     if (!strcmp(tah1, "noznice") && !strcmp(tah2, "papier"))
35         return 1;
36     if (!strcmp(tah1, "papier") && !strcmp(tah2, "kamen"))
37         return 1;
38     return 2; // vyhral druhý hrac
39 }
40

```

Implementácia funkcie **nacitaj_zapas()** by mala byť teraz už priamočiara:

1. Načítame mená hráčov a pomocou funkcie **index_hraca()** určíme ich indexy v globálnom poli hráčov (**h[]**).
2. Načítame a spracujeme jednotlivé kolá zápasu až do konca súboru. Víťaza jedného kola vyhodnotíme funkciou **vitaz_kola()** a v globálnom poli **h[]** mu zvýšime počet vyhratých kôl (**kola**). Nakoniec, ak má zápas svojho víťaza, zvýšime mu počet vyhratých zápasov (**zapasy**).


```
41 void nacitaj_zapas(FILE *f)
42 {
43     char meno1[100], meno2[100], tah1[10], tah2[10];
44     fscanf(f, "%[^\n]\\n%[^\n]", meno1, meno2);
45     int i1 = index_hraca(meno1), i2 = index_hraca(meno2);
46     int k, pocet1 = 0, pocet2 = 0;
47
48     while (fscanf(f, "%s %s", tah1, tah2) == 2)
49     {
50         k = vitaz_kola(tah1, tah2);
51         if (k == 1)
52         {
53             pocet1++;
54             h[i1].kola++;
55         }
56         if (k == 2)
57         {
58             pocet2++;
59             h[i2].kola++;
60         }
61     }
62     if (pocet1 > pocet2) // vitaz zapasu je i1
63         h[i1].zapasy++;
64     if (pocet2 > pocet1) // vitaz zapasu je i2
65         h[i2].zapasy++;
66 }
```


Literatúra

1. HANLY, Jeri R.; KOFFMAN, Elliot B. *Problem solving and Program Design in C (Fifth edition)*. Pearson, 2012.
2. HORTON, Ivor. *Beginning C: From Novice to Professional (Fifth edition)*. Apress, 2013.
3. KELLEY, Al; POHL, Ira. *A book on C; Programming in C (Fourth edition)*. Benjamin-Cummings Publishing, 1994.
4. KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C programming language (Second edition)*. Prentice Hall, 1998.
5. PRATA, Stephen. *C++ primer plus (Fifth edition)*. Sams Publishing, 2004.
6. VAN DER LINDEN, Peter. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

Najnovší vývoj programovacieho jazyka C najlepšie približuje publikácia Jensa Gustedta, ktorý sa podieľa na nových úpravách a štandardizácii jazyka C:

7. GUSTEDT, Jens. *Modern C (draft)*. 2018.
Najnovšia verzia, ktorú autor postupne dopĺňa, je dostupná na URL adrese:
<http://icube-icps.unistra.fr/index.php/File:ModernC.pdf>